Université de Nice-Sophia Antipolis
École doctorale : Sciences pour l'ingénieur

Thèse

présentée pour obtenir le titre de

Docteur
(arrêté du 30 Mars 1992)
spécialité Informatique

par

Claude MICHEL

Modèles et Implémentations d'Interprètes Réflexifs

Soutenue le 5 décembre 1997 devant le jury composé de :

Jean-François PERROT, président et rapporteur
Danièle HÉRIN, rapporteur
Jacques MALENFANT, rapporteur
Pierre COINTE, examinateur
Jacques FERBER, examinateur
Olivier LECARME, examinateur
Michel RUEHER, examinateur
Jean-Pierre REGOURD, directeur de thèse

Remerciements

Je tiens particulièrement à remercier M. Jean-Pierre Regourd, Maître de conférences à l'université de Nice-Sophia Antipolis, de m'avoir guidé tout au long de mes recherches. Il a su, depuis mon stage de DEA, jouer ce rôle de guide tout en respectant mes idées. Par sa disponibilité, ses conseils, sa gentillesse, il a instauré un climat propice à l'élaboration de cette thèse.

Je remercie M. Jean-Fran cois Perrot, Professeur à l'université de Paris VI, de m'avoir fait l'honneur de rapporter sur cette thèse et d'en présider le jury. Je le remercie aussi des précieuses remarques dont il m'a fait part lors de la lecture de ma thèse. Il m'a ainsi notablement aidé à en améliorer le contenu.

Je remercie Mme Danièle Hérin, Professeur à l'université de Montpellier II, qui a su, lors de mon stage de DEA, m'insuffler le goût de la recherche, d'avoir accepté de rapporter sur cette thèse.

Je remercie M. Jacques Malenfant, Professeur à l'université de Bretagne-sud, de m'avoir fait l'honneur d'être rapporteur.

Je tiens particulièrement à exprimer ma gratitude à l'ensemble des rapporteurs qui ont accepté cette charge malgré d'importantes contraintes de temps.

Je remercie M. Pierre Cointe, Professeur à l'école des Mines de Nante, M. Jacques Ferber, Professeur à l'université de Montpellier II, M. Olivier Lecarme, Professeur à l'université de Nice-Sophia Antipolis, et M. Michel Rueher, Professeur à l'université de Nice-Sophia Antipolis qui m'ont fait l'honneur d'être membres du jury.

Je tiens à remercier M. Michel Rueher pour les nombreux conseils qu'il m'a donnés.

Je remercie M. Daniel P. Friedman, Professeur à l'Indiana University, qui lors de son passage à l'I3S, m'a fait la gentillesse de m'accorder un peu de son temps pour m'expliquer avec clarté et précision les fondements de la réflexivité. Il a, par son aide, stimulé mes recherches.

Je remercie M. Jean-Christophe Pazzaglia qui, par ses dialogues riches, ses critiques et ses propositions, a notablement influé sur mes recherches. Il a instauré au sein de notre équipe une dynamique sans laquelle ces années de travail n'auraient pas eu ce piquant qu'il leur a donné.

Je remercie M. Olivier Lhomme, Maître assitant à l'école des Mines de Nante, de m'avoir fait l'amitié de relire certaines parties de cette thèse. Par ses remarques pertinentes, il a contribué à en améliorer le contenu.

Je remercie le laboratoire I3S de m'avoir accueilli et de m'avoir fourni l'environnement nécessaire à tout travail de recherche.

Enfin, je remercie toute ma famille pour son soutien et sa patience ...

Table des matières

Table des figures

Chapitre 1

Introduction

La réflexivité, par la souplesse qu'elle confère à un système de calcul, introduit des possibilités dans la programmation qui vont jusqu'à la modification de la sémantique d'un langage depuis le système qui l'implémente. Elle donne à un système de calcul une certaine «conscience» de lui-même et le rend capable d'opérer sur lui-même à partir de sa propre représentation. Il devient ainsi plus facile d'élaborer des systèmes intelligents qui raisonnent à partir de leur propre état.

La réflexivité est une propriété d'un système de calcul. Elle ne constitue pas un tout en soi mais un apport à un système. Un système qui dispose de cette propriété est un système réflexif.

Un système réflexif est défini par son modèle et son implémentation. Un modèle de réflexivité définit la représentation qu'a le système de lui-même, les outils linguistiques qui permettent de manipuler la réflexivité ainsi que la relation qui lie les entités du système à leur représentation. Autrement dit, le modèle de réflexivité spécifie ce qui est explicité et comment c'est explicité. Ce faisant, il influe sur ce que permet ou ne permet pas la réflexivité.

Le choix d'une représentation du système est extrêmement délicat. Toutes les représentations ne sont pas équivalentes. Une représentation qui explicitera clairement certains aspects du système le fera en laissant implicites d'autres aspects. Or la représentation du système spécifie en grande partie ce que la réflexivité permet de faire. En outre, tout expliciter ne serait pas judicieux : certains aspects de l'implémentation doivent parfois rester implicites pour garantir une stabilité et une portabilité minimale du système et trop d'explicitations rendent difficile la prise en main du système.

Les outils linguistiques qui permettent la manipulation de la réflexivité fixent, eux aussi, ce qu'il est possible de faire et comment cela est possible. Certains fixent définitivement la relation entre une représentation et l'entité représentée dès sa création. D'autres offrent une manipulation plus dynamique de cette relation qui n'incite pas le programmeur à penser à l'utilisation de la réflexivité en même temps qu'il traite les programmes utilisateurs.

La relation qui lie les entités à leurs représentations traduit, elle aussi, les limites de la réflexivité. Certains modèles prônent une approche explicite de l'utilisation de la réflexivité tandis que d'autres sont axés sur une approche plus implicite. Une lambda réflexive ne s'utilise pas de la même façon qu'un méta-objet. L'état d'esprit dans lequel la réflexivité est utilisée est influencé par le caractère explicite ou implicite du lien qui lie une entité à sa représentation.

L'implémentation participe aussi à la définition d'un système réflexif. Un modèle de réflexivité ne spécifie pas tous les aspects de ce qu'est la réflexivité. Une grande partie des capacités réflexives d'un système de calcul passe par l'implémentation de la réflexivité. En particulier, la connexion causale entre une entité du système et sa représentation est pleinement réalisée par l'implémentation.

L'implémentation des systèmes réflexifs diffère d'un système à l'autre. Elle est fortement liée aux objectifs assignés au système. Un système «tout réflexif» ne s'implémente pas de la même façon qu'un système uniquement doté de quelques capacités réflexives. Le modèle joue un grand rôle dans la liberté laissée à l'implémentation. Mais il reste influencé par ce que l'implémentation peut ou ne peut pas faire.

Les systèmes réflexifs sont souvent regardés, à raison, comme inefficaces. L'architecture de tour d'interprètes qui régit ces systèmes est la principale cause de cette inefficacité. Cette réputation d'inefficacité a longtemps nuit à faire de la réflexivité un concept répandu. La réflexivité ne se répandra que lorsque ce problème de performance sera résolu, tout au moins assez pour que ses apports priment sur les performances.

Ces problèmes reviennent tout au long de cette thèse, tant dans l'étude préliminaire de la réflexivité que dans la réalisation de notre objectif final, la définition et l'implémentation d'un modèle de réflexivité pour un interprète d'un langage à objets concurrents basé sur le modèle de calcul défini par G. Agha [Agha 86]. Il existe d'autres interprètes réflexifs basés sur les acteurs, mais aucun n'a satisfait à l'ensemble de nos critères. Dans le cadre des recherches effectuées par notre équipe, nous voulions un interprète réflexif qui soit à la fois simple à utiliser, avec un champ d'action étendu et dont l'efficacité de l'exécution soit compatible avec une utilisation régulière.

Notre approche s'appuie sur deux piliers essentiels: une étude poussée de la réflexivité et un dialogue permanent avec un utilisateur averti, J. C. Pazzaglia [Pazzaglia 97].

L'étude de la réflexivité est basée sur les interprètes de langages fonctionnels. Les langages fonctionnels, historiquement les premiers à avoir bénéficié du concept de réflexivité, ont toujours constitué une plate-forme idéale à l'expérimentation des mécanismes linguistiques. Ils permettent de s'abstraire des contingences liées à l'implémentation d'interprètes pour se focaliser sur les problèmes essentiels. Ils se sont révélés suffisants pour clarifier les principes fondamentaux des interprètes réflexifs et dégager les problèmes essentiels posés par la réflexivité. Ainsi, au delà de notre étude des interprètes réflexifs existants, nous avons pu clarifier le concept de réflexivité globale et étudier en détail l'utilisation de l'évaluation partielle dans le cadre de la réflexivité. Ces deux études nous ont amenées à écarter la réflexivité globale et l'évaluation partielle de notre interprète final qui sont mal adaptées à l'efficacité du système et au contexte des langages d'acteurs.

À partir de cette étude, nous définissons et implémentons un interprète réflexif pour un langage d'acteurs, LActE/R. La définition du modèle s'appuie non seulement sur l'étude de la réflexivité mais aussi sur un dialogue riche avec un utilisateur avancé, J. C. Pazzaglia. Par ses critiques et ses propositions, basées sur son utilisation des différentes versions de LActE/R, J. C. Pazzaglia a influencé le modèle de réflexivité en nous permettant de le rationaliser et de l'affiner.

Cette thèse se découpe en deux parties. La première est constituée de l'étude de la réflexivité. La seconde partie, du modèle et de l'implémentation de LActE/R. Dix chapitres composent cette thèse.

La première partie regroupe l'étude de la réflexivité. Elle est composée des quatre chapitres suivants:

- Le chapitre 2 fixe certains concepts propres à la réflexivité et certains éléments de vocabulaire. Après avoir donné une définition de la réflexivité, nous énonçons les principaux concepts liés à ce domaine.

- Le chapitre 3 propose une étude de la réflexivité dans les langages fonctionnels à travers ses principales caractéristiques. Elle est basée sur un ensemble complet d'interprètes réflexifs de langages fonctionnels.

- Le chapitre 4 adresse le problème de la réflexivité globale à travers l'implémentation d'un interprète muni d'une tour active. Nous y montrons que la présence de la réflexivité globale dépend de l'implémentation de la connexion causale. Une série d'exemples illustre l'utilité de la réflexivité globale, en particulier son aptitude à s'affranchir des contraintes liées à une représentation donnée.

- Le chapitre 5 montre comment l'évaluation partielle résout le problème des performances d'un système réflexif. L'évaluation partielle est souvent présentée comme une technique idéale pour résoudre la question de l'efficacité dans les interprètes réflexifs. L'interprète implémenté offre une vision précise de ce que l'évaluation partielle apporte à la question de l'efficacité mais aussi, souligne les problèmes liés à l'utilisation de cette technique.

La seconde partie de cette thèse regroupe la définition et l'implémentation de notre interprète réflexif pour un langage d'acteurs. Elle est constituée des quatre chapitres qui suivent:

- Le chapitre 6 constitue une étude de la réflexivité dans les langages à objets. Les principaux modèles de réflexivité sont examinés

- Le chapitre 7 décrit le modèle de réflexivité de LActE/R, une couche acteur au dessus de Common Lisp. Le modèle de réflexivité de LActE/R est défini à partir d'un principe fondateur, le principe de localité. Ce dernier influence le modèle par ses aspects dynamiques, le nombre de niveaux contenus dans une tour étant contrôlé dynamiquement par deux opérations explicites, la réification et la réflexion, mais aussi sur ce qu'il est possible de réifier. Le modèle de réflexivité de LActE/R se traduit par une réification localisée aux seules entités manipulées explicitement par l'utilisateur. Ainsi, à chaque méta-acteur est associé un réificateur qui permet de réifier une à une les instructions introduites par la couche acteur.

- Le chapitre 8 décrit les points les plus critiques de l'implémentation du modèle de réflexivité de LActE/R. Le traitement de l'uniformité linguistique ainsi que les implémentations des messages et méthodes du méta, des opérations de réification et de réflexion, des objets de LActE, ainsi que des instructions y sont abordés. Le point le plus délicat de l'implémentation de LActE/R, l'implémentation des opérations de réification et de réflexion y est décrite en détail.

- Le chapitre 9 propose plusieurs exemples d'utilisation de LActE/R. Ces exemples, augmentés de ceux présentés dans la thèse de J. C.Pazzaglia, permettent de valider le modèle de réflexivité de LActE/R. En outre, ils montrent les possibilités offertes par la réflexivité.

Le chapitre 10 constitue la conclusion de cette thèse.

Les différents chapitres qui composent la première partie de cette thèse sont relativement indépendants les uns des autres. Il n'est pas nécessaire d'avoir lu l'un pour comprendre l'autre. Les trois derniers chapitres de la seconde partie sont cependant assez liés. Notons aussi que le premier chapitre de chacune des parties ainsi que le chapitre 2 constituent un matériel d'introduction à la lecture des autres chapitres.

Première partie

Étude de la réflexivité

Chapitre 2

Définitions et concepts

Le concept de réflexivité, né au début des années 80 [Smith 83], est devenu au fil du temps un vaste champ de recherche. La multiplicité des travaux effectués, les nombreux domaines auxquels ce concept a été appliqué, ont produit un foisonnement de résultats caractérisés par des discordances sur les concepts de bases attachés à la réflexivité. La définition même de la réflexivité varie d'un chercheur à l'autre traduisant la diversité des approches et des conceptions. Aussi nous ne pouvons pas nous engager plus en avant dans cette thèse sans avoir auparavant clarifié les concepts qui servent de fondement à notre travail.

Dans ce chapitre, après avoir donné la définition de la réflexivité, nous aborderons un à un les principaux concepts utilisés dans le cadre de la réflexivité. Il ne s'agit pas pour nous d'être exhaustif, mais d'en offrir une vue aussi claire et précise que possible.

2.1 Définitions

2.1.1 Système de calcul

Bien que le domaine principal d'application de la notion de réflexivité soit les langages, son champ d'application est beaucoup plus vaste. Il existe des systèmes de fenêtrage réflexifs [Rao 91], des systèmes d'exploitation réflexifs [Yokote 92], des systèmes de travail en coopération réflexifs [Dourish 92]. Le dénominateur commun de tous ces exemples est d'être des systèmes de calcul.

Un *système de calcul* S_c peut être vu comme étant la donnée de trois éléments: un exécutant E à même de rendre actif le système, un programme P et les données D auxquelles le programme s'applique (Fig. 2.1 page suivante). On a alors:

$$S_c = E \times P \times D$$

La raison d'être d'un système de calcul est le traitement qu'il opère sur un *domaine de calcul*. A travers ses trois composantes, le système de calcul représente un sous-ensemble d'un monde habituellement externe à lui-même. Par exemple, le traitement de texte qui sert à

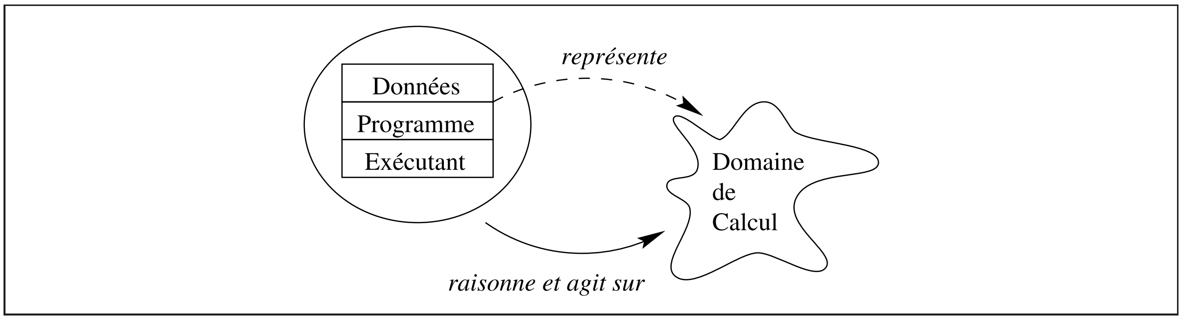

FIG. 2.1 – *Un système de calcul*

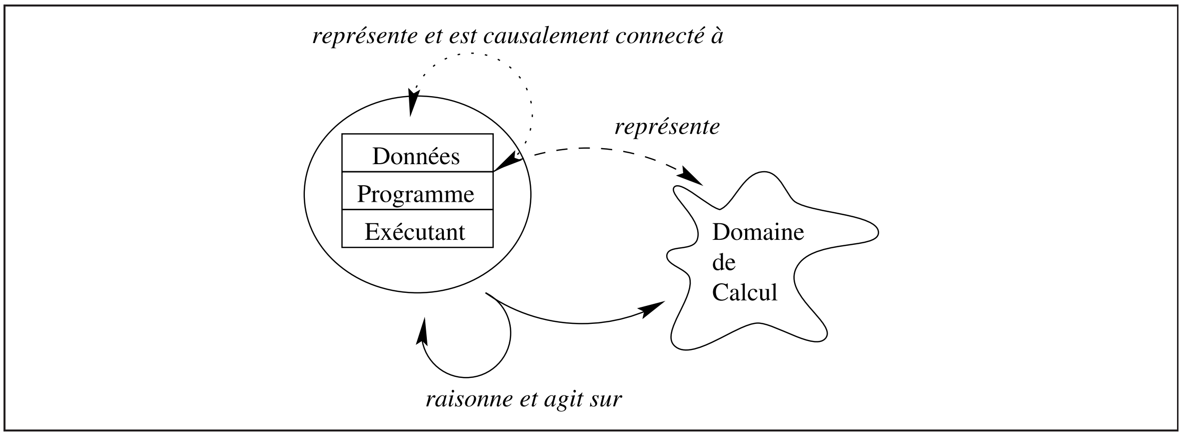

FIG. 2.2 – *Un système réflexif*

écrire ce texte a pour domaine de calcul ce texte.

2.1.2 Système réflexif

Nombreuses sont les définitions de la réflexivité. Chacune d'entre elles apporte ses éclaircissements sur cette notion. La première définition, celle de B.C. Smith est:

«A computer system able to reason about itself ...» [Smith 83]

Un système de calcul S_c réflexif peut donc être vu comme un système capable de se prendre pour objet de son propre calcul (Fig. 2.2). Soit $D_c(S_c)$ le domaine de calcul de S_c on a alors:

$$D_c(S_c) \supset S_c$$

Un système de calcul réflexif peut par conséquent s'examiner lui-même et agir sur lui-même de la même façon qu'il examine, observe et agit sur un domaine de calcul qui lui est externe.

Cette définition est quelque peu idéaliste. En particulier, elle est très contraignante puisqu'elle sous-entend que la totalité du système de calcul est rendue accessible. Or la réalité des différentes implémentations disponibles est toute autre: seule une partie, plus ou moins importante, du système de calcul appartient au domaine de calcul. Dans un tel cas, une terminologie plus appropriée qualifierait ces systèmes de systèmes munis de *capacités réflexives*.

Dans la suite de cette thèse l'acception du qualificatif «réflexif» sera étendue aux systèmes de calcul munis de capacités réflexives pour lesquels une très large partie du système de calcul est mise à la disposition de l'utilisateur. «Capacités réflexives» sera lui restreint aux langages pour lesquels seules quelques constructions linguistiques permettent d'accéder à un sous-ensemble très restreint du système de calcul. Une frontière précise entre les deux types de systèmes est difficilement définissable. Si, pour nous, le premier terme recouvre la quasi totalité des systèmes étudiés pour être réflexifs et le second quelques langages de hauts niveaux dotés d'une ou plusieurs constructions linguistiques permettant d'accéder à certains aspects du système, d'autres classifications restent possibles.

2.2 Aspects d'un système réflexif

2.2.1 Auto-représentation d'un système

L'aptitude d'un système de calcul S_c à se prendre comme objet de son propre calcul nécessite l'existence d'une représentation de S_c au sein de S_c. Autrement dit, S_c doit avoir des capacités d'*auto-représentation*. C'est à travers cette représentation qu'il a de lui-même que le système est capable de s'observer de la même manière qu'il observe un domaine de calcul externe.

De la représentation choisie du système de calcul vont dépendre les possibilités offertes par la réflexivité. L'adoption d'une représentation trop limitée par les aspects du système qu'elle explicite autant que par la manière dont elle les représente réduit considérablement les possibilités réellement offertes par la réflexivité. L'article de J. W. Simmons II [Simmons II 92b] est très éloquent à ce sujet. Il montre comment le choix d'une représentation influe sur les capacités d'un système réflexif à résoudre un problème donné.

Il ne faut pourtant pas croire qu'il suffit d'offrir une représentation aussi détaillée que possible du système pour étendre les applications de la réflexivité. Outre qu'une représentation trop détaillée se traduit par une complexité accrue de la programmation de la réflexivité, choisir une représentation c'est s'en interdire une autre. L'utilisation des fermetures lexicales pour représenter l'environnement de l'interprète dans Brown [Friedman 84] rend difficiles certaines opérations plus aisées sur les paires pointées habituellement utilisées.

Le problème est donc double. Pour doter un système de calcul de capacités réflexives, il faut non seulement introduire une représentation du système en son sein, ce qui n'est pas trivial, mais aussi définir une représentation apte à permettre la manipulation du système. Le choix de ce qui est rendu explicite est tout aussi important et délicat que la façon dont cette explicitation est faite.

2.2.2 Lien causal

L'existence de capacités d'auto-représentation au sein d'un système de calcul ne suffit pas à le rendre réflexif. La représentation du système doit, d'une part, être le reflet de l'état du système, toute modification du système devant entraîner une modification conséquente de sa représentation, mais aussi, et de façon symétrique, l'état du système doit être lié à sa représentation, toute modification de la représentation devant entraîner une modification conséquente du système. Dans un tel cas de figure, le système et sa représentation sont qualifiés de *causalement connectés*.

Le *lien causal*, ou encore *connexion causale*, — le lien qui existe entre deux entités causalement connectées — donne tout son sens à la réflexivité. Ce lien permet à un système réflexif d'agir *effectivement* sur lui-même en tenant compte de l'état dans lequel il est. C'est dans l'existence de la connexion causale que réside, selon nous, l'essence de la réflexivité.

En l'absence d'un tel lien, le système est incapable d'agir sur lui-même ou de raisonner sur lui-même, soit que les modifications de sa représentation n'induisent pas de modifications conséquentes du système, soit que la représentation du système ne soit pas en accord avec l'état courant du système. Il ne serait être de système réflexif sans connexion causale.

2.2.3 Architecture des systèmes réflexifs

Dès l'introduction de 3-Lisp [Smith 83], une architecture des systèmes réflexifs a pu être définie. Elle se base sur la notion de *tour virtuellement infinie de systèmes*. Chaque niveau de la tour est constitué d'un système chargé d'interpréter le système qui se trouve au niveau directement en dessous du niveau courant, la base de la tour, le niveau numéroté 0, étant constitué par le programme défini par l'utilisateur (Fig. 2.3 page ci-contre).

Une telle tour est virtuellement infinie, le nombre de systèmes contenus dans la tour n'étant pas fixé à priori. Des techniques d'implémentation permettent de rendre cette tour virtuellement infinie. Les deux techniques habituellement utilisées sont l'évaluation paresseuse et l'auto-référence du sommet de la tour. Toutes deux intègrent le caractère virtuellement infini de la tour à un système fini calculable sur un ordinateur.

L'architecture des systèmes réflexifs traduit la définition d'un système réflexif en terme de tour de systèmes de calcul. Ce faisant, elle introduit une distorsion entre l'implémentation de ces systèmes et leur définition. Elle traduit la relation circulaire contenue dans la définition, un système réflexif étant un système qui effectue un calcul sur lui-même, en une relation linéaire, la tour de systèmes qui se calculent l'un l'autre. Cette distorsion introduit les premières limitations des systèmes réflexifs qui sont obligés de se voir comme un objet externe à eux-mêmes au moment où ils raisonnent et agissent sur eux-mêmes.

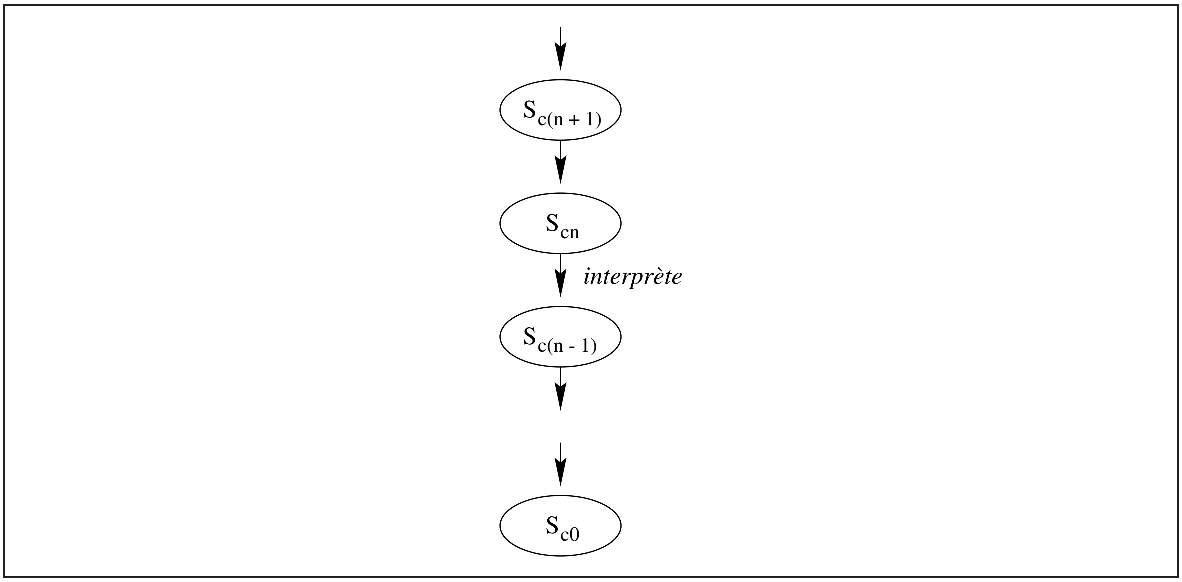

FIG. 2.3 – *Une tour virtuellement infinie de systèmes*

2.2.4 Désignation des systèmes au sein de la tour

Au sein de la tour de systèmes de calcul, les systèmes sont hiérarchisés par la relation d'interprétation qui les lie. Un système de niveau n, S_{cn}, est interprété par un système de niveau $(n+1)$, $S_{c(n+1)}$, et interprète le système de niveau $(n-1)$, $S_{c(n-1)}$. S_{cn} est donc à la fois interprète d'un système et interprété par un autre système.

Le système S_{cn} interprété par le système $S_{c(n+1)}$ est désigné comme étant la *dénotation* de ce dernier. De façon symétrique, le système $S_{c(n+1)}$ est désigné comme étant le *méta* de S_{cn} (Fig. 2.4 page suivante). Dans le cadre des systèmes de calcul, le méta sera souvent un *méta-système*.

La relation telle qu'elle est décrite ici est bijective c'est à dire qu'à une dénotation elle fait correspondre un et un seul méta et qu'à un méta elle fait correspondre une et une seule dénotation. Dans leur diversité, les systèmes réflexifs utilisent d'autres types de relations méta/dénotation. Les langages de classes s'illustrent par un «méta», la méta-classe, qui sert de «méta» à plusieurs «dénotations», les classes. Une relation méta/dénotation existe pourtant toujours sous une forme ou une autre dans les systèmes réflexifs.

2.2.5 Passage d'un niveau à l'autre

Un système réflexif est composé d'une tour de systèmes qui se calculent l'un l'autre. Chaque niveau de la tour qui calcule le niveau qui se trouve en dessous de lui, sa dénotation, doit, au moment où il calcule cette dénotation, disposer d'une représentation de cette dernière. Les informations qui concernent la dénotation ne sont parfois disponibles pour le méta-système que sur sa demande. Un système réflexif doit permettre d'obtenir ces informations mais aussi, pour respecter la connexion causale, permettre d'intégrer les informations manipulées par le

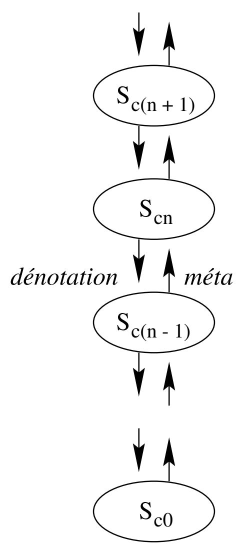

FIG. 2.4 – *Méta et dénotation dans une tour de systèmes*

méta-système dans la dénotation. Ces deux opérations se nomment respectivement *réification* et *réflexion*.

La *réification* est l'opération qui consiste à rendre explicites certains aspects du système afin que son méta-système puisse les manipuler. La *réflexion* est l'opération inverse de la réification : elle redonne à une représentation explicite contenue dans le méta-système un caractère implicite en l'intégrant au système.

Lorsqu'un objet du langage aura subi une opération de réification, on dira qu'il est *réifié*. Il est alors manipulable par les programmes écrits par l'utilisateur.

Seuls certains systèmes réflexifs disposent explicitement de telles opérations. La plupart du temps, ces opérations sont réalisées une fois pour toute. Le méta-système implémente alors sa dénotation qui n'existe plus qu'au travers de sa représentation. D'autres systèmes réflexifs préfèrent offrir un système de réification et de réflexion à la demande. Des choix philosophiques se cachent souvent sous la présence ou l'absence de ces opérations, mais aussi des questions comme celle de l'efficacité de l'exécution des systèmes réflexifs.

2.3 Réflexivité locale et réflexivité globale

Lorsque l'utilisateur d'un système réflexif le programme, la portée des modifications qu'il apporte au système peut être plus ou moins grande. Il peut vouloir contrôler le mécanisme d'une entité qu'il a lui-même introduite ou encore celle d'une entité qui appartient au système. Dans ce dernier cas, si la connexion causale est correctement réalisée, la modification de cette entité du système induira une modification globale du comportement du système.

Si la totalité des systèmes réflexifs sont construits pour permettre la modification des entités définies par l'utilisateur, tous ne permettent pas de modifier globalement un système. Il nous est apparu important de distinguer les systèmes dotés de cette dernière capacité des premiers. Aussi, lorsque la réflexivité d'un système permet de modifier localement ses mécanismes, nous parlerons de *réflexivité locale*. Lorsque cette réflexivité permettra la modification globale des mécanismes du système de calcul, nous qualifierons cette réflexivité de globale et parlerons donc de *réflexivité globale*.

La réflexivité globale est rarement présente. Elle joue cependant un rôle important dans ce que la réflexivité permet de faire ou de ne pas faire. Son coût, au niveau de l'exécution du système, est souvent important et sert de justificatif à son absence. Un autre de ses désavantages est qu'une modification globale des mécanismes du système peut avoir pour conséquence un disfonctionnement global du système. On dit alors que la *méta-stabilité* du système n'est plus assurée. Cependant, afin de mieux comprendre ce qui est alors perdu, nous montrerons son rôle dans l'un des chapitres de cette thèse.

2.4 Programmation de calculs réflexifs

La réflexivité s'utilise avec des programmes qui effectuent des calculs réflexifs. Ces programmes sont écrits par l'utilisateur du système. Ce qui sépare de tels programmes des programmes habituels est l'objet de leur calcul qui n'est plus un domaine externe mais le système lui-même.

La programmation de calculs réflexifs diffère d'un système réflexif à l'autre. Elle dépend des outils linguistiques que la réflexivité met à disposition du programmeur. Il n'existe malheureusement pas à ce jour de modèle unifié de la réflexivité.

Cette programmation peut procéder par ajouts de nouveaux mécanismes, en particulier pour les systèmes dotés d'une réflexivité locale, ou par modification des mécanismes préexistant, pour les systèmes dotés d'une réflexivité globale. L'ajout de mécanismes est la forme la plus courante de programmation de système de calcul. Plus facile à mettre en œuvre, elle ne remet pas en cause la méta-stabilité du système et ne provoque que des perturbations locales du fonctionnement du système.

Dans la suite de cette thèse, pour parler de programmes qui effectuent des calculs réflexifs, nous utiliserons indifféremment les expressions *méta-programme* et *programme réflexif*. Pour désigner la tâche qui consiste à écrire de tels programmes nous utiliserons les expressions *programmation de la réflexivité* et *programmation réflexive*, bien que cette dernière soit impropre.

2.5 Axes de réflexivité

La modélisation d'un système de calcul fait apparaître plusieurs axes qui correspondent aux éléments utilisés par le système pour effectuer son calcul. Nous avons vu précédemment qu'un système de calcul pouvait être vu comme la donnée du triplet $(P \times D \times E)$. L'aptitude d'un système réflexif à rendre compte de l'un des éléments de ce triplet est ce que l'on nomme *axe de réflexivité*.

B. C. Smith [Smith 83] ne distingue que deux axes de réflexivité. Le premier de ces axes, la *réflexivité structurelle*, concerne la manipulation des données utilisées par le système. Le second axe, la *réflexivité comportementale*, concerne la partie programme du triplet. C'est souvent à ces deux axes que se réfèrent la plupart des articles qui discutent de réflexivité.

P. Carle [Carle 92] propose de distinguer quatre axes de réflexivités:

1. La *réflexivité structurelle* qui concerne la mise à disposition des structures utilisées par le système pour représenter les objets du langage.

2. La *réflexivité comportementale* qui est la mise à disposition sous forme de données d'un programme.

3. La *réflexivité opératoire* qui concerne l'utilisation des ressources par un programme pour son exécution.

4. La *réflexivité exécutive* qui concerne les interactions entre les objets du langage. Cet axe de réflexivité intéresse plus particulièrement les langages à objets où chaque objet communique avec ses congénères sous la forme d'envois de messages.

Pour comprendre les distinctions qu'apporte P. Carle, il faut les restituer dans le contexte de sa thèse. Cette dernière porte sur l'implémentation d'un langage pour l'intelligence artificielle distribuée Mering IV [Ferber 89, Ferber 91, Carle 92] où la notion d'interactions entre agents prend une place importante. Dès lors, les différents axes définis par P. Carle sont le reflet de ces préoccupations. Bien que ces axes se retrouvent dans beaucoup de langages, leurs définitions montrent la très grande disparité qui subsiste des conceptions de la réflexivité.

2.6 L'uniformité linguistique

La représentation d'une dénotation au sein d'un méta-système peut prendre différentes formes. Lorsqu'un système réflexif utilise pour se représenter le même langage que celui dont il se sert pour représenter un domaine de calcul externe à lui-même, on parle d'*uniformité linguistique*.

L'uniformité linguistique est souvent tenue pour l'un des caractères essentiels d'un système réflexif. Elle n'est pourtant pas ce qui rend le système réflexif. Il est possible d'imaginer un système réflexif qui changerait de système de représentation à chaque niveau de sa tour. Cependant, pour des raisons évidentes de simplicité, la plupart des systèmes réflexifs utilisent une même représentation de la dénotation à chaque niveau de la tour, représentation qui respecte les concepts utilisés pour représenter un domaine de calcul externe au système.

2.7 Système à implémentation ouverte

G. Kiczales introduit dans [Kiczales 93] la notion de système à implémentation ouverte. Il s'agit d'une notion élargie du concept de réflexivité introduite pour englober son concept de «réflexivité» basée sur un compilateur en coupant court au débat sur ce qui est qualifiable de réflexif et ce qui ne l'est pas. Bien que cette notion englobe les systèmes réflexifs, nous la regardons comme désignant les approches basées sur les compilateurs.

Les systèmes à implémentation ouverte que décrit G. Kiczales sont basés sur la mise à disposition du programmeur du compilateur d'un langage. La programmation du calcul réflexif y est obtenue par la modification du compilateur. Une fois le compilateur modifié, l'ensemble des programmes qui doivent refléter les modifications effectuées doivent être recompilées. L'approche est donc plus statique que dans les modèles basés sur un interprète.

Un système réflexif est, dans cette thèse, basé sur l'approche interprétative. Par là nous entendons les systèmes réflexifs qui modélisent leurs calculs en les décrivant à l'aide d'un modèle interprétatif par opposition aux systèmes à implémentation ouverte dans lesquels le calcul est décrit par un modèle compilatoire. Ces derniers ne sont pas moins «réflexifs» que les premiers mais l'utilisation d'un compilateur complexifie la tâche du programmeur.

2.8 Conclusion

Dans ce premier chapitre nous avons voulu clarifier le vocabulaire que nous allons utiliser par la suite. Ce qui nous est apparu important ici c'est de donner le sens précis des expressions que nous utilisons dans cette thèse, non pas pour introduire un nouveau vocabulaire ou un sens nouveau, mais pour que le lecteur puisse comprendre ces expressions telles que nous les avons comprises.

Les systèmes de calcul sur lesquels portent cette thèse sont des interprètes. Nous parlerons par la suite d'interprète réflexif plutôt que de système de calcul réflexif.

Chapitre 3

Réflexivité dans les langages fonctionnels

Dans ce chapitre, la réflexivité est étudiée à travers les interprètes de langages fonctionnels. Cette restriction à ces seuls langages trouve ses causes dans l'histoire de la réflexivité — ce sont les premiers a avoir bénéficié de ce concept — et dans les qualités de Lisp et Scheme. Ces derniers offrent les caractéristiques nécessaires à l'étude des mécanismes linguistiques et permettent de se focaliser sur les problèmes les plus importants.

3.1 Les interprètes réflexifs de langages fonctionnels

Cette étude de la réflexivité dans les langages fonctionnels s'appuie sur un ensemble complet d'interprètes réflexifs. Depuis 3-Lisp [Smith 83], l'initiateur du concept de réflexivité, de nombreux interprètes réflexifs de langages fonctionnels ont vu le jour. Chacun d'entre eux a apporté sa contribution à la compréhension du concept de réflexivité. Ce que nous voulons dans cette section, c'est donner un bref aperçu des principaux apports de chaque interprète réflexif.

Avec Brown84 [Friedman 84], D. P. Friedman et M. Wand montrent que la tour d'interprètes n'a pas besoin d'exister réellement dans un interprète réflexif. C'est aussi eux qui introduisent la vision des lambdas réflexives comme mécanisme d'introduction de formes spéciales.

La seconde version de Brown, Brown88 [Wand 88], ajoute à la première une tour d'interprètes ainsi qu'un constructeur de lambdas réflexives qui agit par transformation de lambdas classiques, le *make-reifier*.

Dans Blond [Danvy 88], la tour est construite non plus en partant de son implémentation mais en partant de son concept philosophique. Blond introduit les réificateurs γ et δ qui capturent dans la lambda réflexive un environnement différent de l'environnement lexical de leur définition. Pour finir, Blond associe un environnement global différent à chaque niveau d'interprétation. Les autres interprètes réflexifs partagent l'environnement global entre tous les niveaux d'interprétation.

Stepper [Bawden 88] identifie l'indépendance du concept de réflexivité avec le modèle d'évaluation. En particulier, il fait remarquer qu'il n'est pas obligatoire d'utiliser l'évaluation des expressions comme support à l'interprétation ni le triplet (e r k)[1] comme vecteur de paramètres des lambdas réflexives.

Black [Asai 93] est le premier à utiliser l'évaluation partielle pour rendre un interprète réflexif plus efficace. L'évaluation partielle y est utilisée pour spécialiser les interprètes méta-circulaires afin de les rendre plus efficaces. Ce procédé n'est utilisé que lors de la construction de l'interprète réflexif. Les constructions réflexives introduites par l'utilisateur ne bénéficient pas de cette approche.

I_r[Jefferson 92] montre comment construire un interprète réflexif basé sur une tour finie à partir d'un empilement d'interprètes classiques. Il construit un interprète muni d'une tour finie d'interprètes méta-circulaires en partant d'un interprète non-réflexif. En exposant sa démarche, il contribue ainsi à la clarification du rôle de la tour d'interprètes ainsi que de la construction d'interprètes réflexifs.

Refci [Simmons II 92a, Simmons II 92b, Simmons II 93] introduit une construction spécifique pour gérer la réflexivité globale. Il montre l'importance de ce type de réflexivité. Il montre aussi l'importance du modèle de représentation ainsi que les conséquences du modèle sur ce que la réflexivité permet ou ne permet pas de faire.

3.2 Réflexivité et langages fonctionnels

Les langages fonctionnels, les premiers à avoir bénéficié de la réflexivité, ont servi de support à l'étude et à la compréhension de ce concept. Les travaux de B.C. Smith [Smith 83] portent d'ailleurs sur l'introduction du concept de réflexivité dans ce paradigme linguistique.

Parmi les langages de ce type, Lisp [Steele jr 90], l'un des plus anciens, a été pourvu, dès sa naissance, d'outils qui permettent de manipuler quelques aspects structuraux et opératoires du langage. Les listes de Lisp fédèrent sous une forme unique la représentation des objets du langage manipulables par l'utilisateur. Les lambdas de Lisp étaient elles même initialement représentées sous forme de listes réalisant ainsi l'équation *programmes = données*. Common Lisp [Steele jr 90] conserve cette aptitude grâce à la fonction *function-lambda-expression*. Cette dernière rend possible l'écriture d'un mécanisme de trace qui opère en modifiant la définition de la fonction à tracer:

```
(defun trace-fct(name)
  (let* ((fct (symbol-function name))
         (compiled? (compiled-function-p fct)))
    (multiple-value-bind (lambda-expr is-closed? name)
                           ; Récupère la définition de la fonction sous forme de liste
                           (function-lambda-expression fct)
```

1. Notation souvent utilisée pour (expression environnement continuation).

```
      (setf (symbol-function name)
              ; Modification de la définition de la fonction à tracer
          (eval `(lambda ,(cadr lambda-expr)
                   (format t "~&~a called with ~a" ',name (list ,@(cadr lambda-expr)))
                   ,@(cddr lambda-expr)))))
   (when compiled? (compile name))))
```

Ce genre de mécanisme est à rapprocher de la réflexivité structurelle. Common Lisp est aussi doté des mécanismes *evalhook* et *applyhook* qui permettent, dans une certaine mesure, la modification du processus d'interprétation.

Scheme [Abelson 91] va au delà de Lisp dans ces mécanismes. Il offre un outil qui s'est révélé être un apport majeur dans la flexibilité du langage: le *call/cc*. En donnant accès à la continuation du calcul, cette fonction permet la construction de nouvelles structures de contrôle du flot de calcul. Ce type de mécanismes ne concerne plus les aspects structuraux du langage mais ses aspects opérationnels.

Par ailleurs, et en dehors de la norme R4RS, la plupart des implémentations de Scheme sont dotées de fonctions qui permettent de manipuler d'autres aspects de l'interprète comme l'environnement d'évaluation. Lorsqu'une implémentation de Scheme, comme STk de E. Gallesio [Gallesio 96], est en plus dotée des macros, elle dispose alors de tous les outils nécessaires à la construction de lambdas qui ressemblent fortement à des lambdas réflexives:

```
; Un compound-to-reifier en STk
(define-macro (compound-to-reifier lambda-decl)
  (let ((params (cadr lambda-decl))
        (body (cddr lambda-decl)))
    `(macro arguments
       (list 'apply
          (list
           (list 'lambda (list ',(cadr params))
                (list 'lambda ',(car params)
                     (list 'call/cc
                          (cons 'lambda (cons (list ',(caddr params))
                                              ',body)))))
           '(the-environment))
          (list 'quote (cdr arguments)))
       )))
```

```
; Exemple d'utilisation
(define bound??
  (compound-to-reifier
   (lambda ((var) env cont)
     (cont (if (assoc var (car (environment->list env))) #t #f)))))
```

```
STk> (let ((b 2)) (bound?? a))
#f
STk> (let ((a 2)) (bound?? a))
#t
STk>
```

L'écriture d'un *compound-to-reifier* en STk est rendue possible par l'extension de l'interprète vers les macros et l'introduction de fonctions qui manipulent l'environnement. Une telle fonction, si elle permet déjà de s'approcher d'un interprète réflexif, n'autorise pourtant pas toutes les manipulations rendues possibles par la réflexivité. L'environnement et la continuation utilisent des représentations spécifiques qui en limitent l'utilisation. En particulier la continuation ne peut être transformée en donnée. La présence quasi systématique de telles extensions est cependant révélatrice du besoin de réflexivité.

Les langages de la famille Lisp offrent dès leur naissance des capacités linguistiques qui les rapprochent des interprètes réflexives. Ils ont de tout temps été la plate-forme idéale pour l'expérimentation de nouveaux concepts dans le domaine des langages informatiques. Et c'est naturellement qu'ils ont servi de plate-forme à l'expérimentation du concept de réflexivité.

La réflexivité dans les langages fonctionnels s'inscrit dans le monde de l'interprétation. Elle pousse les mécanismes des langages comme Lisp à ses extrêmes aboutissant à des interprètes réflexifs. La démarche suivit par Smith lors de l'élaboration de 3-Lisp est à ce sujet éloquente: partant de Lisp, elle passe par une phase d'épuration de la sémantique conduisant à 2-Lisp avant de construire l'interprète réflexif.

3.3 Architecture des interprètes réflexifs de langages fonctionnels

Du point de vue du modèle, les interprètes réflexifs de langages fonctionnels adoptent tous le même modèle d'architecture. Il consiste à décomposer la relation d'auto-référence propre à la définition de la réflexivité en un empilement d'interprètes formant une tour virtuellement infinie.

Le premier niveau de la tour constitué par les programmes de l'utilisateur est numéroté 0, le suivant 1, ainsi de suite jusqu'à l'infini. Les différents niveaux d'interprétation de la tour sont alors liés par la relation suivante: l'interprète de niveau $(n+1)$ interprète celui de niveau n, pour tout n supérieur ou égal à zéro.

La tour constituée de cet empilement d'interprètes est virtuellement infinie. La régression infinie de la tour est résolue par l'évaluation paresseuse. Les interprètes réflexifs sont en effet construits de telle manière qu'il existe toujours un niveau n d'interprétation à partir duquel les interprètes sont identiques, par leurs définitions. Cette identité permet d'implémenter l'ensemble du sommet de la tour i.e. les interprètes de niveau n à ∞ par un seul interprète qui les simule tous.

Un empilement d'interprètes qui s'interprètent les uns les autres se traduit par une absence totale d'efficacité à l'exécution. Pour résoudre ce problème, la tour est souvent implémentée par une pile de continuations et un seul interprète méta-circulaire. L'interprète simule la tour en effectuant des changements de niveaux à chaque appel de lambdas réflexives. J. des Rivières [desRivières 84] montre l'équivalence de ce modèle muni d'un seul flot de calcul au modèle muni de multiples flots de calculs que suppose la description philosophique d'une tour virtuellement infinie d'interprètes. Cette équivalence ne tient que dans la mesure où la définition des interprètes est identique à chaque niveau.

3.4 La modélisation de l'interprète

Le modèle choisi pour un interprète réflexif conditionne les possibilités de programmation réflexive offertes à l'utilisateur: non seulement il spécifie ce qui est rendu explicite de l'interprète mais aussi comment cela est rendu explicite. Ce faisant, il influe sur les modifications de l'interprète que l'utilisateur pourra opérer ainsi que sur la façon dont il pourra les effectuer. La modélisation des différents aspects du langage constitue donc un des problèmes les plus sensible lors de l'introduction de la réflexivité.

Nous examinons ici les modèles d'interprètes adoptés par les différents interprètes réflexifs pour les langages fonctionnels ainsi que la modélisation de l'un des objets les plus important de ces langages, les lambdas.

3.4.1 Les interprètes basés sur l'évaluation des expressions

Dans la plupart des interprètes réflexifs, l'interprétation fonctionne selon un modèle basé sur l'évaluation des expressions du langage. L'interprète 3-Lisp a adopté lui aussi ce modèle.

Un tel interprète s'organise autour de l'évaluation des expressions du langage. La fonction d'évaluation des expressions de l'interprète est décomposée en fonctions plus élémentaires suivant un découpage qui correspond à celui des expressions du langage : évaluation des symboles, des constantes, des conditionnelles, des abstractions etc. Ce découpage de l'interprète conditionne les modifications que l'utilisateur pourra effectuer en le contraignant à les exprimer sous la forme d'une évaluation des expressions du langage.

Les registres d'un tel interprète sont au nombre de trois: le premier contient naturellement l'expression en cours d'évaluation, le second l'environnement lexical qui spécifie le contexte dans lequel l'évaluation doit avoir lieu et le troisième capture le flot de calcul sous la forme d'une continuation. Chacune des valeurs contenues dans ces registres adopte une représentation qui varie d'un interprète à l'autre.

- Les *expressions* sont toujours représentées sous forme de listes. La liste constitue en effet l'unique structure de donnés adoptée par ces interprètes. Et l'utilisation de langages basés sur cette même structure pour les implémenter facilite ce choix: toutes les primitives nécessaires à leurs manipulations y sont disponibles.

- Deux modèles d'*environnements* ont été utilisés par les différents interprètes réflexifs: les A-listes et les fermetures lexicales. L'utilisation des A-listes permet de limiter le nombre de primitives à introduire dans le langage pour les rendre manipulables par l'utilisateur, les primitives de manipulation des listes y étant déjà présentes. Cette représentation classique de l'environnement est souvent choisie parce qu'elle le laisse ouvert à l'utilisateur, et donc manipulable et modifiable. L'utilisation des fermetures lexicales pour représenter l'environnement, modèle choisi dans Brown ([Friedman 84], [Wand 88]), a aussi ses avantages. En particulier elle facilite l'attachement de procédures spécifiques à la lecture et l'écriture des variables. L'écriture d'une fonction de trace des accès aux variables est ainsi facilitée. Cependant, elle ne permet pas de distinguer la lecture de l'écriture et elle rend difficiles certaines manipulations de la structure de l'environnement. En particulier, l'utilisation de fermetures lexicales lie le degré d'ouverture de ce modèle à celui des lambdas. Si ces dernières utilisent les lambdas du langage servant à l'implémentation, comme c'est le cas dans Brown, la structure globale de l'environnement reste alors cachée à l'utilisateur.

- La *continuation du calcul* est représentée par une lambda à un seul paramètre, le résultat de l'évaluation de l'expression. Dès lors, les manipulations structurelles que l'utilisateur peut opérer sur les continuations sont identiques à celle qu'il peut opérer sur les lambdas. Lorsque les lambdas sont implémentées par des objets du langage qui sert à l'implémentation ([Friedman 84], [Wand 88], [Bawden 88], ...), les manipulations possibles se limitent aux aspects opératoires, c'est à dire, à spécifier une autre continuation ou à enrichir la continuation courante d'un calcul intermédiaire. Bien que ce type de manipulation soit le plus fréquent, l'adoption d'une représentation trop fermée limite les possibilités de la programmation réflexive. Par exemple, l'extraction de la continuation de la continuation courante depuis son environnement lexical est alors impossible.

3.4.2 L'interprétation selon Stepper

L'interprétation dans Stepper [Bawden 88] n'est pas basée sur l'évaluation des expressions du langage. Plutôt que de prendre l'évaluation comme moteur du couple (eval, apply), Stepper s'organise autour de l'application. L'interprétation ne correspond alors pas à l'évaluation d'une expression mais à une transition d'un état à un autre. Chaque état est représenté par un t-uple contenant les valeurs des registres de l'interprète. Interpréter consiste alors à calculer le prochain état de l'interprète i.e. le prochain t-uple.

Pour exemple, prenons l'évaluation d'une constante. Stepper, lorsqu'il doit évaluer une constante, passe directement sa valeur à la continuation du calcul. En cela il reproduit un comportement similaire aux interprètes plus courants. Cette opération ne se traduit pourtant pas par l'appel de la continuation avec comme paramètre la valeur de la constante, mais par la construction d'un t-uple qui représente la prochaine étape de l'interprétation.

```
(define (evaluate-constant k e r)
        (list k e)) ; Retourne le t-uple
```

La fonction de Stepper qui réalise l'évaluation des constantes, *evaluate-constant*, construit le t-uple qui définit l'état suivant de l'interprète. L'activité de Stepper n'est pas le fruit de l'évaluation. Les fonctions de l'interprète ne font que calculer sous forme de t-uple la prochaine étape de l'interprétation. Elles fonctionnent sous la forme de transitions d'états. C'est une fonction externe au processus d'évaluation qui se charge d'activer les différents états de l'interprète, le premier élément du t-uple étant toujours une fonction dont les éléments qui suivent constituent les paramètres.

L'interprétation telle qu'elle est modélisée dans Stepper n'est pas aussi naturelle et courante que l'évaluation d'expressions. L'un des objectifs de Stepper est de démontrer l'indépendance du concept de réflexivité du modèle d'interprétation adopté. Cependant, si cette indépendance est démontrée par Stepper, le modèle plus courant d'évaluation des expressions du langage nous semble plus naturel et mieux répondre à la question du passage d'une représentation sous forme de données à son interprétation.

3.4.3 La représentation des lambdas

La représentation des lambdas adopte deux modèles: une structure contenant sous forme de listes les paramètres de la lambda, son corps ainsi que son environnement lexical, ou une fermeture lexicale du langage qui sert à l'implémentation, la fermeture lexicale décrivant alors les opérations qui correspondent à l'évaluation de la lambda de l'interprète réflexif. Cette seconde représentation qu'utilisent Brown dans ses deux versions ([Friedman 84] et [Wand 88]) ainsi que Stepper [Bawden 88] rompt l'uniformité du langage en faisant appel à des objets appartenant à un autre langage. Ces objets étrangers au langage implémenté sortent du champ des entités manipulables par l'utilisateur. Alors que dans la première des représentations, l'environnement lexical de la lambda est accessible, que la liste des paramètres et le corps de la lambda sont manipulables à souhait, les lambdas de Brown perdent en souplesse et en flexibilité. La réflexivité structurale n'est alors pas pleinement réalisée.

3.5 Outils linguistiques pour la programmation réflexive

Tout l'intérêt des interprètes réflexifs réside dans la possibilité qu'ils offrent d'être modifiés. Cette capacité est supportée dans le langage interprété par des constructions linguistiques spécifiques. Grâce à elles, l'utilisateur a la possibilité de modifier localement le processus d'interprétation en cours.

3.5.1 Les lambdas réflexives

Les lambdas réflexives [Smith 83] constituent la construction linguistique la plus fréquente dans les interprètes réflexifs de langages fonctionnels. A quelques variations près sur la façon de les construire, elles reposent toutes sur l'idée que les lambdas sont un outil adéquat de programmation réflexive.

La distinction entre les programmes d'un même niveau et ceux du niveau supérieur se fait alors à l'aide d'un typage des lambdas. D'un point de vu linguistique, les lambdas réflexives sont obtenues soit par la spécification du type de la lambda lors de sa construction [Smith 83], soit par transformation d'une lambda classique en lambda réflexive à l'aide d'une instruction dédiée [Wand 88].

Le caractère réflexif des lambdas réflexives tient en deux aspects complémentaires.

Le premier de ces aspects concerne le niveau d'exécution de ces lambdas qui, lors de leur application et après avoir été transformées en lambdas classiques, sont appelées comme des lambdas faisant partie intégrante de l'interprète du niveau courant. Leur interprétation n'est alors plus due à l'interprète du niveau courant, mais à l'interprète de l'interprète du niveau courant. Elles permettent ainsi la modification dynamique du flot de calcul d'un interprète en substituant leur exécution à celle des fonctions de l'interprète.

Le deuxième aspect est celui des éléments de l'interprète qu'elles permettent de réifier. À travers leur vecteur de paramètres, l'interprète met à la disposition de ces lambdas l'expression courante (leurs paramètres d'appels non évalués), l'environnement lexical dans lequel la lambda réflexive est appelée, ainsi que la continuation du calcul. Ces éléments correspondent aux registres de l'interprète. Si la continuation et l'environnement sont toujours liés à leurs paramètres respectifs, 3-Lisp offre au niveau de l'expression un mécanisme de liaison d'arbre qui permet de la décomposer lors de son passage en paramètre. Ce mécanisme n'a pas été repris par les autres interprètes dans lesquels le programmeur est astreint à réaliser cette décomposition à la main.

Les valeurs des paramètres qui lui sont passées, combinées avec leur niveau d'exécution, permettent aux lambdas réflexives d'opérer sur l'interprétation en utilisant et, ou modifiant une partie, ou l'ensemble des registres de l'interprète. Elles opèrent comme un interprète local des expressions du langage.

La réflexivité construite sur des lambdas réflexives s'organise autour de l'application. Lorsque l'interprète traite une application, il en examine le premier élément pour en déterminer la nature. L'appel des lambdas réflexives peut ainsi avoir lieu de manière distincte de celles des lambdas classiques.

Les éléments réifiés de l'interprète supposent que leurs valeurs puissent être extraites. Cette extraction implique un choix sur le modèle de l'interprétation. La plupart des implémentations optant pour les lambdas réflexives utilisent un interprète écrit en continuation explicite avec un passage explicite des trois composantes réifiées, l'expression, l'environnement lexical et la continuation aux fonctions principales de l'interprète.

La modification des aspects globaux de l'interprète (réflexivité globale) dans le modèle des lambdas réflexives est possible en utilisant des effets de bords. Cependant, comme nous le montrerons dans le chapitre suivant, la réalisation du lien causal entre les différents niveaux d'interprétation étant souvent partielle, la réflexivité globale n'est pas possible dans la plupart des implémentations.

Les lambdas réflexives ne sont pas des méta-lambdas: elles n'expriment pas une lambda particulière mais permettent d'intervenir localement dans le processus d'interprétation en cours.

Problèmes liés aux lambdas réflexives

Les lambdas réflexives permettent l'introduction de nouvelles formes spéciales. Le traitement des lambdas réflexives en objet de première classe soulève cependant une question: les formes spéciales introduites à l'aide des fonctions réflexives prennent le caractère d'objet de première classe que leur confèrent les fonctions réflexives. L'exemple de [Bawden 88] qui implémente une fonction **quote** est très éloquent: lorsque cette fonction passée en paramètre est appliquée dans l'expression $(f\ x)$ où f est alors liée à **quote** et x à une valeur, c'est en fait x qui est quoté au lieu de sa valeur.

Mais ce caractère d'objet de première classe donné à une forme spéciale n'est pas à notre sens un problème dû aux lambdas réflexives. L'utilisation qui est faite de ces lambdas pour implémenter des formes spéciales nous induit en erreur sur la nature des lambdas réflexives. Cette vision des lambdas réflexives [Friedman 84] est restrictive quant à leurs possibilités. Si elles permettent d'étendre le langage par adjonction de formes spéciales, elles ne sont pas parfaitement adéquates pour cette utilisation. Les formes spéciales sont traitées dans les interprètes comme des branches des différents cas traités par la fonction d'évaluation de l'interprète. Pour éviter de conférer à une forme spéciale le caractère d'objet de première classe, la modification de la fonction d'évaluation est plus appropriée que les lambdas réflexives.

Une autre problématique des lambdas réflexives notée par [Danvy 88] est plus conséquente: la capture de l'environnement lexical de sa définition. Cette capture mime la création des lambdas classiques. Lors de l'exécution d'une lambda réflexive, l'environnement capturé, qui contient les variables lexicales visibles depuis l'endroit où la lambda a été créée au niveau n, et qui est un objet de niveau $(n+1)$, va servir au niveau $(n+2)$ d'environnement de base à l'évaluation du corps de la lambda. Cet objet subit donc un transfert d'un niveau à un autre. Les variables lexicales entourant la création de la lambda réflexive deviennent ainsi directement accessibles dans son corps:

```
  ; Exemple écrit en I_r
  ; (I_r est l'interprète réflexif développé par S. Jefferson dans [Jefferson 92])
(let ((x 5))
     (compound-to-reifier
          (lambda (e r k)
             (k x))))   ; Accès direct à x au niveau (n+1)
```

La capture de l'environnement lexical de la définition d'une lambda réflexive produit un mélange entre des objets appartenant à des niveaux d'interprétation différents. Ce phénomène peut conduire à certaines confusions de la part du programmeur. Pour l'éviter et obtenir une sémantique plus propre, Danvy [Danvy 88] propose deux types d'abstractions jouant un rôle similaire aux lambdas réflexives: les γ-abstractions et les δ-abstractions. Les premières évaluent leur corps dans l'environnement correspondant à l'environnement se trouvant au niveau d'interprétation au dessus de celui habituellement capturé lors de leur définition, tandis que les secondes utilisent l'environnement se trouvant au dessus de celui de leur application. L'accès direct aux variables lexicales de l'environnement de définition est ainsi rendu impossible ainsi que la récupération de ces informations. Pour qu'elles restent accessibles depuis le corps de la lambda réflexive, un lien vers l'environnement de définition doit être explicitement maintenu. Les valeurs des variables lexicales sont alors récupérables en évaluant dans cet environnement les symboles auxquels elles sont liées.

Malgré l'ambiguïté que peut faire naître la capture de l'environnement lexical courant lors de la définition des lambdas réflexives, ce phénomène se révèle être un outil pratique pour communiquer entre les différents niveaux d'interprétation. L'entorse qui est ainsi faite à la propreté de la sémantique devient un allié très utile dès lors que les aspects pratiques sont pris en considération.

3.5.2 Le *exec-at-metalevel*

Black [Asai 93] et RScheme [Ichisugi 92] utilisent une autre construction linguistique que les lambdas réflexives pour la programmation réflexive: le *exec-at-metalevel*. Elle permet de modifier le niveau d'interprétation de l'expression qui lui est passée en paramètre. L'évaluation de cette expression n'est alors plus le fait de l'interprète du niveau courant mais de l'interprète de cet interprète.

Contrairement aux lambdas réflexives qui organisent la réflexivité autour de la notion d'application, la construction *exec-at-metalevel* utilise l'évaluation comme point central de la notion de réflexivité. Il ne s'agit pas de modifier le niveau d'interprétation dans lequel la lambda réflexive sera appliquée, mais de modifier le niveau d'évaluation de l'expression passée en argument.

La restriction des paramètres disponibles lors de l'évaluation de l'expression à cette dernière impose de sévères restrictions sur les actions qui peuvent être effectuées sur le processus d'interprétation en cours. Alors que les lambdas réflexives permettent d'influer sur ce cours en agissant sur les éléments du triplet (expression, environnement, continuation), la construction *exec-at-metalevel* ne permet pas d'agir directement sur l'interprétation en cours.

Ce type de construction pousse à agir de manière globale sur les définitions de l'interprète pour le modifier. En ne mettant à travers *exec-at-metalevel* à la disposition de l'utilisateur aucun des registres de l'interprète, ce modèle contraint à l'utilisation d'effets de bords. Les modifications purement locales du processus d'interprétation ne sont pas possibles.

La connexion causale d'un interprète réflexif basé sur cette construction doit nécessairement être plus forte que celle d'un modèle basé sur les lambdas réflexives. L'utilisation d'effets de bords sur les définitions de l'interprète rend indispensable l'existence d'une connexion causale entre les définitions de l'interprète telles qu'elles sont accessibles à l'utilisateur dans l'environnement global et celles que l'interprète utilise. Les interprètes basés sur les lambdas réflexives ne nécessitent pas un tel lien. Ils se basent uniquement sur la présence d'une réflexivité locale par opposition à la réflexivité supposée globale de Black et RScheme.

Black et RScheme ne réalisent pas complètement ce lien causal. Comme nous le verrons au chapitre suivant, ce lien y est réalisé sur un seul niveau d'interprétation. La réflexivité globale, si elle est très intéressante au niveau des possibilités de programmation réflexive qu'elle procure, est plus lourde et complexe à réaliser que la réflexivité locale.

La construction linguistique *exec-at-metalevel* est un outil moins performant que les lambdas réflexives et plus lourd à mettre en œuvre. Ceci explique sans doute que les lambdas réflexives prévalent sur cette construction.

3.5.3 Le cas de Stepper

Stepper [Bawden 88] présente une construction linguistique pour la programmation réflexive qui bien que s'apparentant aux lambdas réflexives, n'en est pas tout à fait une. Les différences résident dans la nature des lambdas et dans les valeurs des paramètres passés aux lambdas réflexives.

Les raisons de telles différences proviennent du modèle de calcul adopté par Stepper. L'évaluateur de Stepper n'est pas un interprète des expressions du langage. Au lieu d'interpréter les expressions, il construit une fermeture traduisant l'interprétation en transitions d'états exprimées en continuations explicites. Chaque étape de l'interprétation correspond à une opération sur un t-uple dont chaque élément est un registre de l'interprète. Les fonctions de l'interprète opèrent en agissant sur le contenu du t-uple qu'elles retournent comme résultat. L'activité des interprètes au sein de la tour est ensuite traduite par une fonction, externe à l'évaluateur, qui assure l'exécution des éléments du t-uple au niveau d'interprétation souhaité.

La distinction entre fonctions classiques et fonctions réflexives est obtenue en appliquant à une fonction classique une primitive permettant de la transformer en fonction réflexive: *procedure->implementation*. Ce procédé est à rapprocher de celui de Brown-88 [Wand 88] qu'il reproduit à une nuance près: si Brown-88 modifie le «tag» de la structure décrivant une fermeture, Stepper l'encapsule dans une liste «taguée». Une fonction de Stepper peut ainsi être utilisée en même temps comme fonction simple et comme fonction réflexive, la construction de la seconde n'affectant pas la première.

L'adjonction de ces constructions dans Stepper résulte d'une modification non de la fonction d'évaluation ainsi qu'elle est habituellement pratiquée dans les autres interprètes ([desRivières 84], [Friedman 84], [Jefferson 92], ...), mais de celle assurant la gestion de la tour indépendamment du processus d'évaluation. Les paramètres passés à une fonction réflexive de Stepper sont donc évalués, aucune branche de l'évaluateur ne traitant ce cas particulier.

Pour finir, si la continuation du calcul reste accessible au sein des fonctions réflexives, l'environnement ne l'est plus. C'est le résultat d'un choix délibéré de Bawden qui veut nous montrer par là que la réflexivité n'est pas nécessairement liée au modèle d'interprétation basé sur l'évaluation d'expressions — elles ne sont d'ailleurs plus accessibles — ni à l'accès au triplet (expression, environnement, continuation). Si l'interprète qu'il construit démontre bien sa thèse, le type de programmation réflexive qu'il offre reste plus complexe à manipuler par l'utilisateur que le modèle courant.

En montrant que le caractère fondamental des interprètes réflexifs est de permettre à l'utilisateur de modifier le niveau d'exécution de ses fonctions, Bawden dote Stepper d'un modèle de programmation réflexive plus proche de l'*exec-at-metalevel* de Black [Asai 93] que de celui des lambdas réflexives. Mais, comme le note l'auteur de Stepper, l'objectif de cet interprète n'est pas d'être utile mais d'étudier ce qui rend un interprète réflexif.

3.5.4 Le *make-prelim* de Refci

Refci ([Simmons II 92a], [Simmons II 92b], [Simmons II 93]) adresse avec son *make-prelim* une autre dimension de la réflexivité: la réflexivité globale. Alors que les constructions précédentes modifient localement le cours de l'interprétation, cette dernière insère effectivement de nouvelles lignes de code dans l'interprète ce qui provoque une modification de l'ensemble du processus d'interprétation. L'esprit dans lequel travaille le programmeur est fortement modifié par cette approche.

La mise en place de cette construction passe par une révision du statut même de l'interprète: il est traité par Refci comme objet de première classe. De plus, et afin de permettre deux types d'extensions, des actions préliminaires à l'évaluation ainsi que les formes spéciales, l'interprète est séparé en deux fonctions, un «prelim» et un «dispatch». Cette dernière est constituée de l'essentiel de l'évaluation, l'analyse des expressions suivit du traitement de chacun des cas. Le «prelim» ne participe pas à proprement parler à l'évaluation. Il n'est là que pour permettre l'introduction d'actions préalables à l'évaluation. Ces deux fonctions s'ajoutent aux paramètres de l'interprète, leur conférant ainsi le statut d'objet de première classe.

La programmation réflexive dans Refci consiste en l'écriture de nouveaux «prelims». Grâce à la construction *make-prelim*, les modifications introduites affectent l'interprète dans son ensemble. Il ne s'agit plus de se substituer localement à l'interprète le temps de l'appel d'une lambda réflexive, mais d'opérer une modification globale de ce même interprète.

Le traitement de la réflexivité globale par Refci ne nous semble pas approprié. Dans le chapitre suivant, nous montrerons que la présence de la réflexivité globale relève non pas du modèle, comme semble le suggérer Refci, mais de l'implémentation. La clef de la réflexivité globale réside, selon nous, dans l'implémentation de la connexion causale.

En insistant sur la réflexivité globale, Refci n'oublie pourtant pas la réflexivité locale. Les lambdas réflexives cohabitent avec le *make-prelim*. Ce faisant, Refci souligne le rôle complémentaire de chacune de ces constructions.

3.5.5 Des lambdas réflexives à l'*exec-at-metalevel* et vice versa

Les constructions linguistiques précédentes sont dans une certaine mesure équivalentes: elles peuvent être utilisées pour introduire au sein de leur interprète les autres constructions. A priori, il semblerait donc que le choix de l'une ou l'autre d'entre elles soit sans importance. Cependant, outre les aspects pratiques liés à leurs utilisations, elles ne sont pas toutes égales quant aux contraintes qu'elles imposent sur la connexion causale. L'introduction de ces constructions dans des interprètes adoptant une autre d'entre elles révèle mieux ce que sont ces contraintes et quels sont les avantages d'une construction sur l'autre.

L'extension d'un interprète réflexif muni des lambdas réflexives vers l'*exec-at-metalevel* est facilement réalisable. Cette dernière construction se contente d'évaluer au niveau méta l'expression qui lui est passée en paramètre. Un tel mécanisme peut être vu comme un sous ensemble de celui des lambdas réflexives. L'expression passée en paramètre de l'*exec-at-metalevel* correspond au premier paramètre d'une lambda réflexive. L'expression devant être exécutée comme du code appartenant au niveau de l'interprète, son évaluation est le fait de l'interprète de niveau (*n+2*). Il est donc nécessaire d'appeler une seconde lambda réflexive au sein de la première afin de monter de deux niveaux d'interprétation. Le corps de cette deuxième lambda réflexive est alors constitué d'un appel à l'évaluateur du niveau (*n+2*) qui traduit l'exécution du code représenté par l'expression passée à l'*exec-at-metalevel* au sein du niveau (*n+1*). Le code qui suit traduit ce mécanisme:

```
; exec-at-metalevel en lambda réflexive en I_r
(set! exec-at-metalevel
         ; On monte d'un premier niveau
         (compound-to-reifier
           (lambda (e0 r0 k0)
                        ; Puis d'un second niveau pour évaluer l'expression
             (k0 ((compound-to-reifier
                    (lambda (e1 r1 k1) (evaluate e0 r1 k1))))))))
```

Les appels des continuations respectives des deux lambdas réflexives permettent le retour de la valeur qui résulte de l'évaluation de *e0* au niveau (*n*).

La traduction de l'*exec-at-metalevel* en terme de lambdas réflexives est assez directe. Il ne reflète cependant pas une contrainte sur le lien causal: la prise en compte des définitions contenues dans l'environnement global de l'interprète par ce même interprète. Si la connexion causale prend en compte un tel lien nécessaire à l'utilisation de l'*exec-at-metalevel* alors la traduction de cette construction linguistique en terme de lambdas réflexives a un sens. Dans le cas contraire, cette traduction se révèle être un exercice sans véritable fondement.

L'exercice symétrique, celui qui consiste à introduire les lambdas réflexives dans un interprète réflexif basé sur l'*exec-at-metalevel*, est plus complexe. Il nécessite la modification de l'interprète. La possibilité d'une telle modification réside dans la nature de la connexion causale: elle doit permettre de modifier les définitions de l'interprète et prendre en compte ces modifications. La construction linguistique *exec-at-metalevel* nécessitant un tel lien pour pouvoir être utile, Black [Asai 93] en est pourvu.

Les lambdas réflexives nécessitent un constructeur et le traitement de leur cas spécifique lors de l'application. La construction des lambdas réflexives est décomposable en deux étapes: la construction d'une fermeture lexicale classique suivie de sa transformation en lambda réflexive. C'est cette deuxième étape qu'il nous faut ajouter. Une lambda classique est chargée de cette tâche: elle construit une liste composée du symbole *reifier* suivi du *cdr* de la liste qui décrit la lambda à transformer en lambda réflexive. L'objet retourné n'est pas reconnu comme une lambda réflexive par l'interprète. L'application des lambdas réflexives est introduite en modifiant le traitement des applications dans la fonction d'évaluation de l'interprète Black. Pour cela, le premier élément d'une application est évalué avant les autres afin d'en déterminer le type pour adopter le comportement adéquat. Lorsque cet élément est une lambda réflexive, l'application est traitée comme un appel d'une fonction de l'interprète après transformation de la lambda réflexive en lambda classique. Dans Black, ces mécanismes s'écrivent:

```
  ; lambdas réflexives en Black
  ; Le constructeur de lambdas réflexives
(define make-reifier
        (lambda (l) (cons 'reifier (cdr l))))
```

```
; L'application des lambdas réflexives
(exec-at-metalevel
   (set! eval-application
         (lambda (exp env cont)
              ; L'application avec traitement des lambdas réflexives
             (base-eval (car exp) env
                        (lambda (l)
                             (if (and (pair? l)
                                        (eq? (car l) 'reifier))
                                      ; Après transformation en lambda classique,
                                      ; l'appel d'une lambda réflexive se fait
                                      ; directement comme celui d'une fonction de l'interprète
                                    ((cons lambda-tag (cdr l)) (cdr exp) env cont)
                                     ; Application des lambdas classiques
                                    (eval-list (cdr exp) env
                                       (lambda (vals)
                                            (base-apply l vals env cont)))))))))
```

Les lambdas réflexives ainsi introduites dans Black ne sont disponibles qu'au niveau utilisateur. Black adopte pour chaque niveau d'interprétation un environnement global différent. La modification n'affecte donc que l'interprète du niveau auquel elle est effectuée.

La réflexivité rend possible l'introduction de l'une des constructions dans un interprète qui en adopte une autre. Cependant, dans le premier cas, l'utilisation d'une telle modification nécessite la présence d'une connexion causale de l'interprète et de ses définitions présentes dans l'environnement global. Cette connexion ne peut être introduite par l'utilisateur et doit être préalablement présente dans l'interprète. Dans le deuxième cas, une telle connexion est toujours présente parce que rendue nécessaire par le type de programmation réflexive induite par la construction linguistique adoptée. De plus, le non partage de l'environnement global entre les différents niveaux d'interprétation en limite la portée.

Les différentes constructions linguistiques adoptées par les interprètes réflexifs voient les possibilités de programmations réflexives qu'elles offrent limitées par les choix d'implémentation tant au niveau de la connexion causale que du partage ou non de l'environnement global. Nous voyons ici que les capacités offertes par la réflexivité dépendent tout autant du modèle que de l'implémentation.

3.6 Réflexivité globale dans les langages fonctionnels

Les constructions linguistiques qui viennent d'être étudiées sont orientées vers la réflexivité locale. Elles permettent d'exécuter du code utilisateur au niveau de l'interprète. Cette tâche achevée, le code utilisateur est à nouveau exécuté au niveau 0. L'interprétation n'est modifiée que pour la courte durée pendant laquelle le code utilisateur est exécuté au niveau de l'interprète. Le reste du temps l'interprétation suit un cours normal.

Certains problèmes nécessitent parfois de modifier durablement l'interprète. La trace des accès à une variable donnée dans un interprète qui représente l'environnement global par une A-liste est l'un de ces problèmes. Pour implémenter cette trace, les fonctions qui gèrent l'environnement global doivent être modifiées durablement. Une lambda réflexive ne permet d'observer ces accès que durant son exécution. Une fois cette exécution achevée, elle ne permet plus de savoir si une variable est lue ou écrite. Pour obtenir cette information, il faut donc modifier l'interprète en faisant appel à la réflexivité globale.

Les interprètes que nous avons étudiés, s'ils disposent des outils linguistiques pour supporter une telle réflexivité, ne semblent pourtant pas en disposer. Il se pose, dans ces interprètes, un problème de connexion causale.

Pour illustrer ce problème, nous utilisons une version simplifiée de 3-Lisp [Smith 83]: 3-Scheme. Basée sur les techniques d'implémentation présentées par J. des Rivières dans [desRivières 84], 3-Scheme est une version à la sémantique simplifiée dans laquelle abstraction est faite des opérateurs ↑ et ↓ de 3-Lisp. La tour d'interprètes suit strictement l'implémentation présentée pour 3-Lisp dans [desRivières 84]. Les conclusions qui peuvent être tirées de l'étude de la tour sont donc les mêmes que celles que l'on aurait obtenues avec 3-Lisp.

Les lambda réflexives réifient les trois registres principaux de l'interprète du niveau supérieur. L'état de l'interprétation y est traduit à travers les trois composantes que sont l'expression à évaluer, l'environnement d'évaluation et la continuation. L'ensemble des interprètes réflexifs gèrent correctement la connexion causale relative à ces registres. L'utilisation des lambdas réflexives pour ajouter de nouvelles formes spéciales est ainsi rendue possible. Ce type programmation se traduit par une modification locale du comportement de l'interprète. Lors de l'appel d'une lambda réflexive, le code de la lambda est dynamiquement substitué à celui de l'interprète. Le cours normal de l'interprétation reprendra lors de l'appel de la continuation passée en paramètre de la lambda.

La sémantique des langages implémentés par les interprètes réflexifs autorise l'utilisation d'effets de bord à l'intérieur du corps de ces lambdas. L'environnement global de l'interprète est donc modifiable. L'utilisation des effets de bords pour introduire de nouvelles fonctions ou pour définir ou modifier des variables globales est d'ailleurs une pratique courante.

Dans une tour d'interprètes, l'environnement global de l'interprète de niveau n contient les définitions des fonctions de l'interprète de niveau $n-1$. Ces définitions sont modifiables par effet de bord sur cet environnement. Lorsque que l'on opère une modification de l'une des définitions de l'interprète, nous nous attendons à en observer les effets.

Un exemple qui nécessite la modification des définitions de l'interprète est la mise en place d'un système de trace. Avec les lambdas réflexives la trace n'a lieu que sur une portion des activités de l'interprète. Pour tracer l'ensemble des activités de l'interprète, une modification de la fonction d'évaluation est nécessaire.

```
; Tentative de trace des activités de l'interprète (3-Scheme)
(1)> ((lambda reflect (() env cont)                                  (i)
        ((lambda simple (old-interp)
                 ; Modification par effet de bord de l'évaluateur
           (set! interp (lambda simple (exp env cont)
                           (print exp)
                           (old-interp exp env cont))))
         interp)
        (cont #t)))
(1): #T
(1)> (+ 1 2)                                                          (ii)
(1): 3
(1)> interp                                                           (iii)
(1): #<CLOSURE (lambda SIMPLE (EXP ENV CONT)
                       (PRINT EXP)
                       (OLD-INTERP EXP ENV CONT))>
(1)> _
```

En effectuant un effet de bord sur la définition de *interp* au niveau de l'interprète, nous tentons avec (i) d'obtenir une modification du comportement de l'interprète. Théoriquement, chaque expression évaluée par l'interprète devrait être affichée à l'écran. Le calcul d'une expression simple (ii) ne produit aucun affichage supplémentaire. La définition de la fonction *interp* est pourtant modifiée[2] (iii).

Ce comportement de l'interprète est déroutant. L'affichage des expressions évaluées ne s'est pas produit contrairement à ce que nous attendions. La définition de *interp* contenue dans l'environnement global n'est donc pas celle utilisée par l'interprète. Si l'interprète va chercher les définitions des fonctions de l'utilisateur dans l'environnement global, il ne va pas y chercher les définitions de ses propres fonctions qui y sont pourtant présentes. La connexion causale entre l'environnement global et l'interprète n'est pas complète.

2. L'environnement global est en fait partagé entre les interprètes. La définition de *interp* est donc la même à tous les niveaux d'interprétation.

Le même comportement est observable dans la plupart des interprètes réflexifs. Ils revendiquent, le plus souvent explicitement, l'impossibilité d'effectuer des modifications globales de l'interprète. La connexion causale présente dans ces interprètes se limite aux lambda réflexives.

Black [Asai 93] accepte en partie la modification des définitions de l'interprète. La mise en place d'un système de trace en Black donne des résultats probants.

```
; Trace en Black
> (exec-at-metalevel
     (let ((old-base-eval base-eval))
             ; Modification de l'évaluateur par effet de bord
        (set! base-eval
               (lambda (exp env cont)
                  (display exp) (newline)
                  (old-base-eval exp env cont)))))
New level loaded
: base-eval
> (+ 1 2)
(+ 1 2)        ; Trace de l'évaluation de (+ 1 2)
+
1
2
: 3
> _
```

La prise en compte par l'interprète Black de la modification de la définition de *base-eval* démontre bien que la définition de l'interprète contenue dans l'environnement global et l'interprète sont causalement connectés. En arrêtant ici nos tests, nous pourrions conclure que Black réalise parfaitement la connexion causale. Cependant, en poussant notre démarche un peu plus loin, nous allons observer que cette connexion causale n'est pas complète.

La connexion causale de l'interprète peut être mise à mal si l'on s'intéresse non plus à la trace du niveau utilisateur mais à la trace des activités de l'interprète du niveau supérieur. Cette trace se réalise en montant de deux niveaux d'interprétation afin de modifier la fonction d'évaluation du niveau 2.

```
; Trace de l'interprète
> (exec-at-metalevel
     (exec-at-metalevel
        (let ((old-base-eval base-eval))
                ; Modification de l'évaluateur de niveau 2
           (set! base-eval
                  (lambda (exp env cont)
                     (display exp) (newline)
                     (old-base-eval exp env cont))))))
New level loaded
New level loaded
: base-level
> (+ 1 2)
: 3
> _
```

Plus aucune trace n'apparaît ! La connexion causale ne fonctionne plus correctement dès que la modification a lieu deux niveaux au dessus du niveau courant. Les conséquences d'un lien causal incorrect sont aggravées dans Black par le modèle de programmation réflexive mis en avant: le *exec-at-metalevel* offre moins de possibilités que les lambdas réflexives et avantage la réflexivité globale sur la réflexivité locale.

Les exemples de trace précédents démontrent l'absence d'un lien causal complet. Cette absence fait obstacle à la réflexivité globale. Le modèle de tour d'interprètes implémenté ne traduit pas fidèlement son esprit: une tour d'interprètes étant constituée d'interprètes chargés d'interpréter celui du niveau immédiatement en dessous du leur.

La première tentative de trace révèle la nécessité de connecter causalement la définition de l'interprète contenue dans l'environnement global avec l'interprète lui-même. Le deuxième exemple démontre le rôle de l'implémentation de la tour dans la réalisation du lien causal. Ces deux aspects doivent être pris en compte pour obtenir une connexion causale correctement réalisée support de la réflexivité globale.

Dans le chapitre suivant, nous verrons comment obtenir un interprète qui implémente complètement la connexion causale et quelle est l'utilité de la réflexivité globale.

3.7 Implémentation des interprètes réflexifs

L'implémentation de la tour d'interprètes est une des clés de la réalisation du lien causal. Elle se doit d'assurer la cohérence de la tour et de restituer le comportement attendu d'une tour d'interprètes dans laquelle chaque niveau est supposé interpréter le niveau immédiatement en dessous. Si la tour est vue sous l'angle philosophique comme un empilement d'interprètes actifs, les implémentations disponibles utilisent une tour statique. Chaque niveau de la tour est constitué non par un interprète mais par l'état de cet interprète. Un tel choix guidé par des considérations d'efficacité a des conséquences non négligeables sur la réalisation du lien causal.

Dans 3-Lisp [desRivières 84], l'objectif est d'obtenir une implémentation efficace de la tour d'interprètes afin d'en minimiser le coût à l'exécution. Pour atteindre ce résultat, l'interprète adopte une stratégie élaborée. Elle consiste à minimiser à tout instant le nombre de niveaux d'interprétation effectifs qui existent au sein de la tour d'interprètes. Pour ce faire, l'interprète ne monte d'un niveau que lorsqu'il doit exécuter du code provenant de l'utilisateur et redescend dès qu'il doit interpréter une fonction de l'interprète.

Comme le note J. des Rivières, cette stratégie se base sur le postulat que le code de l'interprète n'est pas modifié [desRivières 84]. En particulier l'interprète ne tient pas compte des modifications de ses définitions dans l'environnement global. La décision de monter ou descendre dans la tour se base seulement sur la nature des fonctions à appliquer. Et les actions consistant à monter et descendre, respectivement *shift-up* et *shift-down*, consistent à dépiler et empiler le troisième registre de l'interprète, la continuation, dans une pile d'états (Fig. 3.1

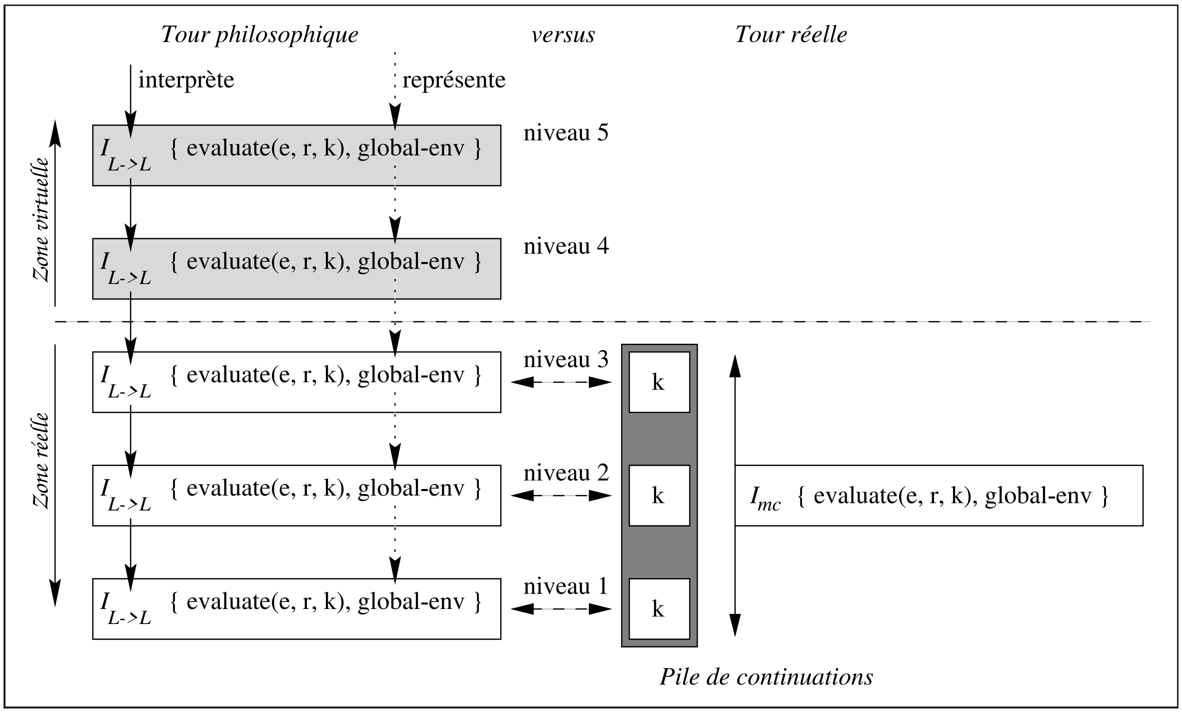

FIG. 3.1 – *Implémentation basée sur une pile de continuations*

page ci-contre).

L'interprète 3-Lisp réduit la tour à une pile d'états des interprètes de chaque niveau. La simulation de l'activité de la tour est obtenue par un unique flot de calcul dû à un interprète méta-circulaire. L'unicité du flot de calcul ne permet pas la prise en compte des éventuelles modifications des définitions des interprètes se trouvant au dessus du niveau courant d'exécution.

L'implémentation du lien causal dans 3-Lisp se limite à assurer la cohérence des registres utilisés par les lambdas réflexives. La programmation réflexive réduite à l'ajout de formes spéciales est ainsi assurée. Mais le lien causal ne tient pas compte de la modification globale des définitions des interprètes de la tour. La réflexivité dans 3-Lisp est uniquement locale.

Brown [Wand 88], dans sa version munie d'une tour d'interprètes, reproduit un schéma similaire à celui de 3-Lisp. La tour est modélisée à l'aide du concept de méta-continuation et est séparée du processus d'interprétation. Les méta-continuations regroupent au sein d'un seul et unique objet deux concepts: les continuations de calcul utilisées par l'interprète écrit en continuations explicites, et les changements de niveaux au sein de la tour. L'appel du paramètre k lié à la continuation au sein d'une lambda réflexive provoque la descente d'un niveau dans la tour d'interprètes. Les méta-continuations jouent donc ici le rôle de pile d'états propre à 3-Lisp. La tour d'interprètes est modélisée dans Brown par un unique flot de calcul qui offre une vision statique de l'empilement des interprètes au sein de la tour et qui ne permet pas la modification des définitions des interprètes.

L'implémentation de Blond [Danvy 88] laisse explicitement de côté la possibilité de modifier globalement l'interprète. L'implémentation de la tour suit un modèle très proche de Brown. Le modèle de tour est statique.

Black [Asai 93] utilise une modélisation spécifique de la tour qui combine duplication des interprètes à chaque niveau et évaluation partielle ([Consel 93], [Mogensen 96]). Les interprètes des différents niveaux d'interprétation y apparaissent explicitement. Mais l'utilisation de l'évaluation partielle pour spécialiser chaque interprète de niveau (n) en fonction de l'interprète de niveau $(n-1)$ dissocie ce dernier des interprètes de niveau $(n+1)$ et plus. Il en résulte le comportement surprenant que nous avons pu observer dans l'exemple de la section précédente. Et l'implémentation finit par réduire la tour à un seul interprète méta-circulaire qui monte et descend de niveau en niveau.

La réalisation du lien causal dans Black est partielle. Si c'est bien de réflexivité globale dont il s'agit là, la rupture du lien causal au delà d'un niveau d'interprétation est manifeste. Comme nous l'avons montré (cf. § 3.6 page 31), il n'est pas possible d'obtenir une trace de l'interprète de niveau (1) en modifiant la fonction d'évaluation de l'interprète de niveau (2). Or la réalisation d'un lien causal complet des différents interprètes de la tour devrait permettre d'influer sur le comportement de l'interprète de niveau (1) en modifiant celui de niveau (2).

Toutes ces implémentations modélisent la tour comme une pile contenant les états de l'interprète. L'activité de la tour y est le fruit de l'activité d'un unique interprète méta-circulaire qui monte et descend au sein de la tour au gré des besoins de l'utilisateur. Ce seul flot de calcul ne peut être pourtant comparé à une tour d'interprètes qui s'interprètent les uns les autres. Dans un tel modèle, il est sous entendu que chaque interprète reste actif. La modélisation par une tour statique n'est une représentation fidèle de la tour que dans la mesure où aucune modification globale de l'interprète n'est autorisée.

3.8 Efficacité des interprètes réflexifs

L'un des obstacles majeurs à l'utilisation des interprètes réflexifs est leur manque d'efficacité. L'exécution de programmes réflexifs se traduit souvent par des temps de réponse trop importants.

L'observation du fonctionnement des interprètes réflexifs permet d'identifier les deux causes principales de leur inefficacité. La première tient dans l'architecture des interprètes réflexifs et la seconde dans la manipulation d'une représentation de haut niveau de l'interprétation.

L'architecture des interprètes réflexifs se base sur le concept de tour virtuellement infinie. Cette dernière se compose d'un empilement d'interprètes dont chacun constitue un étage de la tour. Elle accumule plusieurs niveaux d'interprétation qui conduisent naturellement à rendre l'exécution des interprètes réflexifs inefficace. Le calcul d'une addition au niveau utilisateur se traduit par l'évaluation de l'expression correspondante au niveau méta. Cette évaluation se traduit au niveau méta-méta par une évaluation d'une expression qui représente l'évaluation de l'expression qui représente l'addition ... C'est cette accumulation de multiples niveaux d'interprétation qui rend les interprètes réflexifs inefficaces.

La manipulation d'une représentation de haut niveau de l'état de l'interprétation est dictée par la nécessité de donner accès à ces informations au programmeur. L'uniformité linguistique qui régit la plupart des interprètes réflexifs se traduit par une représentation de ces informations dans le langage du niveau utilisateur, ici Lisp ou Scheme. Ainsi, au moyen des registres de l'interprète, le programmeur peut manipuler l'expression à évaluer, l'environnement courant ou la continuation du calcul avec le même langage qu'il utilise pour manipuler les objets du domaine de calcul externe à l'interprète. L'utilisation d'une représentation de haut niveau de l'état de l'interprète, si elle facilite la programmation réflexive, contribue à rendre les systèmes réflexifs inefficaces.

Dans le cadre des systèmes réflexifs basés sur l'interprétation, peu de solutions au problème de l'efficacité ont été mises en œuvre.

L'implémentation usuelle des interprètes réflexifs basée sur l'utilisation d'une pile de continuations (cf. § 3.7 page 34) constitue la solution la plus courante au problème de l'efficacité. Elle agit en restreignant le champ de la réflexivité à des modifications locales du processus d'interprétation. La connexion causale est alors réduite au maintien des valeurs des registres de l'interprète afin que les lambdas réflexives puissent y accéder lors de leur exécution. Un seul flot de calcul est alors nécessaire à la simulation de la tour. L'interprète méta-circulaire qui implémente la tour monte et descend de niveau en niveau pour interpréter chaque application d'une lambda réflexive.

Cette solution n'est cependant que partielle d'une part, parce qu'elle me s'attaque pas au problème d'une représentation de haut niveau, et d'autre part, parce qu'elle ne résout que le problème de l'accumulation des niveaux d'interprétation due à l'architecture des interprètes réflexifs. Une autre source d'accumulation des niveaux d'interprétation est due à la programmation réflexive. Les lambdas réflexives font souvent appel à la fonction d'évaluation pour interpréter l'expression qui leur est passée en paramètre. Elles se comportent comme un interprète local de cette dernière. Le gain en performance est ainsi perdu lors de l'exécution d'une lambda réflexive.

Une autre solution proposée est l'évaluation partielle. Les spécialistes de cette technique de spécialisation de programmes la présentent souvent comme le candidat idéal au traitement de l'efficacité dans les systèmes réflexifs. Associée à un interprète, l'évaluation partielle agit comme un compilateur et supprime le niveau d'interprétation dû à I. Cependant, peu de travaux sur son utilisation dans le cadre de la réflexivité ont été effectués. Seuls Black et ABCL/R3, un interprète réflexif pour un langage à objets concurrents, y font appel.

Erik Ruf [Ruf 93a] se fait l'avocat de l'utilisation de l'évaluation partielle comme outil d'amélioration des performances des systèmes réflexifs basés sur les interprètes ou sur les compilateurs. Son discours essentiellement centré sur les langages orientés objets reste cependant assez généraliste. Il y montre, plus qu'il n'y démontre, les avantages potentiels de l'utilisation de l'évaluation partielle dans le but d'améliorer les performances de ces systèmes.

L'utilisation de l'évaluation partielle par Black [Asai 93] ne conduit pas à la suppression totale des niveaux d'interprétation contenus dans la tour. Son implémentation est basée sur la duplication et l'évaluation partielle de l'interprète. Chaque niveau de la tour est alors constitué d'un interprète I_1 qui résulte de l'évaluation partielle de l'interprète I_0 en fonction de ce même interprète I_0. Comme nous l'avons vu (cf. § 3.5.5 page 30), cette implémentation se traduit par une rupture du lien causal entre trois niveaux d'interprétation consécutifs. L'absence d'une connexion causale complète nuit à l'utilisation de la réflexivité globale pourtant essentielle à Black. L'évaluation partielle intervient ici comme outil d'amélioration de l'implémentation de chaque niveau de la tour et non comme moyen de supprimer les niveaux d'interprétation.

ABCL/R3 [Masuhara 95] est le premier interprète pour lequel la technique de l'évaluation partielle est utilisée pour supprimer le méta-interprète. La mise en œuvre de l'évaluation partielle y est plus délicate que dans Black, ABCL/R3 étant un langage à objets concurrents et les techniques développées jusqu'à maintenant ne prennent pas en compte la concurrence. La concurrence et les objets sont traités par une conversion du méta-interprète vers un interprète en continuation explicite écrit dans un langage fonctionnel. L'évaluation partielle est étendue par un traitement spécifique des effets de bord afin d'en garantir la conservation et leur position dans le code. Cependant seul le méta-interprète semble être l'objet d'une évaluation partielle, les autres méta-objets n'en bénéficiant pas. Les critères de l'évaluation partielle sont définis statiquement dès la création de l'objet qui fixe son méta-objet à cet instant. L'évaluation partielle n'est donc pas utilisée dans la réduction d'une tour d'interprètes et les problèmes que nous venons de soulever ne sont qu'en partie traités.

L'évaluation partielle semble pourtant une technique prometteuse pour traiter le problème de l'efficacité. Aussi dans un prochain chapitre, nous essaierons de l'utiliser pour supprimer les niveaux d'interprétation contenus dans la tour.

3.9 Conclusion

Nous avons vu dans ce chapitre que les interprètes réflexifs de langages fonctionnels sont constitués d'un empilement d'interprètes méta-circulaire qui simulent une tour virtuellement infinie d'interprètes. Des constructions spécifiques permettent à l'utilisateur de modifier localement l'interprétation en cours en effectuant un changement de niveau au sein de la tour.

Les interprètes réflexifs de langages fonctionnels exposent la plupart des problèmes liés à la notion de réflexivité, en particulier, le choix d'une représentation de l'interprète, l'implémentation de la réflexivité ainsi que le problème de l'efficacité. Ce sont ces problèmes que nous traitons dans la suite de cette thèse.

Au cours de cette étude, nous avons soulevé le problème de la réflexivité globale et celui de l'efficacité des interprètes réflexifs. Les deux chapitres qui suivent les reprennent plus en détail en proposant des implémentations d'interprètes réflexifs basées sur un interprète déjà existant, I_r[Jefferson 92]. Cette réutilisation va nous permettre de nous concentrer sur les problèmes soulignés.

Chapitre 4

Réflexivité globale dans les interprètes réflexifs

Les travaux sur la réflexivité dans les langages fonctionnels se sont axés sur la possibilité d'ajouter de nouvelles formes spéciales à l'interprète. En utilisant les lambdas réflexives l'utilisateur peut étendre à volonté le langage par adjonction de formes spéciales. Le type de programmation permis par les lambdas réflexives correspond à ce que nous nommons la réflexivité locale. De telles lambdas sont conçues pour modifier localement le processus d'interprétation. Elles interviennent dans ce processus en se substituant, lors de leur appel, à l'interprète. L'exécution de la lambda réflexive achevée, le cours normal du processus d'interprétation reprend en ignorant toute modification de la définition de l'interprète.

Le rôle premier des lambdas réflexives n'est pas de permettre la modification des définitions de l'interprète. La plupart des implémentations d'interprètes réflexifs excluant ce type de modifications se privent ainsi de toute une classe de programmes. L'introduction d'extensions qui nécessitent, par exemple, une gestion adaptée de l'environnement n'est pas envisageable dans ce cadre.

Dans ce chapitre, nous allons rendre possible la modification globale de l'interprète en dotant un interprète réflexif de capacités de réflexivité globale. Nous montrerons l'importance de l'architecture de la tour d'interprètes et son rôle dans la réalisation du lien causal. Enfin, nous illustrerons par des exemples les possibilités de programmation réflexive offertes par cet interprète.

4.1 Un interprète réflexif muni d'une tour active

Comme nous l'avons montré (cf. § 3.6 page 31), la plupart des interprètes réflexifs n'intègrent pas la notion de réflexivité globale. Les lambdas réflexives permettent uniquement une modification locale du flot de calcul. L'exécution de la lambda réflexive achevée, l'interprétation reprend son cours normal. Les possibilités de programmation réflexive qu'ils offrent sont limitées.

L'interprète décrit ici, I_{r^∞}, a été conçu pour, d'une part, prendre en compte la réflexivité globale et, d'autre part, offrir un modèle de tour d'interprètes plus proche de la philosophie de la réflexivité. Même si ces objectifs semblent distincts, ils se recoupent par les solutions à mettre en œuvre pour les atteindre.

La prise en compte de modifications des définitions de l'interprète nécessite une connexion causale correcte de l'interprète avec l'environnement global dans lequel il est décrit. Cette connexion ne suffit pourtant pas à permettre une réflexivité globale étendue à l'ensemble de la tour.

La tour est décrite dans [Smith 83] comme étant constituée par un ensemble d'interprètes dont chacun d'entre eux est en charge de l'interprétation de l'interprète qui se trouve au niveau immédiatement en dessous. La modification de l'interprète du niveau immédiatement supérieur doit donc influer directement sur le niveau courant. Mais la modification de l'interprète qui se trouve i niveaux au dessus du niveau courant doit influer indirectement sur l'interprétation du niveau courant. Un lien causal n'est complet que s'il permet l'ensemble de ces modifications.

La philosophie de la tour d'interprètes suggère que l'ensemble des niveaux de la tour sont actifs. Or les implémentations précédentes ([Danvy 88], [desRivières 84], [Wand 88]) se basent sur une vision statique de la tour dans laquelle un seul flot de calcul existe: celui de l'interprète méta-circulaire. Ce modèle, dans lequel la tour se rapproche d'une pile des états des interprètes, n'est pas assez puissant pour permettre une implémentation correcte du lien causal. S'il permet, dans une certaine mesure, de connecter un environnement global avec l'interprète qu'il décrit, il ne prend pas en compte les modifications des interprètes se trouvant plusieurs niveaux au dessus.

4.1.1 Architecture d'une tour active

Afin de nous rapprocher de la philosophie de la tour d'interprètes, notre interprète I_{r^∞} adopte une architecture plus simple directement issue de l'idée que chaque niveau de la tour a en charge l'interprétation du niveau immédiatement en dessous [Smith 83]. Cette idée suggère l'activité simultanée de l'ensemble des interprètes contenus dans la tour. Une telle activité ne peut être correctement modélisée par une pile d'états des interprètes. Par opposition à ce modèle statique, notre tour est composée par un ensemble d'interprètes qui s'interprétent effectivement les uns les autres.

Une tour active s'obtient en empilant les interprètes les uns sur les autres. Un tel empilement est à rapprocher des couches logicielles existant au sein d'un ordinateur. De la même manière, une tour d'interprètes peut être construite manuellement en chargeant dans un interprète I pour un langage L un interprète $I_{L \to L}$ pour L écrit en L. En appelant une boucle *read-eval-print* de $I_{L \to L}$ depuis I, une tour composée de deux interprètes actifs est mise en place. Le processus peut être réitéré plusieurs fois pour obtenir une tour composées d'un nombre fini de niveaux d'interprétation. Cette démarche proposée par Jefferson

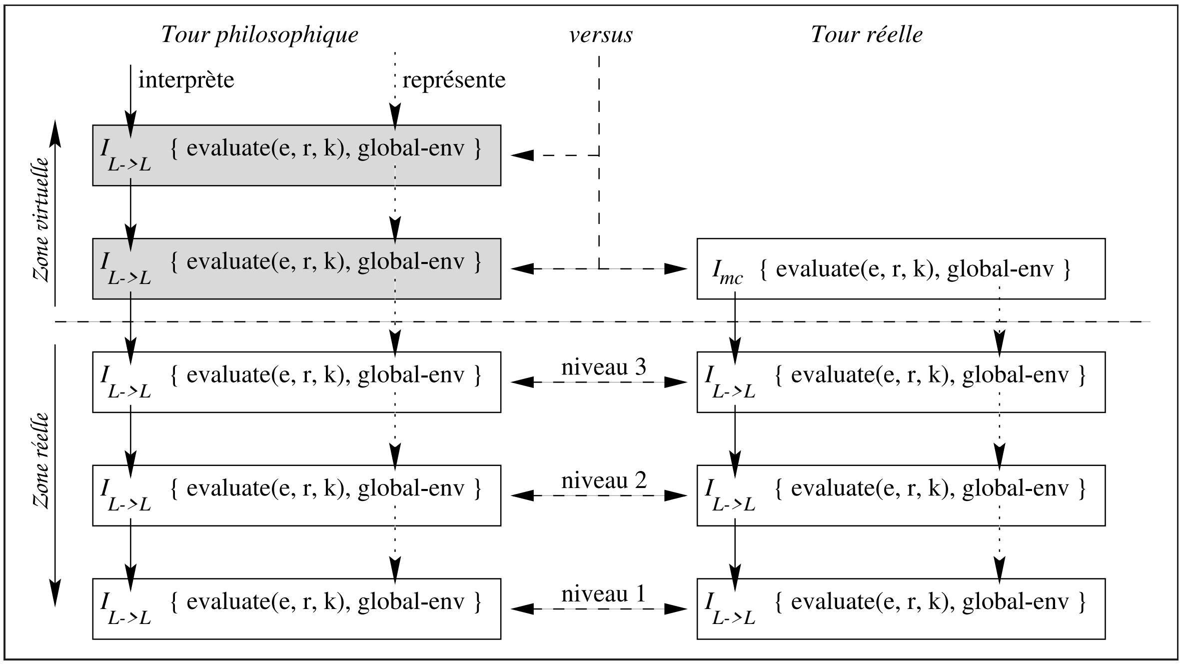

FIG. 4.1 – *Implémentation d'une tour active*

dans [Jefferson 92] conduit à l'obtention d'une tour active mais finie, le nombre de niveaux contenus par la tour étant déterminé à l'avance par l'utilisateur.

La tour que nous voulons obtenir doit être virtuellement infinie. Sa taille réelle doit rester cachée à l'utilisateur pour qu'il la perçoive comme infinie. Le modèle visible de la tour est donc un empilement apparemment infini d'interprètes $I_{L \to L}$ qui s'interprètent de niveau en niveau. Contrairement aux interprètes de la même famille que 3-Lisp [desRivières 84], la tour n'est pas représentée par une pile des états des interprètes mais est directement implémentée par un empilement d'interprètes actifs.

Pour l'implémenter, il faut pouvoir résoudre le problème du caractère infini de la tour. Ce caractère infini peut être résolu en se fondant sur deux observations:

1. Tout programme utilisateur qui se termine ne peut parcourir l'ensemble de la tour.

2. Les implémentations des autres interprètes réflexifs de langages fonctionnels savent simuler une tour infinie dont tous les niveaux sont identiques.

La première observation en induit une autre: il existe un niveau dans la tour à partir duquel tous les interprètes sont identiques, non modifiés par l'utilisateur et dont la définition correspond à la définition initiale de l'interprète telle qu'elle est présentée à l'utilisateur. Un programme correct, qui se termine, ne peut opérer sur l'ensemble des interprètes de la tour, aucune instruction du langage ne lui permettant d'agir sur elle ou même de la désigner.

En combinant les observations (1) et (2), une implémentation possible de la tour virtuellement infinie peut être déduite: la tour (Fig. 4.1) peut être composée d'un interprète méta-circulaire I_{mc} qui simule l'ensemble des interprètes identiques qui composent les niveaux n à l'infini de la tour, et d'un empilement d'interprètes $I_{L\to L}$ pour les niveaux qui vont de 1 à $(n-1)$, le niveau 0 étant constitué des programmes de l'utilisateur. Les techniques nécessaires à la mise en place d'une telle architecture sont assez courantes. Leur utilisation pour l'interprète méta-circulaire n'est cependant valable qu'à une condition: que l'utilisateur ne puisse venir modifier cet interprète.

L'accès à l'interprète méta-circulaire est contrôlable sachant que, pour monter d'un niveau, une lambda réflexive est nécessaire. L'interprète méta-circulaire peut être préservé en insérant un interprète $I_{L\to L}$ en lieu et place de l'interprète méta-circulaire dès que ce dernier doit exécuter une lambda réflexive. I_{mc} est alors remonté d'un niveau simulant les niveaux qui se trouvent non plus à partir de n mais de $(n+1)$. L'utilisateur n'a ainsi aucun moyen d'atteindre l'interprète méta-circulaire et donc de le modifier.

L'implémentation telle qu'elle est décrite jusqu'ici respecte l'idée que la tour est active, mais la possibilité de modifier les interprètes n'est pas encore prise en compte. Pour que les interprètes soient modifiables par l'utilisateur, leurs descriptions doivent être accessibles et causalement connectées. Les définitions des fonctions de chaque interprète doivent donc se trouver dans l'environnement global et leurs modifications induire une modification conséquente du comportement des interprètes.

Afin de permettre de modifier séparément chacun des interprètes de la tour, les descriptions des fonctions des interprètes sont contenues dans des environnements séparés. L'interprète de niveau i étant chargé depuis l'interprète de niveau $(i+1)$, ses définitions sont disponibles dans l'environnement global de l'interprète de niveau $(i+1)$. L'environnement global n'est plus partagé entre les différents niveaux d'interprétation comme dans [Jefferson 92]. Chaque interprète dispose de son propre environnement global.

Une autre raison qui s'oppose au partage de l'environnement global est la cohérence du modèle. Si l'environnement global était partagé et les interprètes causalement connectés aux définitions contenues dans l'environnement global, la modification de l'une des définitions devrait se traduire par la modification du comportement de l'ensemble des interprètes de la tour. L'interprète méta-circulaire devrait lui aussi modifier son comportement en accord avec les modifications effectuées par l'utilisateur. Or nous ne savons pas comment rendre l'interprète méta-circulaire modifiable.

4.1.2 Implémentation de I_{r^∞}

La tour d'interprètes de I_{r^∞} se compose d'un interprète méta-circulaire I_{mc} qui simule les niveaux qui vont du niveau n encore jamais atteint par l'utilisateur à l'infini, et d'un empilement d'interprètes $I_{L\to L}$, chaque interprète ayant son propre environnement global. Pour implémenter une telle architecture, nous devons disposer de l'interprète méta-circulaire I_{mc} et de l'interprète $I_{L\to L}$ qui sera copié autant de fois que nécessaire.

Plutôt que de créer un interprète ex nihilo, nous réutilisons pour $I_{L\to L}$ les travaux de Jefferson [Jefferson 92]. L'interprète qu'il décrit traite un sous-ensemble de Scheme [Abelson 91], sous-ensemble qu'il utilise pour l'écrire. Sa définition méta-circulaire convient parfaitement pour notre utilisation. $I_{L\rightarrow L}$ reprend directement toutes ses définitions aux fonctions de «bootstraping» près qui restent ignorées dans notre cas, cette phase étant assumée par I_{mc}.

L'essentiel du travail d'implémentation consiste pour nous à écrire l'interprète méta-circulaire I_{mc}. L'interprète méta-circulaire emprunte beaucoup à l'interprète de Jefferson. Le reste de cette section s'attache à décrire les aspects essentiels de cette implémentation réalisée à l'aide de l'interprète Scheme STk [Gallesio 96].

Le triplet (e, r, k)

Le rôle de l'interprète méta-circulaire est de simuler le haut de la tour en assurant sa construction au gré des besoins de l'utilisateur. Dès qu'il interprète une lambda réflexive, il insère un interprète $I_{L\rightarrow L}$ au niveau où il s'exécute et monte d'un niveau afin de se préserver de toute modification. Lorsqu'a lieu cette substitution, les trois éléments du triplet (e, r, k) sont réifiés avant d'être mis à disposition de l'interprète inséré. L'interprète méta-circulaire doit donc disposer de ces éléments dans une représentation identique à celle utilisée par les interprètes des niveaux inférieurs.

Les trois éléments du triplet, l'expression à évaluer, l'environnement lexical et la continuation du calcul, font l'objet d'une manipulation permanente par l'interprète méta-circulaire qui les utilise en paramètres de ses fonctions. La réification des registres de l'interprète est réalisable de deux façons:

(1) L'interprète méta-circulaire utilise une représentation interne des éléments du triplet et dispose de fonctions de conversion de ces éléments vers la représentation adoptée par les interprètes manipulés par l'utilisateur. Comme la plupart des interprètes Scheme [Abelson 91], STk [Gallesio 96] dispose d'outils qui permettent d'accéder à une forme réifiée des éléments requis par de telles fonctions (accès à l'environnement lexical, aux corps des lambdas). Et, au prix d'une réorganisation minime de ces informations, des fonctions de conversion des éléments du triplet peuvent être écrites.

Nous avons écrit une fonction de conversion:

| *(STk-to-Ill* $<e_{STk}>$ $<r_{STk}>$ $<k_{STk}>$*)* $\longrightarrow$ ($<e>$ $<r>$ $<k>$)

La fonction *STk-to-Ill* accepte en paramètre les éléments du triplet représenté en STk et retourne une liste composée des éléments du triplet représenté en $I_{L\rightarrow L}$. La fonction ne sera pas détaillée ici. L'implémentation est dépendante des instructions de STk qui mettent à disposition de l'utilisateur l'environnement lexical d'une procédure et sa définition. Ces objets subissent ensuite quelques transformations pour les rendre conformes à la représentation de $I_{L\rightarrow L}$.

(2) L'interprète méta-circulaire manipule directement les éléments du triplet dans la représentation utilisée par $I_{L\rightarrow L}$. Les éléments sont alors déjà sous leur forme réifiée.

Un exemple de technique adoptant ce procédé est donné dans [desRivières 84]. L'implémentation de l'interprète méta-circulaire qui y est décrite manipule directement les objets 3-Lisp. La conversion des objets en Lisp intervient lors de leur utilisation par l'interprète. Par exemple, la conversion des lambdas de 3-Lisp est effectuée lorsqu'elles sont appliquées. Le même principe peut être appliqué à notre interprète méta-circulaire.

Dans cette technique, les fonctions de l'interprète méta-circulaire sont découpées pour séparer les corps des fonctions des continuations qu'elles construisent. Prenons par exemple le cas de *evaluate-if*:

```
(set! evaluate-if
  (lambda (e r k)
    (evaluate (test-part e) r
              (lambda v)  ; Continuation de evaluate-if
                (if v
                  (evaluate (then-part e) r k)
                  (evaluate (else-part e) r k))))))
```

Si on laisse l'interprète méta-circulaire directement construire la continuation de la fonction *evaluate-if* (en gris dans le code), l'objet «continuation» sera une procédure Scheme dont la représentation ne correspondra pas à celle employée par $I_{L \to L}$. La solution consiste à construire la continuation comme un objet de $I_{L \to L}$. La définition de *evaluate-if* devient:

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

La construction de la continuation de *evaluate-if* est déléguée à une fonction dédiée. Cette fonction, *make-evaluate-if-cc*, construit une fermeture lexicale à partir des paramètres qui lui sont passés dans la représentation utilisée par $I_{L \to L}$. La définition de *make-evaluate-if-cc* est:

```
(set! make-evaluate-if-cc
  (lambda (e r k)
    (list 'compound
              ; Paramètre de la lambda
          '(v)
              ; Corps de la lambda
          '((if v
              (evaluate (then-part e) r k)
              (evaluate (else-part e) r k)))
              ; Environnement lexical capturé
          (extend empty-env
                  '(e r k)
                  (list e r k)))))
```

À chaque constructeur de continuation correspond une fonction de l'interprète méta-circulaire capable d'effectuer la tâche équivalente. La continuation de *evaluate-if* a pour fonction équivalente dans l'interprète méta-circulaire la fonction *evaluate-if-cc* qui a pour définition:

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

Les paramètres de *evaluate-if-cc* sont ceux de la continuation de *evaluate-if* suivis des variables lexicales capturées lors de la création de la fermeture. Malgré des similitudes dans l'écriture, il faut remarquer que *evaluate-if-cc* n'est pas un objet de L mais un objet de Scheme. L'interprète méta-circulaire peut donc directement l'appeler contrairement à la continuation qu'il construit dans *evaluate-if*.

L'application des continuations construites par l'interprète méta-circulaire ne peut plus se faire directement, les objets qui les représentent n'appartenant plus à Scheme. L'interprète utilise la fonction *call-cc*[1] pour établir une correspondance entre la continuation construite par l'interprète méta-circulaire et sa fonction équivalente.

```
(set! call-cc
      (lambda (kL v)
        ((lambda (kS-descriptor)
                  ; Appel de la fonction équivalente
           (apply (1st kS-descriptor)
                  (cons v  ; Valeur passée à la continuation
                           ; Recherche des valeurs des variables
                           ; lexicales dans l'environnement lexical
                           ; de la continuation.
                           (map (lambda (var)
                                  (get-pair var
                                    (procedure-environment kL)
                                    (lambda (pair) (cdr pair))
                                    (lambda ()
                                      (wrong "Shouldn't happen !"))))
                                (3rd kS-descriptor)))))
                 ; Recherche du descripteur de la continuation
                 ; dans la table des descripteurs de continuations
                 ; (cc-list)
         (get-S-closure-descriptor kL cc-list))))
```

La fonction *call-cc* commence par rechercher le descripteur de continuation contenu dans la variable globale *cc-list* qui correspond à la continuation appelée. Un descripteur de continuation contient les éléments permettant d'identifier une continuation et d'appeler la fonction Scheme de I_{mc} équivalente. C'est une liste de trois éléments dont le premier est la fonction équivalente, le second une instance de la continuation dont les variables lexicales reçoivent des valeurs de substitution et le troisième élément est la liste des noms des variables capturées lexicalement par la continuation.

```
kS-descriptor ::== ( <I_mc-fonction> <I_L→L-continuation>
```

1. Ne pas confondre avec le *call/cc* de Scheme. Ce dernier permet d'obtenir la continuation courante tandis que le notre se charge d'appeler la fonction Scheme équivalente à une continuation en L de I_{mc}.

| *<liste-des-noms-des-variables-lexicales>*)

Toutes les continuations possibles de l'interprète sont représentées dans cette liste. L'identification d'une continuation se base sur l'examen de son corps, de ses paramètres ainsi que sur le nombre et les noms des variables lexicales. L'interprète méta-circulaire ne pouvant être modifié par l'utilisateur, sa définition, et donc ses continuations sont fixes et connues à l'avance.

À partir du descripteur de la continuation appelée, la fonction *call-cc* effectue l'appel de la fonction Scheme équivalente. Les paramètres de cette fonction doivent préalablement être construits. Le premier de ces paramètres est la valeur passée à la continuation par la fonction qui l'appelle. Les autres paramètres dépendent de la continuation et correspondent aux variables lexicales capturées par la continuation.

Les appels des continuations ont désormais lieu grâce à la fonction *call-cc*. Toutes les fonctions de l'interprète méta-circulaire qui appellent une continuation doivent par conséquent être modifiées. Par exemple la fonction *evaluate-constant* est modifiée de la manière suivante:

de

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

elle est réécrite en:

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

La première technique décrite est relativement simple à mettre en œuvre dès que l'on dispose des outils nécessaires dans le langage qui sert à l'implémentation. La réécriture complète d'un interprète n'est pas nécessaire. Seule la fonction *evaluate-combination* qui contrôle l'interprétation des lambdas réflexives doit être modifiée. Cette modification peut intervenir après le chargement de l'interprète $I_{L \to L}$ qui sert alors d'interprète méta-circulaire.

L'autre avantage de cette technique est d'être plus efficace en terme d'exécution. Le coût de la traduction d'une représentation à une autre n'intervient que lors de l'ajout d'un nouveau niveau d'interprétation au sein de la tour. Le reste du temps, l'interprète méta-circulaire opère de façon identique à un interprète $I_{L \to L}$ directement chargé en Scheme.

Il y a cependant un aspect négatif: cette technique utilise des outils spécifiques de STk qui ne font pas partie de la définition de Scheme. Des outils équivalents existent dans la plupart des implémentations d'interprètes Scheme, mais la portabilité de la fonction de traduction est difficile à assurer.

La technique extrapolée de l'implémentation de 3-Lisp est elle portable. Son efficacité en terme d'exécution est cependant bien moindre: l'interprète méta-circulaire passe une grande partie de son temps à traduire les continuations en Scheme.

I_{r^∞} a été implémenté dans deux versions, chacune d'entre elles utilisant l'une de ces techniques. Des deux interprètes, celui basé sur la première technique est plus performant en terme d'efficacité de l'exécution que le second. L'utilisateur ne perçoit pas les différences d'implémentation de ces deux interprètes. Le coût principal de l'exécution d'une tour active d'interprètes ne provient pas tant de l'implémentation de l'interprète méta-circulaire que de l'empilement des niveaux d'interprétation.

Insertion d'un niveau d'interprétation dans la tour

La différence principale entre l'interprète méta-circulaire I_{mc} et les autres interprètes de la tour est sa capacité à insérer des niveaux d'interprétation dans la tour. L'illusion de manipuler une tour infinie est ainsi donnée à l'utilisateur.

L'interprète méta-circulaire ne simule que le haut de la tour. Il est construit de manière à n'être jamais modifié par l'utilisateur. Doté d'un mécanisme d'insertion d'un niveau d'interprétation dès qu'il doit exécuter une lambda réflexive, l'interprète méta-circulaire garantit sa cohérence avec la définition initiale des interprètes de la tour.

L'insertion d'un niveau au sein de la tour a lieu lorsqu'une lambda réflexive doit être exécutée par l'interprète méta-circulaire. La fonction *evaluate-combination* de l'interprète détermine si elle doit interpréter une lambda classique ou une lambda réflexive. Le processus normal de l'interprète est d'exécuter les lambdas réflexives comme si elles faisaient partie intégrante de l'interprète. Cette exécution est effectuée après avoir transformé la lambda réflexive en lambda simple:

| ((*reifier-to-compound proc*) (*operands-part e*) *r k*) *(i)*

Une telle ligne de code n'est pas exécutable par l'interprète méta-circulaire. Au dessus de l'interprète méta-circulaire se trouve l'interprète du langage qui sert à l'implémentation de I_{r^∞}. Cet interprète, ici STk, ne sait pas traiter directement les objets de notre interprète. C'est en substituant à l'interprète méta-circulaire I_{mc} un interprète $I_{L\to L}$ et en remontant I_{mc} d'un niveau que le problème est résolu.

La mise en place du mécanisme d'insertion d'un interprète est assurée en substituant la ligne de code (i) dans la fonction *evaluate-combination* de l'interprète méta-circulaire par un appel à la fonction *call-reifier*:

| (*call-reifier proc e r k*)

La fonction *call-reifier* a en charge l'insertion d'un niveau d'interprétation au sein de la tour. Cette insertion doit être transparente à l'utilisateur. Une fois l'insertion achevée, le calcul doit reprendre comme si tous les niveaux alors présents dans la tour avaient toujours été là.

Si la tour était infinie, le niveau d'interprétation n occupé par l'interprète méta-circulaire le serait par un interprète $I_{L\to L^{(n)}}$ et le niveau $(n+1)$ par un interprète $I_{L\to L^{(n+1)}}$. L'exécution de la ligne de code (i) par l'interprète $I_{L \to L^{(n)}}$ se traduirait par une évaluation du code correspondant par l'interprète $I_{L \to L^{(n+1)}}$ dans un environnement lexical constitué au moins des éléments nécessaires à cette évaluation. La continuation du calcul appelée par la fonction *evaluate* est similaire à celle mise en place par 3-Lisp [desRivières 84] lorsque son interprète méta-circulaire monte vers un niveau encore jamais atteint. Le code exécuté par $I_{L \to L}$ a donc la forme suivante:

```
(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))))                              (ii)
```

La continuation du calcul est construite par la fonction *make-openloop-cc* de l'interprète méta-circulaire. Elle correspond à la continuation de la fonction *openloop* de l'interprète qui est une boucle *read-eval-print*. Son implémentation, que nous ne détaillerons pas ici, diffère selon la technique utilisée pour traiter le triplet (expression, environnement, continuation): elle est construite comme un objet Scheme dans la première technique alors que dans la seconde, elle est construite comme un objet $I_{L \to L}$.

La fonction d'insertion d'un niveau supplémentaire d'interprétation dans la tour traduit cet ajout au niveau du flot de calcul en utilisant le code (ii). Mais, avant de relancer le calcul au sein de la tour d'interprètes, l'environnement global de l'interprète inséré doit être généré. Aucune modification n'ayant pu avoir eu lieu à ce niveau, cet environnement est identique à l'environnement global initial lui même identique à l'environnement global de l'interprète méta-circulaire. Pour l'essentiel, il contient les définitions des fonctions de l'interprète ainsi que les primitives du langage. L'insertion d'un niveau d'interprétation se traduit donc au niveau des environnements globaux par l'ajout par le haut d'un environnement global contenant tous les autres (la variable globale *global-env* de chaque environnement global désigne l'environnement global du niveau inférieur).

La définition de la fonction *call-reifier* est la suivante:

```
(set! call-reifier
      (lambda (proc e r k)
              ; Le nombre total de niveaux est incrèmenté
         (set! depth (+ depth 1))
              ; Insertion d'un interprète: l'environnement global
              ; de cet interprète est inséré
         ((lambda (ge)
             (set! global-env (extend (copy-env initial-env)
                                      '(global-env)
                                      (list ge))))
          global-env)
              ; Le calcul est relancé en tenant compte d'un niveau
              ; d'interprétation supplémentaire
         (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))))
```

Après l'appel de cette fonction, l'interprète méta-circulaire se trouve un niveau au dessus de son niveau précédent qui est alors occupé par un nouvel interprète de type $I_{L \to L}$. Le calcul est ensuite réamorcé un niveau plus haut, l'activité des interprètes de la tour étant initiée par l'activité de l'interprète méta-circulaire.

4.1.3 Performances de I_{r^∞}

Afin d'estimer les performances en terme d'exécution de chacune des implémentations de I_{r^∞}, nous avons déterminé le temps nécessaire au calcul de 5! avec des tours d'interprètes composées de 1 à 3 niveaux d'interprétation. Le nombre de niveaux d'interprétation contenus dans la tour est augmenté en exécutant une lambda réflexive. Elle provoque la création d'un niveau supplémentaire dans la tour.

La version basée sur 3-Lisp affiche les résultats suivants:

Niveaux	Temps (en secondes)
1	0,250
2	25,41
3	3021,15

La version basée uniquement sur STk affiche les résultats suivants:

Niveaux	Temps (en secondes)
1	0,031
2	3,51
3	366,01

Les deux tableaux précédents laissent apparaître deux résultats importants :

- La version uniquement basée sur STk est approximativement 8 fois plus rapide que celle basée sur 3-Lisp. Le maintien constant par cette dernière d'une représentation du triplet (erk) constitue donc un lourd handicap. Avec une conversion des données localisée à l'exécution d'une lambda réflexive par I_{mc}, la première version est beaucoup plus efficace.

- Mais, quelle que soit la version, l'ajout d'un niveau d'interprétation dans la tour se traduit par une dégradation d'un facteur 100 des performances. Même avec la version la plus rapide, le calcul de 5! prend plus de 6 minutes avec 3 niveaux d'interprétation. L'empilement constant des interprètes se fait donc lourdement sentir sur les performances du système.

L'optimisation de l'interprète méta-circulaire I_{mc} est donc une condition nécessaire mais pas suffisante pour obtenir un système efficace. Le coût de la réflexivité globale réside non dans cet interprète mais dans la dégradation des performances due à l'accumulation des niveaux d'interprétation. Quel que soit l'intérêt de la réflexivité globale, intérêt que nous allons tenter de montrer dans la section suivante, il est relativisé par les performances du système.

4.2 Mise en œuvre de la réflexivité

4.2.1 Un système de trace

Pour mettre en évidence l'absence d'une connexion causale complète dans la plupart des interprètes réflexifs existants, nous nous sommes appuyés sur un exemple de trace. Cet exemple est repris ici afin de montrer l'apport de la tour active dans la réalisation d'une connexion causale complète.

Le premier exemple effectue une trace sur un niveau. La fonction *evaluate* de l'interprète est modifiée de manière à afficher toute expression évaluée. Cette modification s'obtient par un effet de bord sur la définition de *evaluate* contenue dans l'environnement global de l'interprète après être monté d'un niveau dans la tour d'interprètes. La modification est obtenue de la manière suivante:

```
; exemple de la trace sur 1 niveau
: (boot-tower)
0-0> ((compound-to-reifier
        (lambda (e r k)
          (set! evaluate
                ((lambda (old-evaluate)
                   (lambda (e r k)
                     (display e)
                     (newline)
                     (old-evaluate e r k))) evaluate))
          (k 'done))))
1-0: done
1-0>
```

Puisque notre tour active réalise une connexion causale complète entre les définitions de l'interprète contenues dans l'environnement global et l'interprète, la modification de l'interprète est effective. Aussi, si une expression est entrée, l'évaluation de cette expression est tracée :

```
1-0> (+ 1 2)
(+ 1 2)
+
```

```
1
2
1-0: 3
1-0>
```

Les lignes précédentes correspondent à l'évaluation de l'expression (+ 1 2). Les étapes de cette évaluation apparaissent à l'écran suite à la modification de la fonction *evaluate* de l'interprète. La connexion causale entre deux niveaux d'interprétation contigus est donc bien réalisée dans notre interprète comme dans Black [Asai 93].

Cependant, notre interprète va plus loin que Black dans la réalisation de la connexion causale. L'exemple suivant montre cela en effectuant une trace sur deux niveaux d'interprétation. La fonction *evaluate* n'est plus modifiée sur un niveau d'interprétation mais sur deux. Ces modifications prennent la forme suivante :

```
; trace sur 2 niveaux
: (boot-tower)
0-0> ((compound-to-reifier
        (lambda (e r k)
          ((compound-to-reifier
            (lambda (e r k)
                ; Modification de evaluate du niveau 2
              (set! evaluate
                    ((lambda (old-evaluate)
                         (lambda (e r k)
                           (display e)
                           (newline)
                           (old-evaluate e r k))) evaluate))
              (k 'done))))
            ; Modification du evaluate du niveau 1
          (set! evaluate
                ((lambda (old-evaluate)
                     (lambda (e r k)
                       (display e)
                       (newline)
                       (old-evaluate e r k))) evaluate))
          (k 'done))))
; Début de la trace
(set! evaluate ((lambda (old-evaluate) (lambda (e r k) (display e) (newline) ...
...
0(display > )
display
>
> (evaluate (read) empty-env (lambda (v) (display-write-prompt level) (if (eq? v (void))
...
read
```

La mise en place de la trace est d'abord faite sur le second niveau d'interprétation. Cette modification qui est immédiatement effective induit la trace de la modification du premier niveau d'interprétation. Lorsque ces modifications sont en place, les deux niveaux d'interprétation sont tracés en même temps. L'évaluation de l'expression (+ 12) donne alors :

```
(+ 1 2) ; Expression à évaluer
empty-env
(lambda (v) (display-write-prompt level) (if (eq? v (void)) ...
(display e)
display
e
(+ 1 2)(newline)
newline
...
```

Pour des raisons de place la trace est abrégée. Mais nous pouvons cependant voir les informations des deux niveaux d'interprétation s'afficher. La connexion causale est donc bien réalisée dans I_{r^∞} sur l'ensemble des niveaux d'interprétation. Et la modification de l'interprète de niveau 2 a des conséquences visibles sur l'ensemble des niveaux qui se trouve en dessous.

L'ensemble des modifications réalisées ici affectent globalement l'interprète : ce n'est pas une portion de code qui est tracée mais toute expression passée à l'évaluateur. La réflexivité utilisée est donc bien de la réflexivité globale.

Ces modifications sont effectuées à l'aide de simple effet de bord sur les définitions de l'interprète. Aucun outil linguistique supplémentaire n'est nécessaire pour modifier globalement l'interprète contrairement à Refci [Simmons II 92b].

4.2.2 Communication entre différents niveaux d'interprétation

L'adoption d'un modèle où chaque interprète dispose de son propre environnement global soulève le problème de la communication entre les différents niveaux d'interprétation. Le partage de l'environnement global entre les interprètes de la tour, modèle suivi par de nombreux interprètes, facilite la communication entre les interprètes. La définition d'une nouvelle fermeture lexicale dans l'environnement global est alors naturellement accessible à tous les niveaux d'interprétation. Il en va de même pour les variables globales.

I_{r^∞} ne bénéficie pas de cette facilité. La programmation réflexive s'en trouve complexifiée. Cependant, il reste possible de communiquer entre deux niveaux d'interprétation, de partager des variables entre deux environnements globaux, etc. .

Le premier moyen d'extraire une information du niveau immédiatement au dessous est d'aller la chercher en évaluant le symbole qui la désigne. Si la variable a contient la valeur recherchée, elle peut naturellement être obtenue au niveau méta avec le code suivant:

```
(compound-to-reifier
    (lambda (e r k)
        ...
        (evaluate 'a r
                (lambda (value-of-a)
                    ...))
        ...))
```

L'utilisation systématique de *evaluate* pour obtenir la valeur de a est un peu lourde et coûteuse. Dans certains cas, la capture de l'environnement lexical du niveau de création par les lambdas réflexives permet de partager des informations. Mais ces informations restent localisées à la lambda réflexive qui les a capturées.

À l'aide de ces mêmes lambdas réflexives, un mécanisme de partage de paires pointées entre environnements peut être mis en place. I_{r^∞} stocke les couples (symbole, valeur) sous forme de paires pointées. L'environnement est structuré sous la forme d'une A-liste. Les paires pointées peuvent être partagées entre deux environnements globaux. La lambda réflexive *share-global-binding* qui suit réalise ce partage:

```
(set! share-global-binding                                  (n)
      (compound-to-reifier
        (lambda (e r k)                                     (n+1)
          (find-pair (car e) global-env
            (lambda (binding)
              (k ((compound-to-reifier
                    (lambda (e r k)                         (n+2)
                      (set! global-env (cons binding global-env))
                      (k 'done))))))
            (lambda ()
              (k 'unbound))))))
```

share-global-binding accepte comme premier paramètre un symbole qui désigne la variable globale à partager. La recherche de la paire pointée qui correspond à la variable partagée s'effectue par un appel à la fonction *find-pair* de l'interprète appliquée à *global-env* qui désigne alors l'environnement global du niveau n depuis lequel l'appel a eu lieu. Lorsque la paire pointée recherchée est trouvée, le partage est effectué.

Le partage est réalisé en ajoutant la paire pointée dans l'environnement global du niveau $(n+1)$ où *share-global-binding* s'exécute. Pour accéder à cet environnement global sous sa forme réifiée, il faut monter d'un niveau supplémentaire dans la tour. L'accès au niveau $(n+2)$ se fait en appelant une nouvelle lambda réflexive. L'environnement global du niveau $(n+1)$ est alors enrichi à l'aide d'un simple *cons* de la paire pointée sur *global-env* qui, dans le corps de la lambda réflexive s'exécutant au niveau $(n+1)$, contient la forme réifiée de l'environnement global du niveau $(n+1)$. La paire pointée se retrouve ainsi dans l'environnement global du niveau n et dans celui du niveau $(n+1)$. De ces deux niveaux, la même variable est alors accessible en lecture et en écriture:

```
; Exemple d'utilisation de share-global-binding
> (set! a 1)        ; Création de la variable globale a
:
> (share-global-binding a)  ; Partage de a
: done
> ((compound-to-reifier   ; Modification de a au niveau méta
     (lambda (e r k)
       (set! a 5))))
:
> a    ; Lecture de la nouvelle valeur de a au niveau courant
```

```
: 5
> _
```

share-global-binding peut bien sur s'applique à lui même. Sa définition se propage ainsi de niveau en niveau jusqu'au niveau désiré:

```
; Propagation de la définition de share-global-binding
> (share-global-binding share-global-binding)
:
> ((compound-to-reifier
      (lambda (e r k)
         (share-global-binding share-global-binding))))
:
>_
```

La définition de *share-global-binding* est montée dans l'exemple précédent jusqu'au niveau 2. En appliquant à nouveau ce même processus, elle peut être encore propagée sur autant de niveaux que nécessaire.

Ce mécanisme de partage des paires pointées est valable quelle que soit la valeur contenue. Il peut donc être aussi utilisé pour partager une fermeture lexicale ou même une lambda réflexive. Cependant, l'environnement global auquel accédera cette fermeture lexicale dépendra du niveau auquel elle sera exécutée. Si elle utilise d'autres variables globales contenant des valeurs ou d'autres fermetures lexicales, elles devront avoir été préalablement partagées afin que l'exécution de la fermeture lexicale partagée se déroule normalement.

Le partage des paires pointées ne résout pas tous les problèmes que pose l'existence d'un environnement global spécifique pour chaque niveau d'interprétation. Le cas de programmes faisant appel à un nombre de niveaux d'interprétation inconnu reste délicat. En étendant la fonction précédente de partage d'une variable globale au partage sur plusieurs niveaux d'interprétation, la question des lambdas réflexives récursives est partiellement résolue.

L'écriture d'une fonction de partage d'une paire pointée entre plusieurs niveaux d'interprétation est complexe. Les limitations du langage traité par $I_{r\infty}$ conduisent à découper une telle fonction en deux: l'une pour extraire la paire pointée de l'environnement et l'autre pour insérer la paire pointée dans les environnements respectifs des niveaux d'interprétation qui doivent la partager. Cette dernière fonction, réflexive pour accéder aux environnements globaux des interprètes, est récursive. Une lambda réflexive étant seulement définie dans son environnement de création et n'ayant pas à priori la connaissance d'elle même, elle n'a pas la capacité de s'appeler récursivement.

Une première manière de rendre une lambda réflexive capable de récursivité est de la redéfinir à chaque niveau d'appel. Le nombre de niveaux d'interprétation qui partagent la même paire pointée étant à priori inconnu, la redéfinition de la lambda réflexive doit se faire au fur et à mesure de la progression de la récursivité. La lambda réflexive est la seule à connaître son niveau courant d'appel et est donc la seule à savoir si elle doit se redéfinir ou non.

La construction de lambdas réflexives capables de se redéfinir de niveau en niveau est prise en charge par une nouvelle forme spéciale, *self-redefining!*. Sa syntaxe permet la définition simultanée de plusieurs lambdas réflexives, chaque nom donné à une lambda réflexive étant suivi de sa définition.

```
; Syntaxe de self-redefining!
(self-redefining!
        nom_1 λ_decl^1
        ...
        nom_k λ_decl^k)
```

Les lambdas contenues dans un même *self-redefining!* sont toutes redéfinies à chaque niveau d'interprétation où l'une d'entre elles est appelée. Cette caractéristique autorise la construction de lambdas réflexives qui font appel à des récursivités croisées. La définition de *self-redefining!* est la suivante:

```
(set! self-redefining!
      (compound-to-reifier
        (lambda (e r k)
          (evaluates-closures e r
            (lambda (cls)
              (mapper (lambda (binding)
                        (set! global-env
                              (cons
                                (cons (car binding)
                                      (make-self-redefine
                                        (cdr binding) cls))
                                global-env)))
                      cls)
              (k (void)))))))
```

À partir des déclarations, *self-redefining!* commence par construire une A-liste dont les paires pointées sont composées du nom de la lambda réflexive et de sa définition. C'est *evaluate-closures* qui se charge de cette construction. Le résultat de *evaluate-closures* est:

$((nom_k\ .\ \lambda_{decl^k})\ \ldots\ (nom_1\ .\ \lambda_{decl^1}))$

Les lambdas réflexives sont ensuite insérées dans l'environnement global non sous leur forme déclarée mais sous la forme d'une lambda réflexive, λ_{redef^i}, capable de redéfinir de niveau d'appel en niveau d'appel les lambdas réflexives déclarées. Ces lambdas réflexives sont construites par *make-self-redefine*. L'environnement global prend alors la forme suivante:

Niveau	*global-env*
$(n+1)$	$\ldots$
n	$((nom_k\ .\ \lambda_{redef^k})\ \ldots\ (nom_1\ .\ \lambda_{redef^1})\ \ldots)$

Le traitement des paramètres non évalués de *self-redefining!* est effectué par la fonction *evaluate-closures*. Cette lambda classique est appelée depuis le corps d'une lambda réflexive.

Elle doit donc être définie dans l'environnement global de l'interprète depuis lequel la lambda réflexive est exécutée. Pour définir *evaluate-closures* au niveau de l'interprète, c'est à dire, un niveau au dessus du niveau utilisateur, il faut préalablement monter d'un niveau. Ce passage au niveau supérieur est effectué en enveloppant la définition de la lambda dans un appel à une lambda réflexive. La définition de *evaluate-closures* est la suivante:

```
((compound-to-reifier
   (lambda (e r k)
     (set! evaluate-closures
          (lambda (defs r k)
            (if (null? defs)
               (k '())
               (evaluate (car (cdr defs)) r
                  (lambda (cl)
                    (evaluate-closures (cdr (cdr defs)) r
                      (lambda (cls)
                        (k (cons (cons (car defs) cl)
                                 cls)))))))))
     (k 'done))))
```

evaluate-closures parcourt les définitions des lambdas réflexives en les évaluant une à une dans l'environnement lexical capturé par *self-redefining!*. Le résultat de chaque évaluation est passé à une continuation qui construit une A-liste en réalisant un *cons* du nom donné à la lambda réflexive avec le résultat de l'évaluation.

La construction des lambdas réflexives se définissant de niveau en niveau est dévolue à *make-self-redefine*. Dans un premier temps, les lambdas réflexives déclarées par *self-redefining!* sont remplacées dans l'environnement global par des lambdas réflexives qui se redéfinissent au niveau où elles sont exécutées. Dès qu'elles sont appelées, ces dernières remplacent leurs définitions dans l'environnement global d'appel par les définitions données par l'utilisateur. Elles appellent ensuite la forme non réflexive des lambdas réflexives qu'elles remplacent en leur passant en paramètres ceux qu'elles ont reçus. *make-self-redefine* construit les lambdas réflexives servant à la redéfinition de niveau en niveau des lambdas réflexives déclarées. Sa définition est:

```
((compound-to-reifier
   (lambda (e r k)
     (set! make-self-redefine
          (lambda (cl closures)
                ; Construction de la lambda réflexive qui redéfinit de niveau
                ; en niveau celles de l'utilisateur
            (compound-to-reifier
              (lambda (e r k)
                    ; Remontée des définitions des fermetures au niveau
                    ; courant et redéfinition au niveau d'appel
                (mapper (lambda (b)
                               ; Récupération de la valeur dans
                               ; l'environnement global
                          (find-pair (car b) global-env
```

```
                                        (lambda (pair)
                                          ((compound-to-reifier
                                             (lambda (e r k)
                                               ; Insertion de la valeur dans
                                               ; l'environnement global du niveau
                                               ; courant
                                               (set! global-env
                                                     (cons (cons (car b) (cdr pair))
                                                           global-env))
                                               (k (void)))))
                                           ; Attribution de la valeur initiale de la
                                           ; fermeture au niveau d'appel
                                           (set-cdr! pair (cdr b)))
                                        (lambda () 'unbound)))
                                   closures)
                              ; Appel effectif de la fermeture
                              ((reifier-to-compound cl) e r k)))))
      (k 'done))))
```

Appelé par *self-redefining!* à son niveau d'exécution, *make-self-redefine* doit être défini à un niveau au dessus du niveau de définition de *self-redefining!*, ce qui est réalisé en enveloppant sa définition par un appel à une lambda réflexive.

La lambda réflexive qui permet la redéfinition automatique des lambdas déclarées capture dans son environnement lexical la lambda réflexive qu'elle remplace momentanément ainsi que les noms et définitions des autres lambdas réflexives déclarées dans la même forme spéciale *self-redefining!*.

$$\lambda_{redef^i} = \lambda_{redef^i}[\lambda_{decl^i}\ ((nom_k.\lambda_{decl^k})\ldots(nom_1.\lambda_{decl^1}))]$$

Lorsqu'elle est appelée, elle commence par remonter sous leurs formes courantes, c'est à dire, sous la forme de lambdas réflexives se redéfinissant, l'ensemble des lambdas réflexives construites en même temps qu'elle. En redéfinissant l'ensemble des lambdas déclarées et non la seule lambda réflexive appelée, les appels récursifs croisés entre ces lambdas sont possibles, toutes les lambdas déclarées étant maintenant disponibles au niveau d'exécution de la lambda appelée.

Niveau	*global-env*
$(n+1)$	$((nom_k\ .\ \lambda_{redef^k})\ \ldots\ (nom_1\ .\ \lambda_{redef^1})\ \ldots)$
n	$((nom_k\ .\ \lambda_{redef^k})\ \ldots\ (nom_1\ .\ \lambda_{redef^1})\ \ldots)$

La seconde étape, qui se déroule parallèlement à la première, consiste à modifier la définition des lambdas réflexives dans l'environnement global d'appel. Leur valeur initiale leur est rendue. Elles sont désormais disponibles au niveau d'appel sous leurs formes déclarées.

Niveau	*global-env*
$(n+1)$	$((nom_k\ .\ \lambda_{redef^k})\ \ldots\ (nom_1\ .\ \lambda_{redef^1})\ \ldots)$
n	$((nom_k\ .\ \lambda_{decl^k})\ \ldots\ (nom_1\ .\ \lambda_{decl^1})\ \ldots)$

Finalement, la lambda réflexive déclarée est appelée sous sa forme non réflexive. Elle est donc exécutée au niveau courant avec les paramètres d'appel. La redéfinition reste transparente aux lambdas réflexives ainsi construites. Si ces lambdas s'appellent récursivement jusqu'au niveau n, les lambdas réflexives déclarées seront définies du niveau 0 au niveau n compris. Le niveau n contiendra non pas les lambdas sous la forme de leurs définitions initiales, mais sous la forme de lambdas réflexives qui permettent leur définition au niveau supérieur.

La forme spéciale *self-redefining!* peut être utilisée pour modifier *share-global-binding* de manière à permettre le partage d'une même paire pointée entre plusieurs niveaux d'interprétation successifs. Cette modification consiste à traiter un paramètre supplémentaire spécifiant le nombre de niveaux partageant la paire pointée et à appeler une autre lambda réflexive, *sharing-binding*, chargée d'effectuer le partage entre ces niveaux. La définition de *share-global-binding* devient:

```
(self-redefining!
  share-global-binding
    (compound-to-reifier
      (lambda (e r k)  ; e ::== (<nb-niveaux> <symbole>)
                 ; Récupération de la paire dans l'environnement global
        (find-pair (car (cdr e)) global-env
          (lambda (pair)
            (evaluate (car e) r
              (lambda (level)
                         ; Insertion de la paire dans les environnements
                         ; des <nb-niveaux> supérieurs
                (k (sharing-binding level binding))))))))
  sharing-binding
    (compound-to-reifier
      (lambda (e r k)  ; e ::== (<nb-niveaux> <paire>)
        (evaluate (car (cdr e)) r
          (lambda (binding)
                     ; Ajout de la paire pointée
                     ; dans l'environnement global
            (set! global-env (cons binding global-env))
            (evaluate (car e) r
              (lambda (level)
                (k (if (= level 1)
                       'done
                           ; Même procédé au niveau supérieur
                       (sharing-binding (- level 1)
                                        binding))))))))))
```

La définition de *share-global-binding* est à peu de choses identique à la précédente. Elle diffère sur la fin où le nombre de niveaux devant partager la paire pointée est évalué et la lambda réflexive effectuant le partage est appelée. Cette dernière évalue le nombre de niveaux restant et, si ce nombre est différent de 0, insère la paire pointée partagée dans l'environnement global avant de s'appeler elle même. La nature réflexive de cette lambda lui fait monter un niveau d'interprétation à chacun de ses appels.

L'exemple qui suit illustre la mise en œuvre du nouveau *share-global-binding*. La variable *a* créée au niveau utilisateur est partagée avec les niveaux d'interprétation 1 et 2. En incrémentant sa valeur de 1 à chaque niveau d'interprétation, sa valeur au niveau utilisateur devient 2.

```
> (set! a 0)
:
> (share-global-binding 2 a)
:
> ((compound-to-reifier
     (lambda (e r k)
       (set! a (+ a 1))   ; Modification de a au niveau 1
       ((compound-to-reifier
          (lambda (e r k)
            (set! a (+ a 1))  ; Modification de a au niveau 2
            (k 'done))))
       (k 'done))))
:
> a
: 2
> _
```

Grâce à *self-redefining!*, les deux lambdas réflexives *share-global-binding* et *sharing-binding* sont désormais définies du niveau 0 au niveau 2. Elles peuvent donc toutes deux être directement appelées à l'un de ces niveaux.

La forme spéciale *self-redefining!* présente deux avantages: elle permet de diffuser la définition de lambdas réflexives sur plusieurs niveaux d'interprétation et elle permet l'écriture de lambdas réflexives mutuellement récursives. Cependant elle nécessite un processus d'interprétation supplémentaire pour remonter les définitions de niveau en niveau. Le coût de cette opération n'est pas négligeable et alourdit l'interprétation des lambdas réflexives.

Le cas de *share-global-binding* ne nécessite pas une infrastructure aussi lourde. La récursivité nécessaire au partage de la paire pointée n'a lieu que sur une seule lambda. Or il est possible de permettre à une lambda de s'appeler récursivement sans qu'elle soit redéfinie à chaque niveau.

Pour qu'une lambda réflexive puisse s'appeler récursivement, il faut qu'elle se connaisse. Lors de la construction des lambdas réflexives, l'environnement lexical dans lequel elles sont déclarées est capturé. L'évaluation d'une lambda réflexive a lieu dans cet environnement lexical. En ajoutant à cet environnement une paire pointée constituée du nom donné à la lambda réflexive et de la lambda, on lui donne la connaissance d'elle même. En se connaissant elle même, une lambda réflexive a la capacité de s'appeler récursivement. La forme spéciale *set*!* permet la construction de telles lambdas. Sa définition est:

```
(set! set*!
      (compound-to-reifier
        (lambda (e r k)   ; e ::== (<symbole> <lambda>)
          ((lambda (name)
             ((lambda (binding)
                         ; Evaluation de l'expression dans
                         ; l'environnement enrichi d'une paire
                (evaluate (car (cdr e)) (cons binding r)
                  (lambda (cl)
                              ; Attribution de la valeur à la paire
                    (set-cdr! binding cl)
                              ; Déclaration dans global-env
                    (set! global-env (cons (cons name cl)
                                           global-env))
                    (k (void)))))
              (cons name '***)))
           (car e)))))
```

L'environnement lexical est d'abord enrichi d'une paire pointée composée du symbole nommant la lambda réflexive et d'une valeur fictive. Le second paramètre de *set*!* est alors évalué dans l'environnement lexical modifié. Par effet de bord sur la paire pointée, la valeur résultant de l'évaluation est stockée dans le *cdr* de la paire pointée. L'environnement global est ensuite modifié pour contenir la nouvelle définition de façon similaire à un **set!** du langage.

Le fonctionnement du *set*!* lui permet un usage plus général que celui des seules lambdas réflexives. Toute construction capturant l'environnement lexical et nécessitant une référence à elle même peut l'utiliser.

L'application de *set*!* à *share-global-binding* est assez directe. La définition de la fonction *share-global-binding* est strictement identique à la précédente et doit être créée à l'aide d'un **set!** classique. La lambda réflexive récursive *sharing-binding* doit être créée à un niveau d'interprétation au dessus en utilisant la nouvelle forme spéciale *set*!*. Cette forme spéciale étant créée au niveau utilisateur, il est nécessaire de la remonter d'un niveau avant de l'utiliser.

```
((compound-to-reifier
   (lambda (e r k)
             ; Récupération de la valeur de set*!
     (evaluate 'set*! r
       (lambda (value)
                 ; Définition de set*! au niveau courant
         (set! set*! value)
                 ; Définition de sharing-binding à l'aide de set*!
         (set*! sharing-binding
           (compound-to-reifier
             (lambda (e r k)  ; e ::== (<nb-niveau> <paire>)
               (evaluate (car (cdr e)) r
```

```
                (lambda (binding)
                  (set! global-env (cons binding global-env))
                  (evaluate (car e) r
                    (lambda (level)
                      (k (if (= level 1)
                             'done
                             (sharing-binding (- level 1)
                                              binding))))))))))
            ; Retour au niveau utilisateur
        (k 'done))))))
```

share-global-binding est alors utilisable de la même manière que lors de sa construction avec *self-redefining!*. La différence essentielle est qu'il ne sera pas redéfini de niveau en niveau. Son utilisation est alors restreinte au niveau utilisateur.

En choisissant une architecture dans laquelle chaque interprète a un environnement global spécifique, nous avons complexifié la tâche du programmeur. L'objectif de I_{r^∞} étant non d'offrir une nouvelle implémentation d'interprète réflexif mais de fournir un modèle clair et cohérent qui corresponde de près à la philosophie de la tour d'interprètes, sa facilité d'utilisation n'est pas un critère primordial. Les exemples précédents montrent qu'il est cependant possible de composer avec les contraintes imposées par cette architecture.

4.2.3 Les abstractions quasi statiques

Les abstractions quasi statiques [Lee 93] fournissent un moyen de partager des variables lexicales appartenant à des portées lexicales distinctes. En Scheme [Abelson 91], seules les variables globales permettent à deux fermetures lexicales créées dans des contextes lexicaux différents de partager une valeur. En déclarant des variables quasi statiques dans un contexte lexical et en résolvant leur valeur dans un autre contexte lexical, les abstractions quasi statiques autorisent le partage de valeurs sans passer par l'environnement global respectant ainsi l'esprit de Scheme.

Pour gérer les variables quasi statiques, les abstractions quasi statiques introduisent deux nouvelles formes spéciales, *qs-abstraction* et *resolve*. La première forme sert à la déclaration de variables quasi statiques lors de la construction d'abstractions quasi statiques. Ces variables ne sont pas liées dans le contexte lexical de leur déclaration. À l'aide de la seconde forme, les variables quasi statiques d'une abstraction quasi statique reçoivent une valeur dans un autre contexte lexical que celui de leur déclaration. La valeur donnée à une variable quasi statique est alors partagée entre les deux contextes lexicaux.

Dans ([Simmons II 92b], [Simmons II 93]), Simmons et Friedman nous proposent leur étude de l'impact du choix d'une représentation dans les interprètes réflexifs sur les possibilités de programmation réflexive. Ils appuient leur argumentation sur l'implémentation des abstractions quasi statiques dans les interprètes réflexifs. En examinant les caractéristiques requises pour implémenter les abstractions quasi statiques, ils montrent comment la représentation choisie pour un interprète réflexif influe sur la faculté de ces interprètes à accepter cette extension.

Avec l'implémentation des abstractions quasi statiques dans notre interprète muni d'une tour active, nous allons montrer comment la réflexivité globale permet de pallier à certaines limites dues au choix d'une représentation donnée.

Simmons et Friedman décrivent deux implémentations possibles des abstractions quasi statiques qui divergent essentiellement dans le traitement des variables quasi statiques. Dans la première, un système de marqueurs permet de différencier les variables statiques des autres. La fonction *lookup* est alors modifiée pour tenir compte des différents types de variables et traiter les variables quasi statiques comme des variables non liées. La présence d'une construction linguistique dédiée à la modification de l'interprète permet dans Refci [Simmons II 92a], l'interprète réflexif utilisé par Simmons, une forme de réflexivité globale. La deuxième approche se base sur l'idée que les variables quasi statiques sont comparables à des variables non liées dans l'environnement lexical courant. Les variables portant le même nom que les variables déclarées quasi statiques sont supprimées de l'environnement lexical. La fonction *lookup* peut alors être utilisée directement. Cette solution laisse une ambiguïté sur le traitement des variables globales portant le même nom que des variables déclarées quasi statiques.

Ces deux solutions bénéficient d'une facilité qu'offre la représentation de l'environnement par des fermetures lexicales : les couples (symbole, valeur) y sont partageables sans difficulté. Dans I_{r^∞}, notre interprète, l'environnement est représenté par une A-liste. Le *cdr* des paires pointées y contient la valeur associée au symbole. Cette structure ne facilite pas le partage des valeurs des couples (symbole, valeur). L'affectation se traduisant par une application de la primitive *set-cdr!* sur les paires pointées, la simple recopie du pointeur sur la valeur ne correspond pas à un partage des valeurs.

La réalisation du partage effectif des valeurs dans I_{r^∞} passe nécessairement par une modification de la représentation des couples (symbole, valeur) pour les variables quasi statiques. Les fonctions de traitement de l'environnement dans notre interprète doivent conséquemment être adaptées. Sans la réflexivité globale, de telles modifications seraient impossibles, les lambdas réflexives ne le permettant pas.

Pour partager une valeur entre deux fermetures lexicales distinctes, nous utilisons une indirection sur la paire pointée qui la contient. Cette indirection est réalisée à l'aide d'une paire pointée dont le premier élément constitue un marqueur. Ce marqueur, *<*qs-binding*>*, permet de distinguer les variables quasi statiques des autres structures. Le *cdr* de la paire pointée est constitué de la paire pointée contenant la valeur partagée.

```
( <qs-variable-name> '*<qs-binding> . ( <symbol> . <value> ) )
```

Cette représentation autorise la gestion de multiples indirections sur la valeur partagée. D'une part plusieurs fermetures lexicales ont la possibilité de partager une même valeur, mais une variable quasi statique non encore résolue peut être désignée comme valeur d'une autre variable quasi statique, la connaissance effective de la valeur partagée n'étant requise qu'au moment de l'évaluation d'une des fermetures lexicales s'y référant.

Afin d'accepter notre représentation des variables quasi statiques, l'interprète doit être modifié en conséquence. L'utilisation d'indirections impose la réécriture de la fonction de recherche d'une paire pointée dans l'environnement. Les traitements de l'affectation et du retour d'une valeur sont préservés par le retour de la paire pointée contenant la valeur effective des variables quasi statiques. La fonction de recherche modifiée permet à l'effet de bord effectué lors d'une affectation d'avoir lieu sur la bonne paire pointée.

Pour changer une définition de l'interprète des expressions du niveau courant, il faut effectuer un effet de bord dans le contexte de cet interprète. Chaque interprète de I_{r^∞} ayant son propre environnement global, l'effet de bord doit avoir lieu au niveau d'interprétation correspondant. Les lambdas réflexives sont le seul outil apte à s'exécuter dans le contexte de l'interprète du niveau courant. En enveloppant les effets de bord souhaités dans un appel à une lambda réflexive, nous obtenons la modification des définitions des fonctions de l'interprète du niveau supérieur.

```
((compound-to-reifier
        ; Passage au niveau d'interprétation supérieur
  (lambda (e r k)
          ; Une pincée de sucre syntaxique
    (set! qs-binding? (test-tag '*<qs-binding>*))
          ; Filtrage des indirections dues aux
          ; variables quasi statiques
    (set! get-actual-binding
      (lambda (binding success failure)
        (if binding
          ((lambda (binding-value)
             (if (qs-binding? binding)
                   ; Gestion des indirections
                 (get-actual-binding (cdr binding-value)
                                     success failure)
                 (success binding)))
           (cdr binding))
          (failure))))
          ; Modification de la fonction de recherche dans
          ; un environnement
    (set! find-pair
      (lambda (elt alist success failure)
        (get-actual-binding (assq elt alist) success failure)))
        ; Retour au niveau d'appel avec une valeur arbitraire
    (k 'done))))
```

La recherche d'une paire pointée dans un environnement est dans I_{r^∞} l'œuvre de *find-pair*. Lorsqu'elle trouve une paire pointée qui correspond au symbole recherché, elle la passe à une fermeture chargée de la traiter. En cas d'échec, une fermeture spécifique est appelée. L'intégration des variables quasi statiques impose un traitement supplémentaire en cas de succès: la paire pointée associée au symbole recherché doit être examinée pour déterminer s'il s'agit d'une indirection vers une valeur partagée. Lors de la présence d'une indirection, la paire pointée indirectement désignée doit être à son tour examinée. La fonction *get-actual-binding*, en s'appelant récursivement sur chaque indirection, garantit le retour de la paire pointée qui contient la valeur partagée. *find-pair* fait désormais appel à *get-actual-binding* pour prendre en compte les variables quasi statiques.

Les valeurs pouvant être partagées entre des fermetures lexicales distinctes, les abstractions quasi statiques peuvent être implémentées. Cela nécessite la mise en place de deux formes spéciales: *qs-abstraction* et *resolve*.

Des deux implémentations proposées par Simmons et Friedman [Simmons II 92b], nous avons opté pour celle basée sur une conservation des informations sur les variables quasi statiques au sein de l'environnement lexical. Les variables quasi statiques sont insérées lors de leur déclaration par la forme spéciale *qs-abstraction* dans l'environnement lexical courant.

Le mécanisme de partage des variables que nous venons de décrire va être directement utilisé à cet effet. Lors de la détection d'une indirection, *get-actual-binding* s'appelle récursivement pour traiter la paire pointée désignée. L'utilisation de #f au lieu d'une paire pointée contraint *get-actual-binding* à se comporter comme si il n'avait pas trouvé de paire pointée correspondant au symbole recherché. Le traitement correspondant à un échec de la recherche sera alors enclenché lors de l'évaluation d'une variable quasi statique non résolue.

La forme spéciale *qs-abstraction* adopte la syntaxe suivante:

```
(qs-abstraction ( <qs-variable>* ) <lambda> )
```

L'évaluation de la lambda de *qs-abstraction* a lieu dans l'environnement lexical courant enrichit des variables quasi statiques déclarées. La lambda réflexive qui implémente la forme spéciale commence par étendre l'environnement lexical courant avec les paires pointées correspondant aux variables quasi statiques. La structure les décrivant comme des variables quasi statiques non liées est obtenue par un *cons* du marqueur des variables quasi statiques, *<*qs-binding*>*, avec le booléen #f. La recherche de la valeur d'une variable quasi statique non résolue se traduira par un échec.

L'environnement lexical étendu sert d'environnement d'évaluation à la lambda déclarée par *qs-abstraction*. Le résultat de l'évaluation est une fermeture lexicale qui a capturé l'environnement lexical étendu. Nous construisons les abstractions quasi statiques comme des fermetures lexicales. Elles sont applicables directement par l'interprète au même titre que les autres fermetures lexicales sans qu'aucune modification ne soit requise. Tant que les variables quasi statiques ne sont pas résolues, l'application d'une abstraction quasi statique résulte en une erreur.

```
(set! qs-abstraction
  (compound-to-reifier
    (lambda (e r k)
      (evaluate (car (cdr e))
                ((lambda (qs-var-names)
                   (extend r qs-var-names
                           (mapper (lambda (symbol)
```

```
                                        (cons '*<qs-binding>* #f))
                                    qs-var-names)))
                   (car e))
                 k))))
```

La simple évaluation de la lambda d'une abstraction quasi statique fait bénéficier l'utilisateur d'une certaine souplesse. La lambda peut être construite de manière complexe, par exemple en l'encapsulant dans un environnement lexical supplémentaire. La seule obligation est la capture de l'environnement lexical étendu.

La résolution des variables quasi statiques est effectuée par la forme spéciale *resolve*. Elle permet de spécifier la variable qui doit être partagée entre l'environnement lexical courant et celui d'une abstraction quasi statique qui lui est passée en paramètre. L'environnement lexical de l'abstraction quasi statique pouvant contenir plusieurs variables quasi statiques, la variable quasi statique faisant l'objet de la résolution doit aussi être spécifiée.

La forme spéciale *resolve* adopte la syntaxe suivante:

```
(resolve <variable> <qs-variable> <qs-abstraction>)
```

resolve commence par évaluer l'expression qui donne l'abstraction quasi statique afin d'obtenir l'environnement lexical contenant la variable quasi statique servant au partage de la valeur. La valeur à partager se trouve dans l'environnement lexical courant. Un simple *assq* sur cet environnement permet d'obtenir la paire pointée contenant la valeur à partager. C'est sur cette paire pointée que l'indirection de la variable quasi statique doit avoir lieu.

```
(set! resolve
  (compound-to-reifier
    (lambda (e r k)
      (evaluate (car (cdr (cdr e))) r
                (lambda (qsa)
                  (k ((lambda (binding qs-variable)
                        (if binding
                          (if qs-variable
                            ((lambda ()
                               (set-cdr! (cdr (cdr qs-variable))
                                         binding)
                               (void)))
                            (void))
                          (void)))
                      (assq (car e) r)
                      (get-qs-variable
                               (car (cdr e))
                               (procedure-environment qsa)))))))))
```

La recherche de la variable quasi statique est plus complexe. L'interprète est enrichi d'une fonction dédiée à cette recherche, la fonction *get-qs-variable*. Cette fonction recherche dans l'environnement lexical de l'abstraction quasi statique, la première paire pointée désignant une variable quasi statique correspondant au symbole passé en paramètre.

```
((compound-to-reifier
   (lambda (e r k)
     (set! get-qs-variable
           (lambda (symbol a-list)
             (if (null? a-list)
                 #f
                 ((lambda (binding)
                    (if (eq? symbol (car binding))
                        (if (qs-binding? (cdr binding))
                            binding
                            (get-qs-variable symbol (cdr a-list)))
                        (get-qs-variable symbol (cdr a-list))))
                  (car a-list)))))
     (k 'done))))
```

Lorsqu'elle dispose de la paire pointée dont la valeur doit être partagée et de la paire pointée contenant la variable quasi statique, la forme spéciale *resolve* modifie la valeur de la variable quasi statique en une indirection sur la valeur partagée. La variable quasi statique résolue est désormais utilisable de la même manière que les autres variables, la fonction de recherche dans l'environnement tenant compte des indirections pour retourner la paire pointée contenant la valeur partagée.

Les abstractions quasi statiques se révèlent particulièrement adaptées à la résolution de problèmes nécessitant un partage de valeurs. Simmons et Friedman montrent plusieurs exemples d'utilisations, y compris dans un langage à objets [Simmons II 92b]. L'exemple qui suit sert uniquement à montrer le fonctionnement des abstractions quasi statiques:

```
> (set! qsa-1 (qs-abstraction (a) (lambda (a) a)))
:
> ((lambda (e) (resolve e a qsa-1)) 5)
:
> (qsa-1)
: 5
> _
```

Quelle que soit l'implémentation choisie, les abstractions quasi statiques n'auraient pu être implémentées dans notre interprète sans la présence de la réflexivité globale. La représentation des environnements n'autorise pas un partage des valeurs entre les paires pointées sans modifier les fonctions de gestion des environnements. La redéfinition de ces fonctions ne peut avoir lieu que dans le cadre de la réflexivité globale. L'utilisation des lambdas réflexives limitée à l'introduction de nouvelles formes spéciales n'est pas assez puissante pour résoudre ce problème.

La réflexivité globale, en permettant d'élargir les possibilités de modifications de l'interprète, offre la possibilité de contourner certaines limitations dues au choix d'une représentation. Elle offre donc plus de souplesse que la réflexivité uniquement basée sur les lambdas réflexives. Elle se révèle, du point de vue de la programmation réflexive, être une caractéristique souhaitable dans les interprètes réflexifs.

4.3 Comparaison avec d'autres travaux

I_r[Jefferson 92] sert de base à $I_{r\infty}$. Ces deux interprètes diffèrent néanmoins par leurs objectifs et leurs implémentations. Si dans le premier l'accent est mis sur la construction d'un interprète réflexif, le second a pour objectifs la cohérence entre l'implémentation et la philosophie de la tour d'interprètes ainsi que l'élaboration d'un lien causal complet. Leurs tours d'interprètes respectives sont sensiblement différentes: $I_r$utilise une tour finie construite à la main par l'utilisateur, alors que $I_{r\infty}$ utilise une tour infinie construite automatiquement par l'interprète méta-circulaire.

Avec Refci [Simmons II 92a] la réflexivité globale fait son entrée dans les interprètes réflexifs. La possibilité de modifier globalement les définitions de l'interprète est un des objectifs avoués de Refci. Cependant cette capacité s'y traduit par l'ajout d'une construction linguistique dédiée. Avec $I_{r\infty}$, nous avons montré que la possibilité de modifier l'interprète tient dans la réalisation d'une connexion causale complète et non dans l'ajout d'une construction linguistique supplémentaire. La présence de la réflexivité globale relève de l'implémentation et non de la modélisation. C'est par une implémentation complète de la connexion causale que la réflexivité globale est réalisée dans $I_{r\infty}$ et non par l'introduction d'un outil linguistique supplémentaire.

4.4 Conclusion

Dans ce chapitre nous avons montré comment implémenter un interprète réflexif muni d'une réflexivité globale en nous appuyant sur une tour active d'interprètes. C'est par l'implémentation d'une connexion causale complète que la réflexivité globale est introduite dans $I_{r\infty}$. Aucun outil linguistique supplémentaire n'a été nécessaire à cette réalisation.

La réflexivité globale se révèle être un outil complémentaire de la réflexivité locale. Les exemples présents dans ce chapitre démontrent l'utilité de la réflexivité globale dont $I_{r\infty}$ est doté. En autorisant la modification effective de l'interprète, elle permet de pallier aux limitations introduites par le choix d'une représentation. Le champ des programmes réflexifs envisageables est fortement élargi par la présence de ce type de réflexivité. Elle augmente donc le degré de liberté qu'offre la réflexivité.

Ce type de réflexivité a cependant un coût non négligeable. L'empilement définitif d'un niveau d'interprétation à chaque exécution d'une lambda réflexive au niveau de l'interprète méta-circulaire se traduit par une dégradation d'un facteur 100 des performances à l'exécution. L'introduction de la réflexivité globale dans un interprète réflexif ne peut donc se faire sans prendre en compte ce problème de l'efficacité.

Dans le chapitre suivant, nous allons nous intéresser au problème de l'efficacité des interprètes réflexifs en appliquant l'évaluation partielle à un interprète réflexif. Bien que limité à une réflexivité locale et à un sous-ensemble strictement fonctionnel de Scheme, cet interprète qui s'appuie toujours sur l'interprète $I_r$de S. Jefferson [Jefferson 92], offre une solution au problème de l'efficacité dans les interprètes réflexifs.

Chapitre 5

Un interprète réflexif basé sur l'évaluation partielle

Le modèle de tour virtuellement infinie d'interprètes présente un caractère récursif qui, si son implémentation respecte toutes les contraintes imposées par une connexion causale complète, induit une inefficacité indésirable de l'exécution. Ce problème a été soulevé dès la naissance de la réflexivité par B.C. Smith [Smith 83] et par J. des Rivières [desRivières 84]. L'implémentation de 3-Lisp qu'ils proposent reste un modèle en matière d'implémentation efficace de la réflexivité. La connexion causale n'est cependant pas complète au niveau global et si les niveaux d'interprétation au sein de la tour d'interprètes ne sont empilés que lorsque nécessaire, ils s'accumulent néanmoins lors de l'interprétation de lambdas réflexives. L'efficacité de l'exécution n'est par conséquent pas aussi grande que pour un interprète classique. La plupart des autres interprètes basés sur un modèle interprétatif (Brown [Wand 88], Blond [Danvy 88], ...) souffrent du même problème.

Les chercheurs travaillant sur l'évaluation partielle ([Consel 93], [Mogensen 96], [Ruf 93b], [Weise 90], ...) se sont fait les premiers avocats de l'utilisation de leur technique [Ruf 93a] afin de résoudre la question de l'inefficacité inhérente au modèle de la tour d'interprètes. Malgré tout, peu de travaux allant dans ce sens ont été effectués. L'évaluation partielle semble pourtant être un excellent candidat à l'élimination du surcoût de l'interprétation. En plus de sa capacité à éliminer les niveaux d'interprétation, elle peut potentiellement diminuer le coût de l'utilisation de représentations de haut niveau en les intégrant directement dans le code final. Sachant que les principales raisons de l'inefficacité des interprètes réflexifs résident non seulement dans l'empilement de plusieurs niveaux d'interprétation, mais aussi dans l'utilisation d'une représentation de haut niveau des données manipulables par l'utilisateur, l'utilisation de l'évaluation partielle dans le cadre de la réflexivité devrait être bénéfique.

Ce chapitre vise à mieux cerner les avantages et les inconvénients de l'utilisation de l'évaluation partielle dans un interprète réflexif. Il s'agit montrer comment l'évaluation partielle peut être appliquée à un interprète réflexif et quel est son apport dans le traitement de l'efficacité de l'exécution. Après une brève présentation de l'évaluation partielle, un interprète réflexif basé sur cette technique est décrit. Cet interprète, bien que limité, en démontre l'intérêt.

5.1 Introduction à l'évaluation partielle

5.1.1 Introduction

L'évaluation partielle ([Consel 93], [Mogensen 96], [Ruf 93b], [Weise 90], ...) est une technique de transformation de programmes qui permet de spécialiser un programme en fonction d'une spécification partielle de ses paramètres d'entrée. Le résultat de l'évaluation partielle d'un programme p en fonction de la spécification partielle de certains de ses paramètres d'entrée est un autre programme $p_\prime$. p et $p_\prime$ sont liés par une équation dite équation de l'évaluation partielle :

Soit un évaluateur partiel PE acceptant en entrée un programme p écrit dans un langage L, une spécification partielle S des paramètres $(v_1 \ldots v_n)$ et générant en sortie une nouvelle version p_S de p, PE vérifie l'équation suivante:

$$PE(p, [S, \ldots]) = p_S \qquad \Longrightarrow \qquad p_S([S_1]) = p([S, S_1]) \qquad \forall \ \ S_1$$

Ce qui signifie que p_S peut être utilisé en lieu et place de p à chaque fois que les paramètres passés vérifient S et que pour ces paramètres p retourne une valeur.

L'évaluation partielle trouve ses fondements dans le théorème *s-m-n* de Kleene [Kleene 52]. La possibilité d'obtention d'une forme spécialisée d'un programme est démontrable de façon naïve. Ayant une fonction f de trois arguments, une forme spécialisée f_s de f pour laquelle le premier argument prendrait la valeur $\underline{v}$. f_s peut naïvement s'écrire (en pseudo Scheme):

```
(define (f_s a1 a2 a3) (f v a2 a3))
```

Naturellement, l'intérêt d'une telle réécriture reste encore à démontrer. Et, si l'évaluation partielle s'arrêtait là, elle ne serait pas l'objet de nombreuses activités. Ce qui rend l'évaluation partielle précieuse tient dans la manière dont le programme spécialisé est obtenu. Lors de l'évaluation partielle tout est mis en œuvre afin d'évaluer tous les éléments qui peuvent l'être en exploitant toutes les informations disponibles de façon à diminuer autant que possible le coût de l'exécution du programme spécialisé. Si une expression définit la somme de deux variables dont les valeurs sont connues dès l'évaluation partielle, alors cette expression sera remplacée dans le programme spécialisé par le résultat de la somme des valeurs des variables. Le programme spécialisé n'aura plus à effectuer cette somme mais utilisera directement le résultat.

L'aspect le plus délicat de l'évaluation partielle est le traitement des appels de fonctions. À chaque appel de fonction, l'évaluateur partiel doit faire un choix :

1. l'appel peut être remplacé par le corps de la fonction partiellement évalué,

2. ou il peut être remplacé par un appel à une version spécialisée de la fonction appelée.

Un certain antagonisme oppose ces deux possibilités. Choisir de dérouler la fonction, le premier choix, améliore l'efficacité de l'exécution du code en supprimant un appel à la fonction, mais augmente la taille du code. Le deuxième choix garantit la taille du code mais diminue l'efficacité de l'exécution du programme spécialisé. Ce dilemme doit aussi tenir compte du problème de la terminaison de l'évaluation partielle car même en restreignant le problème aux programmes qui terminent, un évaluateur partiel qui choisirait d'opter systématiquement pour (1) ne pourrait terminer dans la totalité des cas.

Ce choix est déterminé par un algorithme de terminaison. Lors de la spécialisation d'un programme, ce dernier choisit de dérouler ou de résidualiser un appel de fonction. Cet algorithme détermine non seulement la terminaison de l'évaluation partielle mais aussi la qualité du programme spécialisé. De son choix dépend l'intérêt même de l'utilisation de l'évaluation partielle. Un algorithme absolument sûr qui, comme le décrit E. Ruf [Ruf 93b], terminerait dès qu'il détecterait une boucle, produit un programme spécialisé lent parce qu'il manquera des opportunités d'optimisation. Il traduit donc le compromis entre la qualité du code spécialisé et la nécessité de terminer l'évaluation partielle et joue un rôle critique dans le fonctionnement d'un évaluateur partiel.

L'algorithme de terminaison diffère d'un évaluateur partiel à un autre. En particulier, une distinction, importante pour la suite de notre propos, doit être effectuée entre les évaluateurs partiels dits automatiques qui basent leur algorithme de terminaison sur la seule connaissance du programme à spécialiser et de la spécification des entrées, des évaluateurs qui utilisent un système d'annotations, introduites par le programmeur, du programme pour les guider dans leurs décisions. L'utilisation de ces derniers dans le cadre de la réflexivité est rendue impossible par la nécessité de ne pas requérir à l'intervention du programmeur dans le processus d'évaluation partielle d'un programme.

5.1.2 Exemple d'évaluation partielle:

Afin de donner une meilleure idée de l'évaluation partielle, prenons un exemple concret. Nous nous servons pour cela d'un évaluateur partiel automatique basé sur Fuse ([Ruf 93b], [Weise 90], [Weise 91]), et qui traite un sous-ensemble de la partie purement fonctionnelle du langage Scheme [Abelson 91].

La fonction *app*, ci dessous, permet la concaténation de deux listes. Elle est écrite de façon classique sous forme récursive.

```
(define (app l1 l2)
  (if (null? l1)
    l2
    (cons (car l1) (app (cdr l1) l2))))
```

L'évaluation partielle de *app* en spécifiant que *l1* prend la valeur '(1 2 3) et que *l2* est inconnu au moment de l'évaluation partielle donne le résultat[1] suivant:

1. Les résultats d'évaluation partielle présentés ici ne correspondent pas toujours mot à mot au résultat

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

app-1 donnera le même résultat que *app* pour *l1* prenant la valeur '(1 2 3) et cela quelle que soit la valeur de *l2*. C'est une illustration de l'équation de l'évaluation partielle. La sémantique est donc respectée. Par contre l'exécution de *app-1* est plus rapide que celle de *app*. Non seulement les appels récursifs ont été éliminés, mais aussi l'expression **if**.

L'évaluation partielle permet dans ce cas un gain significatif. Cependant, l'évaluation partielle de la même fonction *app* mais en donnant cette fois à *l2* la valeur '(1 2 3) et laissant *l1* inconnu donne un tout autre résultat:

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

La version spécialisée *app-2* de *app* ne permet pas de réel gain et la substitution à *l2* de sa valeur ne justifie pas l'utilisation d'un évaluateur partiel. Cet exemple montre que l'utilisation de l'évaluation partielle ne peut se faire au hasard sans risquer d'obtenir des résultats décevants.

Les résultats obtenus avec l'évaluation partielle deviennent impressionnants dès qu'elle est appliquée à des programmes basés sur des mécanismes interprétatifs. La connaissance préalable du programme à interpréter se traduit par un programme spécialisé qui le plus souvent élimine le niveau d'interprétation introduit par le programme.

L'exemple suivant illustre ce dernier point. La fonction *calc* implémente une petite calculatrice fonctionnant en notation post-fixée. Le premier argument de cette fonction est une liste dont le *car* est une liste de paramètres et le *cdr* la liste des opérations à effectuer. Le deuxième argument contient sous forme de liste les valeurs des paramètres du programme.

```
(define (ext env vars vals)
  (if (null? vars)
    env
    (cons (cons (car vars) (car vals))
          (ext env (cdr vars) (cdr vals)))))
```

```
(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)
```

produit par l'évaluateur partiel. Les techniques employées pour générer le code produit le rendent parfois illisible. C'est pourquoi les résultats des exemples d'évaluation partielle ont été réécrit afin de les rendre plus lisibles.

```
                                (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)
        (ext '() (car prgm) values)))
```

Le calcul de l'expression :

$$2a + 3b$$

s'effectue avec

$$a = 4 \qquad et \qquad b = 5$$

de la manière suivante:

```
> (calc '((a b) a 2 * b 3 * +) '(4 5))
23
>
```

Si maintenant on spécialise *calc* en spécifiant le premier paramètre comme étant le même que celui du calcul précèdent et le second paramètre comme étant inconnu, le programme spécialisé prend la forme suivante:

```
(define (calc-32 f values)
  (+ (* 2 (car values)) (* 3 (car (cdr values)))))
```

Le résultat est une fonction Scheme qui calcule directement l'expression $2a+3b$. Le niveau d'interprétation introduit par *calc* a été complètement supprimé. Non seulement toute la partie interprétative de *calc* a disparu, mais aussi la manipulation de l'environnement de calcul, la fonction résultante extrayant maintenant directement les valeurs utilisées de la liste des valeurs. Le gain est particulièrement significatif. Il faut cependant noter que l'évaluation partielle de *calc* avec le premier paramètre inconnu et le second connu n'aurait permis qu'une suppression des calculs relatifs à l'environnement, ce qui, bien que déjà intéressant, reste moins spectaculaire que le cas présenté. La suppression complète d'un niveau d'interprétation dépend donc des éléments connus au moment de l'évaluation partielle.

L'évaluateur partiel utilisé pour produire les résultats présentés est issu de Fuse [Ruf 93b]. L'algorithme de terminaison tel qu'il est décrit dans [Weise 91] ne permet pas d'obtenir des résultats tels que celui du deuxième exemple s'il est utilisé tel quel. Cet algorithme peut informellement se définir de la façon suivante: lors d'un appel de fonction récursive (que ce soit une récursivité directe ou indirecte), si l'appel courant était effectué de façon sûre lors de l'évaluation, cet appel est déroulé, et s'il est spéculatif, i.e. conditionné par un **if** dont la valeur du test n'a pu être déterminée, il est laissé résiduel. Les interprètes étant souvent récursifs, cet algorithme à tendance à terminer l'évaluation partielle trop tôt, en particulier lorsque le programme spécialisé est un interprète qui doit évaluer des expressions conditionnelles.

5.1.3 Évaluation partielle et compilation

Les exemples précédents permettent déjà de faire un parallèle entre évaluation partielle et compilation. Plus qu'un compilateur, un évaluateur partiel est un interprète. Il est capable du même travail qu'un interprète si l'ensemble des paramètres du programme à spécialiser est connu. La complexité d'un évaluateur partiel étant plus grande que celle d'un interprète classique et son efficacité moindre, son utilisation en tant qu'interprète est sans intérêt.

L'évaluation partielle est très souvent utilisée pour la compilation. Le second exemple de la section précédente démontre l'intérêt et l'efficacité de cette utilisation. Elle n'est pas nouvelle. Futamura [Futamura 71], qui le premier considère l'évaluateur partiel aussi bien comme un programme que comme une donnée possible de l'évaluation partielle [Mogensen 96], est à l'origine des trois «projections de Futamura» qui définissent l'utilisation de l'évaluation partielle pour la compilation, la génération de compilateurs et la génération de générateurs de compilateurs. Notre exemple est en fait une mise en application de la première de ces projections.

Ces projections ne sont possibles qu'en imposant une contrainte sur l'évaluateur partiel. Elles ne sont possibles, du moins pour les deux dernières, que si le langage traité par l'évaluateur partiel, celui utilisé pour le programme spécialisé généré ainsi que celui dans lequel l'évaluateur partiel est écrit, sont égaux.

En tant que compilateur, i.e. lorsqu'elle est associée à un interprète, l'évaluation partielle se montre particulièrement prometteuse dans le cadre de la réflexivité. Cette association présente une différence majeure par rapport à un compilateur classique: *elle est fonction de l'interprète*. Alors qu'un compilateur classique compile un langage donné avec une sémantique donnée, la modification de l'interprète est immédiatement prise en compte par l'évaluateur partiel utilisé en tant que compilateur. Et le résultat d'une évaluation partielle peut être lui-même associé à l'évaluateur partiel pour former un nouveau compilateur. Ce comportement du couple (évaluation partielle, interprète) en fait un candidat potentiel pour la compilation d'une tour d'interprètes réflexifs.

5.2 Réflexivité et évaluation partielle

Le modèle de la tour virtuellement infinie d'interprètes se traduit par des implémentations dont l'efficacité n'égale pas celle des interprètes classiques. La raison de cette inefficacité réside dans l'accumulation de plusieurs niveaux d'interprètes, que ce soit de façon globale comme dans l'implémentation présentée par Jefferson [Jefferson 92] et dans I_{r^∞}, ou localement comme dans celle de 3-Lisp [desRivières 84]. Une efficacité réelle, comparable à celle d'interprètes plus classiques, ne peut être atteinte qu'en supprimant le coût dû à l'accumulation de plusieurs niveaux d'interprétation.

L'évaluation partielle utilisée comme compilateur permet le plus souvent la suppression d'un niveau d'interprétation. Partant de cette constatation, l'application de l'évaluation partielle à une tour virtuellement infinie d'interprètes réflexifs laisse espérer sa réduction à un seul niveau d'interprétation, ou même sa suppression complète.

Étudions maintenant les conditions de l'application de cette technique à la réduction de la tour d'interprètes.

5.2.1 Principe de réduction de la tour d'interprètes

Une tour infinie d'interprètes réflexifs n'est que virtuellement infinie. C'est une condition nécessaire pour pouvoir rendre possible son implémentation [desRivières 84]. Il existe toujours[2] un niveau k d'interprétation à partir duquel tous les niveaux d'interprétation sont identiques.

Un évaluateur partiel appliqué à un interprète I permet la suppression du niveau d'interprétation dû à I. Pour que cette application soit possible, il faut que le langage qui sert à l'implémentation de l'interprète I soit égal à ou inclus dans celui traité par l'évaluateur partiel. Une tour d'interprètes peut être physiquement constituée par un interprète méta-circulaire I_{mc} en son sommet, interprète qui simule le caractère infini de la tour, suivi des interprètes I_k à I_1 créés dynamiquement lors de l'exécution des programmes de l'utilisateur. I_{mc} présente la particularité d'être constant quels que soient les programmes qu'il exécute. Simplement, dans le cas où il détecte un changement de niveau dans un programme utilisateur qui atteindrait son niveau courant d'exécution, il créerait un niveau supplémentaire d'interprétation qui viendrait s'exécuter à son niveau courant, lui-même s'exécutant désormais un niveau plus haut. Ce comportement qui interdit à un programme utilisateur de s'exécuter au niveau courant de I_{mc} en garantit le caractère constant. Si L est le langage qui sert à l'implémentation de l'interprète méta-circulaire, un évaluateur partiel PE_L pour ce langage peut être appliqué à I_{mc}. En prenant comme un des paramètres de cette évaluation partielle I_k, le résultat obtenu est un nouvel interprète $I_{mc,k}$:

$$PE_L(I_{mc}, (I_k, \ldots)) = I_{mc,k}$$

2. L'existence d'un tel niveau est soumise à une seule condition: que les programmes interprétés se terminent !

L'équation de l'évaluation partielle assurant l'équivalence suivante:

$$I_{mc}(I_k(d)) \qquad \Longleftrightarrow \qquad I_{mc,k}(d) \qquad \forall\ d$$

Les deux derniers niveaux de la tour d'interprètes peuvent être remplacés par $I_{mc,k}$.

$$I_{mc}(I_k(I_{k-1}(\ldots I_0(\ldots)\ldots))) \qquad \Longleftrightarrow \qquad I_{mc,k}(I_{k-1}(\ldots I_0(\ldots)\ldots))$$

Sous la condition que le produit de l'évaluation partielle utilise le même langage que celui traité par l'évaluateur partiel, le procédé peut être réitéré pour $I_{mc,k}$ et I_{k-1}.

$$PE_L(I_{mc,k},(I_{k-1},\ldots)) = I_{mc,k,k-1}$$

L'interprète ainsi obtenu peut lui remplacer les trois derniers niveaux de la tour:

$$I_{mc}(I_k(I_{k-1}(\ldots I_0(\ldots)\ldots))) \qquad \Longleftrightarrow \qquad I_{mc,k,k-1}(\ldots)$$

La réitération du procédé jusqu'à I_0 permet d'obtenir un interprète $I_{mc,k,k-1,\ldots,0}$ capable de remplacer l'intégralité de la tour :

$$I_{mc}(I_k(I_{k-1}(\ldots I_0(\ldots)\ldots))) \qquad \Longleftrightarrow \qquad I_{mc,k,k-1,\ldots,0}(\ldots)$$

Dans le cas optimal où l'évaluation partielle a éliminé l'intégralité des niveaux d'interprétation, l'interprète résultant simule une tour réduite à un seul niveau. L'efficacité de l'interprète ainsi obtenu doit être comparable à celle d'un interprète classique, ce qui correspond à notre objectif.

Cette présentation est idéaliste. Elle ne dit pas ce que sont les différents interprètes et quels sont les paramètres qu'ils traitent. L'évaluation partielle s'applique en général à un programme sous forme textuel, défini par un ensemble de fonctions dans lequel un point d'entrée est spécifié, et produit un autre ensemble de fonctions sous forme de texte. Or la solution que nous présentons ici se base sur un évaluateur partiel intégré à l'interprète réflexif. Cette intégration de l'évaluateur partiel avec l'interprète réflexif doit, elle aussi, être décrite.

Un autre aspect de l'évaluation partielle a été laissé de côté: lors de l'évaluation partielle, les données spécifiées sont rendues statiques. À chaque fois que cela est possible, elles seront intégrées dans le résultat de l'évaluation partielle, soit directement en apparaissant dans le «texte» final ou indirectement lorsqu'elles auront servi à conditionner le flot de contrôle du résultat final.

La véritable problématique de l'utilisation de l'évaluation partielle dans le cadre de la réflexivité réside dans l'éclaircissement de ces derniers points. Le procédé mis en avant dans cette section constitue plus une suggestion de la solution qu'une réelle solution. Dans la suite, nous allons donc essayer d'éclaircir ces questions.

5.2.2 Comment évaluer partiellement une tour interprète

Les exemples de mise en œuvre de l'évaluation partielle qui illustrent la première des projections de Futamura [Futamura 71] font appel à des interprètes dont tous les aspects sont connus. La réflexivité en dotant le langage de capacités à modifier ses mécanismes permet à l'utilisateur de changer complètement l'aspect d'un interprète. Aucune hypothèse ne devrait être faite sur les interprètes modifiés par l'utilisateur. La seule constante connue et sûre au sein d'une tour d'interprètes est l'interprète méta-circulaire utilisé pour l'implémenter. Entre le niveau auquel s'exécute cet interprète et le niveau utilisateur, les modifications effectuées peuvent être telles que la fonction *eval* habituellement utilisée pour l'évaluation d'une expression prenne une toute autre sémantique.

Le niveau auquel s'exécute l'interprète méta-circulaire n'est pas non plus une constante. Selon les implémentations, il va augmenter jusqu'à atteindre un niveau pas encore atteint par les programmes utilisateurs (cas de l'interprète modifié de Jefferson), ou encore il va constamment monter et descendre dans la tour selon le code à exécuter (cas de 3-Lisp [desRivières 84]). Et pour complexifier encore, l'ajout d'un niveau d'interprétation au sein de la tour s'obtient en insérant un nouvel interprète entre l'interprète méta-circulaire et l'interprète qu'il exécutait.

Le caractère dynamique de la tour, la modification possible à tout moment du nombre d'interprètes contenus dans celle-ci, rendent difficilement envisageable de la réduire en une seule fois. Une solution raisonnable est de la réduire au fur et à mesure de sa construction par applications successives de l'évaluation partielle.

Les modifications de l'interprétation que permet la réflexivité sont de deux types. Le premier type concerne les lambdas réflexives. Ces dernières permettent d'insérer dynamiquement du code dans l'interprète en modifiant l'exécution des expressions qui y font appel. La définition de l'interprète n'est pas modifiée mais uniquement le cours de l'interprétation. Le deuxième type de modifications concerne la modification à proprement parler de l'interprète que ce soit par effet de bord lors de l'exécution d'une lambda réflexive, ou par utilisation de constructions spécifiques comme dans Refci [Simmons II 93]. Si les conséquences des modifications effectuées sont dans le premier cas limitées aux expressions qui font appel à ces lambdas réflexives, elles peuvent dans le deuxième cas influer sur l'ensemble des niveaux d'interprétation allant du niveau modifié au niveau utilisateur et nécessiter alors la recompilation de toutes les fonctions qui appartiennent à ces différents niveaux.

Compilation de l'interprète

La réduction d'une tour d'interprètes suppose la capacité de l'évaluation partielle à compiler un interprète. Les interprètes réflexifs de langages fonctionnels implémentent les interprètes au travers de l'environnement global. Un interprète est alors constitué d'un ensemble de fermetures dont les valeurs sont stockées dans l'environnement global. Pour compiler un interprète, il faut donc compiler l'ensemble des fonctions qui le constitue.

Identification de l'interprète d'une fermeture

La compilation des fermetures lexicales, définies par l'utilisateur ou qui composent un interprète, se base sur l'évaluation partielle d'un interprète spécialisé en fonction de la fermeture à compiler. L'application de l'évaluation partielle nécessite donc la connaissance de l'interprète de la fermeture à compiler. Or l'interprète d'une fermeture n'est pas connu lors de sa définition mais seulement lors de son exécution.

La quasi totalité des interprètes réflexifs de langages fonctionnels acceptent le passage d'une fermeture lexicale comme valeur d'un des paramètres d'une autre fermeture, en particulier, d'une fermeture réflexive. Une telle fermeture réflexive a la capacité d'exécuter une fermeture provenant du niveau utilisateur à son propre niveau d'exécution, i.e. au sein de l'interprète. L'interprète de la fermeture définie au niveau utilisateur n'est alors plus l'interprète des fermetures du niveau utilisateur mais l'interprète de cet interprète. Une fermeture réflexive permettant d'exécuter une fermeture simple du niveau utilisateur peut être définie, en respectant la syntaxe de I_r[Jefferson 92], de la façon suivante:

```
(set! call-at-interp-level
  (compound-to-reifier
    (lambda (e r k)
      (evaluate (car e) r
        (lambda (closure)
          (evaluate (cadr e) r
            (lambda (first-arg)
              (evaluate (caddr e) r
                (lambda (second-arg)
                  (k (closure first-arg second-arg))))))))))))
```

En appelant la fermeture de deux paramètres dont la valeur lui est passée, la lambda *call-at-interp-level* en modifie l'interprète. La possibilité qu'a une fermeture réflexive de modifier dynamiquement le niveau d'évaluation d'une autre fermeture rend l'interprète d'une fermeture donnée indéterminé jusqu'à ce qu'elle soit exécutée.

Pour pouvoir résoudre ce problème de l'indéterminisme de l'interprète à la création d'une fermeture plusieurs solutions sont envisageables:

- Une première solution consiste à compiler la fermeture à chaque changement d'interprète. Un nouveau champ de la structure décrivant les fermetures est alors utilisé pour conserver l'ensemble des couples (interprète, forme compilée associée) afin d'éviter une recompilation de la fermeture à chaque appel.

- Une autre solution consiste à attribuer statiquement un interprète à chaque fermeture. Chaque fermeture étant interprétée par son propre interprète, l'appel d'une fermeture depuis une autre fermeture donne lieu à un changement d'interprète, l'évaluation des valeurs des paramètres de l'appel ayant lieu dans le contexte de l'appelant. Un interprète est défini par l'ensemble des fermetures qui le composent, ensemble qui prend la forme d'un environnement. Ces fermetures de l'interprète ont elles-même un interprète. La tour d'interprètes n'est alors plus vue de manière monolithique comme dans les interprètes réflexifs développés jusqu'à maintenant. Elle prend un aspect plus diffus pourtant bien adapté à l'utilisation de l'évaluation partielle. Une telle architecture pose cependant un problème de circularité qui peut être résolu par la définition d'un interprète méta-circulaire initial dont les fermetures qui le composent ont pour interprète ce même interprète. L'implémentation d'un tel interprète, bien que non triviale, est possible à l'aide des techniques traditionnelles d'implémentation d'interprètes méta-circulaires.

La première solution est plus lourde qu'il n'y parait: elle nécessite, afin de garantir le lien causal, une recompilation de la fermeture à chaque fois que l'interprète est modifié. Le maintien d'une liste des couples (interprète, forme compilée) grève l'occupation mémoire et les multiples compilations et recompilations nécessaires diminuent l'efficacité d'exécution.

Une solution intermédiaire peut être adoptée en choisissant de ne pas recompiler les fermetures lors d'une modification de l'interprète. Si cette approche améliore sensiblement l'efficacité du système, elle ne fait pourtant pas perdre à la première des solutions la totalité de sa lourdeur.

La deuxième solution est, elle, plus légère que la première parce qu'elle permet d'adopter une définition statique d'un interprète et cela de façon plus cohérente avec la gestion du lien causal. Sa mise en place nécessite de revoir la gestion des interprètes par les environnements. Si habituellement l'ensemble des fermetures qui composent l'interprète se trouve dans l'environnement global, la gestion de multiples interprètes qui doivent être appelés les uns depuis les autres passe par l'adoption d'une structure qui les regroupe spécifiquement et les gère indépendamment de l'environnement global. L'adjonction d'une telle structure, bien qu'elle complexifie l'interprète, devrait permettre l'implémentation de l'architecture décrite dans le second choix.

Un point d'entrée dans l'interprète

Un interprète pour un langage de type Scheme s'organise autour de deux fonctions principales: *eval* et *apply*, la première ayant pour fonction l'évaluation des expressions et la seconde, l'application des fonctions. La plupart des interprètes réflexifs de langages fonctionnels s'accordent pour modéliser l'état de l'interprétation par les trois variables que sont l'expression en cours d'évaluation, la continuation du calcul et l'environnement courant.

Pour pouvoir appliquer l'évaluation partielle à un interprète, une fonction de l'interprète doit servir de point d'entrée. La fonction qui vient naturellement à l'esprit est *eval*. Cependant la tour n'est pas construite directement par *eval* mais par *apply* qui insère un nouveau niveau d'interprétation lors de l'application d'une lambda réflexive par l'interprète méta-circulaire.

Pour pouvoir construire notre tour de façon incrémentale, niveau par niveau, l'évaluation partielle sera appliquée à l'équivalent de *apply* dans notre interprète[3].

En optant pour *apply*, le modèle d'exécution adopté se rapproche de celui d'un interprète classique qui accepte des fonctions compilées. La possibilité de mélanger fonctions interprétées et fonctions compilées, en permettant aux unes d'appeler les autres et inversement, pose des problèmes identiques à ceux que nous allons rencontrer dans notre interprète réflexif basé sur l'évaluation partielle. La gestion des appels aux fonctions compilées est déléguée à une nouvelle primitive de l'interprète, la primitive *call*. La fonction *apply* prend en compte la possibilité d'appeler une fonction compilée en examinant un nouveau champ dans la structure qui décrit les fermetures lexicales dont la valeur est, lorsque la fonction a été compilée, le résultat de la compilation. L'application par notre interprète d'une fonction f, dont la forme compilée est $f_{compilée}$ prend alors la forme suivante:

| $(apply\ f\ args\ k) \iff (k\ (call\ f_{compilée}\ args))$

Cependant la nature réflexive de l'interprète ne permet pas d'établir une équivalence entre le nom d'une lambda et sa sémantique. Afin de rendre l'application de l'évaluation partielle intéressante, la fonction d'application d'une fonction doit être connue. Si cette fonction n'est pas connue au moment de l'évaluation partielle, l'évaluation partielle ne pourra réduire cette application et laissera l'expression correspondante résiduelle i.e. à évaluer dynamiquement. Pour garantir aussi souvent que possible la connaissance de la fonction d'application d'une fonction, sa spécification est intégrée dans la définition des lambdas. Plus précisément, la fonction d'application usuelle est découpée en une fonction d'application qui évalue les arguments à passer à la fonction appelée avant d'appeler une fonction nommée «applicateur» qui se charge elle de l'application à proprement parler de la fonction en question. On a alors:

$$(apply\ f\ args\ k) \text{ se traduit par } ((applier\ f)\ args\ k)$$

avec

| $((applier\ f)\ f\ args\ k) \iff (k\ (call\ (compiled\text{-}procedure\ f)\ args))$

Cette modification permet l'application de l'évaluation partielle dès la définition d'une fonction:

| $f_{compilée} = PE\ (\ applier, [f,\ args,\ \ldots])$

Elle définit par conséquent quand l'évaluation partielle doit être utilisée: à chaque définition de fonction. Cependant, étant donné que l'évaluation partielle d'une fonction a un coût non négligeable, son application ne peut être liée à toute création de fonction. Il n'est pas intéressant de compiler une fonction qui ne sera appliquée qu'une seule fois. Aussi, l'application de l'évaluation partielle est liée non à la création d'une fonction, mais à la déclaration

3. Dans un interprète réflexif basé non sur les lambdas réflexives mais sur la construction *exec-at-meta* comme dans Rscheme [Ichisugi 92], la réduction de la tour aurait pu être basée sur la fonction d'évaluation.

d'une nouvelle fonction dans l'environnement global. Une telle fonction est susceptible d'être appelée un nombre suffisant de fois pour justifier l'utilisation de l'évaluation partielle.

Évaluation partielle appliquée à la réflexivité locale

Les lambdas réflexives influent dynamiquement sur le cours de l'interprétation. Aucun des interprètes d'une tour virtuelle n'est physiquement modifié par l'exécution d'une telle lambda. Les modifications interviennent uniquement dans l'interprétation des expressions qui font appel aux lambdas réflexives.

L'interprète d'une lambda réflexive n'est pas connu à priori, le niveau d'interprétation auquel elle sera exécutée n'étant pas défini par cette lambda. En particulier, elle peut être appelée depuis une autre lambda réflexive et modifier l'interprétation de cette lambda. L'interprète de cette lambda réflexive n'est plus l'interprète de l'interprète de l'expression initiale, mais l'interprète qui se trouve trois niveaux au dessus:

$(n+3)$	*Interprète de* $f_{reflect2}$
$(n+2)$	(((de-reflect $f_{reflect2}$) ...)
$(n+1)$	((de-reflect $f_{reflect1}$) ...) $\longrightarrow$ (... ($f_{reflect2}$ ...) ...)
(n)	(... ($f_{reflect1}$ ...) ...)

La compilation d'une lambda réflexive ne peut donc se faire lors de sa création mais lors de son utilisation, i.e. lors de la compilation d'expressions l'utilisant.

La compilation des lambdas réflexives peut être optimisée de deux façons. Une première façon étant de conserver l'ensemble des formes compilées correspondant aux différents niveaux d'interprétation auxquels elle est appelée. Une deuxième façon consiste à attribuer à chaque lambda un interprète déterminé statiquement. Cette deuxième solution, si elle fait perdre une partie du caractère dynamique du langage, permet d'effectuer une seule compilation par lambda réflexive.

Évaluation partielle appliquée à la réflexivité globale

La modification d'une des définitions d'un des interprètes de la tour virtuellement infinie d'interprètes se répercute causalement sur l'interprétation de l'ensemble des fonctions des interprètes qui se trouvent en dessous. La forme compilée d'une fermeture lexicale devant refléter son interprétation, la recompilation de l'ensemble des fonctions affectées par la modification est nécessaire.

Pour que la compilation des fonctions soit possible il est nécessaire de connaître le contexte dans lequel cette compilation a lieu. Pour une tour d'interprètes ce contexte dépend de l'ensemble des niveaux d'interprétation qui se situent au dessus de la fonction compilée. Si les modifications globales des interprètes sont dynamiques, la recompilation devra se faire suite à chacune des modifications. Le coût d'un tel mécanisme serait trop important et nécessiterait de recompiler plusieurs fois un même ensemble de fonctions.

En n'autorisant que des modifications statiques des définitions, la tour d'interprètes peut être compilée en une seule fois, en réduisant les niveaux d'interprétation les uns après les autres. La compilation d'une telle tour est déjà lourde. Une modification de la définition d'une fonction de l'interprète de niveau 4 entraîne en effet d'une part la compilation de la nouvelle fonction, mais aussi la recompilation de l'ensemble des fonctions de l'interprète de niveau 3, ce qui entraîne la recompilation de l'ensemble des fonctions de niveau 2, ... ainsi de suite jusqu'aux fonctions du niveau utilisateur, le niveau 0.

La compilation des fonctions de l'interprète ne pose pas de problèmes spécifiques. L'unique difficulté propre à la réflexivité globale est, si l'on veut continuer à bénéficier des performances dues à la compilation, la gestion rationnelle des modifications des définitions des interprètes pour assurer un nombre minimal de recompilations.

5.3 Un interprète réflexif basé sur l'évaluation partielle

L'interprète qui va être maintenant décrit a été construit dans un objectif précis: montrer ce que l'évaluation partielle peut apporter à la réflexivité. Afin d'obtenir de meilleurs résultats lors de l'évaluation partielle, mais aussi pour des problèmes de temps, le langage traité est un sous-ensemble purement fonctionnel de Scheme. En particulier l'affectation est traitée comme une commande de la boucle *read-eval-print* et non comme une instruction du langage. Ces limitations facilitent l'utilisation de l'évaluation partielle et diminuent les risques d'expression résiduelles, i.e. qui n'ayant pu être évaluées lors de l'évaluation partielle restent à évaluer lors de l'exécution.

Une autre limitation de notre interprète est qu'il est limité à une réflexivité locale, les interprètes de la tour étant identiques. La modification des fonctions de l'interprète par l'utilisateur est interdite. Cette limitation provient d'une part de la limitation sur le langage traité et d'autre part, elle simplifie grandement l'implémentation: les définitions des fonctions des interprètes ne pouvant être modifiées, l'interprète est connu quel que soit le niveau d'interprétation. Il en résulte que, quel que soit le niveau d'interprétation, l'interprète qui s'exécute à ce niveau est parfaitement déterminé et identique à ceux des autres niveaux. La compilation d'une fermeture peut donc se faire dès sa définition.

La limitation à la réflexivité locale de notre implémentation et sa restriction à un sous-ensemble purement fonctionnel ne nous a pas permis de réutiliser I_{r^∞}. Nous nous appuyons cependant sur le même interprète, l'interprète de Jefferson [Jefferson 92], que nous avons modifié en l'adaptant à nos contraintes, en lui supprimant certaines constructions linguistiques, en en ajoutant d'autres, et en l'étendant vers une tour infinie.

Cette section décrit cet interprète en faisant abstraction de l'évaluation partielle, mais en tenant compte de la possibilité pour une fonction d'être compilée. La compilation des fonctions sera abordée dans la section suivante.

5.3.1 Modifications de l'interprète

Dans cette section les définitions données sont méta-circulaires. Elles respectent par conséquent en tout point le langage interprété. En particulier, certaines constructions sont dictées par les limitations du langage traité par l'interprète.

Construction des compounds

L'utilisation de l'évaluation partielle va permettre la compilation des compounds. Le langage interprété est étendu vers un traitement intégré des compounds compilés et de l'applicateur associé. La modification des compounds, ou fermetures lexicales, intervient à deux niveaux:

1. leur définition est étendue en contraignant l'utilisateur à spécifier l'applicateur et,
2. la structure qui les décrit accepte deux nouveaux champs, l'un contenant l'applicateur de la fonction, le second sa forme compilée.

La définition d'une fermeture lexicale prend désormais la forme suivante[4]:

```
(lambda <applicateur> <paramètres> <instruction>)
```

Le constructeur des compounds s'écrit en suivant cette syntaxe:

```
(set! make-compound
  (lambda default-applier (args body env applier proc)
    (list 'compound args body env applier proc)))
```

Le *default-applier* est l'applicateur prédéfini. La seconde expression de la construction **lambda**, qui spécifie son applicateur, est évaluée dans le contexte lexical courant. La fonction *evaluate-abstraction* appelée par l'évaluateur se charge de cette évaluation ainsi que de la construction de la fermeture lexicale dont le résultat est passé à la continuation. Elle s'écrit:

```
(set! evaluate-abstraction
  (lambda default-applier (e r k)
    (evaluate (second e)      ; Évaluation de l'applicateur
              r
              (lambda default-applier (applier)
                (k (make-compound (third e) (fourth e) r
                                  applier #f))))))
```

evaluate-abstraction force l'évaluation de l'applicateur du compound créé dans le contexte de création. Ce processus permet de déterminer l'ensemble des paramètres nécessaires à la compilation dès la création des compounds.

4. La limitation à une seule instruction par fonction provient de notre évaluateur partiel.

Application des compounds

L'application des fermetures lexicales constitue la principale modification de l'interprète. La présence de l'applicateur d'une fermeture ainsi que la possibilité qu'elle a d'être compilée doivent être pris en compte. Une fermeture interprétée va être passée à son applicateur qui se chargera de son application effective. Lorsqu'une fermeture lexicale est compilée, elle l'est en fonction de son applicateur, et avec une continuation par défaut qui retourne immédiatement le résultat de l'appel de la fermeture lexicale. De ce comportement similaire à celui des primitives du langage est issu un traitement similaire à celui des primitives.

```
(set! apply-procedure
  (lambda default-applier (proc args k)
    (if (compound? proc)
        ; Application d'un compound
        ((lambda default-applier (compound-procedure)
           (if (procedure? compound-procedure)
               ; Forme compilée
               (k (call compound-procedure  args))
                 ; Forme interprétée
               ((compound-applier proc) proc args k)))
         (compound-proc proc))
         ; Appel d'une primitive
        (k (apply-primitive (primitive-name proc) args)))))
```

L'applicateur défini par défaut, *default-applier*, s'occupe de l'application effective d'une fonction. Il prend comme paramètres la fonction à appliquer, les valeurs des paramètres passés à cette fonction, ainsi que la continuation du calcul. Le corps des fermetures étant dans notre langage constitué d'une seule expression, *default-applier* appelle directement *evaluate* en lui passant comme argument l'expression à évaluer ainsi que le contexte dans lequel elle doit l'être.

```
(set! default-applier
  (lambda default-applier (proc args k)
    (evaluate (compound-body proc)
              (extend (compound-environment proc)
                      (compound-parameters proc)
                      args)
              k)))
```

La définition donnée ici, comme toute les autres définitions de cette section, est une définition méta-circulaire: *default-applier* s'utilise lui même dans sa propre définition. Nous verrons plus loin, lors de la description de l'interprète méta-circulaire, comment est brisée la méta-circularité de sa définition.

Le *evaluate*

La définition de *evaluate* est très peu modifiée. Comme toutes les fonctions de l'interprète de Jefferson [Jefferson 92], elle est réécrite en suivant la syntaxe de notre interprète. De plus, le traitement de l'affectation est supprimé.

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

La boucle *read-eval-print*

La boucle *read-eval-print* est modifiée, d'une part pour satisfaire aux contraintes imposées par notre langage, et d'autre part pour traiter l'affectation comme une commande. Les entrées de l'utilisateur sont analysées pour déterminer si c'est une affectation ou une simple expression du langage. Les affectations sont globales et traitées par la primitive *update-global-env*. Cette approche fige la structure de l'environnement global, ce que nécessite l'application de l'évaluation partielle.

```
(set! openloop
  (lambda default-applier (read-prompt write-prompt)
    ((lambda default-applier (ignored input)
        ((lambda default-applier (set?)
           (evaluate (if set? (third input) input)
                     '()
                     (lambda default-applier (v)
                            ; Les paramètres de cette fonction ne servent
                            ; qu'a évaluer les expressions qui suivent
                         ((lambda default-applier (i1 i2 i3 i4 i5)
                            (openloop read-prompt write-prompt))
                          (if set?
                            (update-global-env (second input) v)
                            #f)
                          (format #t "~a" (list write-prompt))
                          (if (eq? v (void))
                             "Display nothing"
                             (format #t " ~a" (list v)))
                          (format #t "~\%" '()))))))
          (if (pair? input) (eq? (first input) 'set!) #f)))
      (format #t "~a" (list write-prompt))
      (read))))
```

La primitive *write* utilisée par Jefferson [Jefferson 92] a été remplacée par la primitive *format* dont la syntaxe est légèrement différente de celle de Common Lisp [Steele jr 90]. Elle se base sur la syntaxe de STk [Gallesio 96], implémentation de Scheme que nous avons utilisée, avec comme seule différence le passage des valeurs à afficher. Cette syntaxe permet d'avoir un nombre fixe de paramètres, ce qu'exige notre interprète.

L'utilisation de la primitive *update-global-env* ne garantit pas à priori que l'utilisateur ne fasse pas d'affectation au sein de son code. Seul le protocole le permet. Pour garantir que cette primitive ne soit pas utilisée, il faudrait la sortir de l'interprète. Une implémentation plus propre peut être faite en sortant toute la boucle *read-eval-print* de l'interprète. *openloop* ne serait alors plus visible par l'utilisateur. Une telle implémentation garantirait que l'utilisateur ne puisse pas modifier la fonction *openloop* ainsi que la primitive *update-global-env* qui ne ferait plus alors partie du langage.

Les nouvelles primitives

Outre le remplacement de la primitive *write* par la primitive *format* plus souple, quelques primitives à effet de bord (*set-car!*, *set-cdr!*, ...) sont supprimées assurant le caractère «fonctionnel» de notre langage.

Pour permettre le traitement des fonctions compilées, les primitives *procedure?* et *call* sont ajoutées. La primitive *procedure?* provient du prédicat Scheme du même nom. La primitive *call* permet l'appel des formes compilées et est implémentée au moyen de la fonction *apply* de Scheme.

5.3.2 L'interprète méta-circulaire

La définition méta-circulaire donnée de l'interprète le rend impropre à être exécuté directement en Scheme comme cela est fait dans [Jefferson 92]. Pour pouvoir rendre notre interprète opérationnel il faut le doter d'une implémentation en Scheme. Cette implémentation doit aussi résoudre certains problèmes de méta-circularité comme la définition de *default-applier.*

Résolution de la circularité

L'écriture d'un interprète méta-circulaire fait appel à des techniques relativement classiques bien que non triviales. Les différents interprètes réflexifs présentés dans les chapitres précédents utilisent une variété assez complète de ces techniques. La technique utilisée ici se base sur deux caractéristiques de notre interprète. D'une part elle utilise la possibilité qu'ont les fermetures lexicales du langage d'avoir une forme compilée, l'interprète méta-circulaire étant implémenté à travers ces formes compilées. D'autre part elle se base sur l'interdiction de modifier les définitions de l'interprète.

L'exemple suivant montre l'implémentation de *evaluate-abstraction*. La fonction Scheme *def-global* ajoute une définition dans l'environnement global de notre interprète. L'identité de représentation des listes au sein de notre langage et en Scheme autorise l'utilisation des primitives de Scheme pour la construction des compounds. Le compound qui définit *evaluate-abstraction* est construit tel qu'il apparaît à l'utilisateur de l'interprète i.e. sous forme d'une liste taggée par le symbole *compound* suivi de la liste des paramètres, de l'expression constituant le corps de la fermeture, de l'environnement lexical, de l'applicateur de la fermeture puis de sa forme compilée.

```
(def-global evaluate-abstraction
  (list 'compound
        '(e r k)
        '(evaluate (second e) r
           (lambda default-applier (applier)
             (k (make-compound (third e) (fourth e) r applier #f))))
        '()
        default-applier
        (lambda (e r k)
          (call-compound 'evaluate (car (cdr e)) r
            (list 'compound
                  '(applier)
                  '(k (make-compound (third e) (fourth e) r applier #f))
                  (extend '() '(e r k) (list e r k))
                  default-applier
                  (lambda (applier)
                    (call-compound k
                      (call-compound 'make-compound
                        (car (cdr (cdr e)))
                        (car (cdr (cdr (cdr e))))
                        r applier #f))))))))
```

La forme compilée d'une fermeture est une fonction Scheme dont la structure n'est pas disponible à l'utilisateur de l'interprète qui peut seulement en identifier la nature à l'aide de la primitive *procedure?* et l'appeler à l'aide de la primitive *call*. Afin d'assurer la connaissance au sein de l'interprète des continuations et des environnements manipulés par l'interprète méta-circulaire, ces objets sont construits et manipulés sous la forme qu'ils adoptent dans notre langage. Ceci explique que la continuation passée à *evaluate* dans la forme compilée de *evaluate-abstraction* soit construite comme un compound tel qu'il est défini dans le langage implémenté plutôt que comme une fonction Scheme.

Les fermetures lexicales *extend* et *default-applier* sont définies à la fois dans l'environnement global du Scheme utilisé pour l'implémentation ainsi que dans l'environnement global de l'interprète. Cette double définition n'est nullement nécessaire et n'est là que pour faciliter l'écriture de l'interprète ainsi que pour améliorer sa lisibilité. Ces définitions ne pouvant être modifiées par l'utilisateur, la question de la cohérence des définitions entre les deux environnements ne se pose pas.

La forme compilée du compound présenté fait de nombreux appels à la fonction Scheme *call-compound*. Cette fonction joue un rôle primordial dans notre implémentation. Sa vocation est d'assurer l'interface entre les fonctions compilées et interprétées en permettant aux premières d'appeler ces dernières tout en s'assurant du bon niveau d'interprétation. Les lambdas réflexives peuvent modifier la continuation du calcul, introduisant de nouvelles continuations qui, si elles n'ont pu être compilées, doivent pouvoir être appelées en mode interprété.

Les changements de niveaux

Une des particularités de l'interprète méta-circulaire d'un langage réflexif est qu'il doit gérer les niveaux d'interprétation. Dans un interprète réflexif classique le changement de niveau d'interprétation s'effectue lorsqu'une lambda réflexive doit être interprétée au niveau où s'exécute l'interprète méta-circulaire.

Lorsque l'interprète doit exécuter une lambda réflexive à son niveau d'exécution, il appelle la fonction Scheme *call-reifier*. Cette fonction examine si la lambda réflexive est compilée ou non. Dans le cas d'une lambda réflexive compilée l'exécution peut se poursuivre au même niveau, l'interprète Scheme étant capable d'interpréter directement la forme compilée. Une lambda réflexive non compilée suit le processus habituel de création d'un niveau d'interprétation supplémentaire avant son interprétation par le nouvel interprète créé.

```
(define (call-reifier reifier e r k)
  (let ((compiled? (list-ref reifier 5)))
    (if compiled?
      (call-compound compiled? (cdr e) r k)
      (call-compound 'evaluate ; L'artillerie de l'interprétation
        '((reifier-to-compound proc) (rest e) r k)
        (extend '() '(proc e r k) (list proc e r k))
        (make-openloop-cc ...)))))
```

Le langage implémenté présente une deuxième cause de changement de niveau: les applicateurs. L'applicateur d'une fermeture lexicale doit être exécuté au même niveau que l'interprète. Là encore, les fermetures compilées doivent être différenciées de celles qui ne le sont pas. L'appel des fermetures compilées est traité explicitement par le langage. Les fermetures interprétées suivent un traitement similaire à celui des lambdas réflexives:

```
((compound-applier proc) proc args k)
```

L'expression (*compound-applier proc*) traduit l'appel au niveau d'exécution courant de l'interprète de l'applicateur de la fermeture à évaluer. Cet appel pose des problèmes similaires à l'appel des lambdas réflexives: la nécessité de pouvoir changer de niveau. La fonction Scheme *call-compound* assure aussi la gestion de cet appel, le problème se posant de façon identique pour l'appel des fermetures interprétées depuis les fermetures compilées. L'appel à l'applicateur s'écrit donc de la manière suivante:

```
(call-compound (list-ref proc 5) proc args k)
```

où *call-compound* s'écrit:

```
(define (call-compound compound . args)
  (if (procedure? compound)
    (apply compound args)
    (let ((compound (if (symbol? compound)
                        (cdr (assq compound global-env))
                        compound)))
      (let ((scheme-proc (list-ref compound 5)))
```

```
(if (procedure? scheme-proc)
  (apply scheme-proc args)
  (call-compound (list-ref compound 4)
                 compound args
                 (list 'compound '(v) 'v '()
                       default-applier
                       (lambda (v) v))))
```

Si l'applicateur du compound est compilé, la forme compilée sera directement appelée. Dans le cas où il est interprété, un changement de niveau est opéré en appelant l'applicateur de l'applicateur par un appel récursif de *call-compound*. Tous les applicateurs finissant par converger vers *default-applier* qui bénéficie d'une forme compilée, la boucle introduite par la récursivité se terminera lors de l'appel à *default-applier*.

Le *default-applier*

La définition méta-circulaire de *default-applier* utilise *default-applier*. L'interprète méta-circulaire doit briser cette circularité. Pour cela *default-applier* est défini sans préciser quel est son applicateur. *default-applier* ayant pour applicateur *default-applier*, ce dernier est fixé en faisant pointer le champ devant contenir le pointeur sur l'applicateur sur *default-applier*.

```
(define default-applier
  (list 'compound
        '(proc args k)
        '(evaluate (compound-body proc)
                   (extend (compound-environment proc)
                           (compound-parameters  proc)
                           args)
                   k)
        '()
        '????  ; L'applicateur de default-applier
               ; est en cours de définition
        (lambda (proc args k)
          (call-compound 'evaluate (car (cdr (cdr proc)))
            (call-compound 'extend
                           (car (cdr (cdr (cdr proc))))
                           (car (cdr proc))
                           args)
            k))))
```

```
; Installation de la circularité
(set-car! (cddddr default-applier) default-applier)
```

```
; Définition de default-applier dans l'environnement global
(def-global default-applier default-applier)
```

Un autre problème pourrait provenir de l'implémentation circulaire de *default-applier*: une boucle infinie lors de son application. L'application d'une fermeture au sein de notre interprète faisant appel à son applicateur, l'applicateur de *default-applier* étant *default-applier*, l'interprète risque de boucler lors de l'application de cette fermeture. Cependant, *default-applier* ayant une forme compilée, c'est cette dernière qui sera exécutée. La forme compilée d'une fermeture ne faisant plus appel à son applicateur, l'application de *default-applier* ne se traduira pas par une boucle infinie.

5.4 Évaluation partielle de l'interprète

Un évaluateur partiel ne peut être directement utilisé avec notre interprète même après les modifications qui viennent d'être décrites. Son application à un interprète réflexif nécessite quelques aménagements que nous nous proposons de décrire dans cette section. En particulier l'évaluateur partiel doit pouvoir être appelé depuis l'interprète réflexif et accepter de travailler à partir des informations contenues dans l'environnement global. Il doit aussi travailler en symbiose avec l'interprète en lui fournissant des résultats directement exploitables et en traitant correctement les appels des compounds, qu'ils soient réflexifs ou non. Nous commençons par expliquer pourquoi notre choix s'est porté sur Fuse.

5.4.1 Un évaluateur partiel automatique

L'utilisation de l'évaluation partielle pour compiler un interprète réflexif impose un certain nombre de contraintes sur l'évaluateur partiel. À chaque création d'un nouveau compound, qu'il soit réflexif ou non, l'évaluateur partiel est appelé dynamiquement par l'interprète. De plus cet évaluateur partiel doit être capable de travailler à partir des informations disponibles dans l'interprète et le résultat d'une évaluation partielle doit correspondre aux données attendues par l'interprète. Enfin, pour notre propos, la qualité du programme spécialisé prime sur la rapidité de la spécialisation. L'ensemble de ces contraintes et de ces critères nous a guidé dans notre choix d'un évaluateur partiel.

Plusieurs évaluateurs partiels pour Scheme ou un sous-ensemble de Scheme sont disponibles. Les seuls qui soient relativement complets et performants sont Similix [Bondorf 93] et Schism [Consel 90]. Ces deux évaluateurs ont la particularité d'être «offline», c'est à dire qu'ils prennent leur décision d'évaluer ou non une expression à partir d'une analyse préalable du texte du programme. Les essais effectués avec ces évaluateurs partiels ont montré leurs limites. Similix a tendance à laisser trop d'expressions résiduelles lors de son utilisation avec notre interprète. Il semble que ses difficultés proviennent de l'écriture en continuation explicite de l'interprète. Schism nécessite d'utiliser son système d'annotation des programmes pour obtenir des résultats satisfaisants. Les annotations sont fournies par l'utilisateur qui les insère manuellement dans le programme. Cette démarche va à l'encontre des contraintes imposées par la multiplicité des niveaux d'interprétation: le résultat de la compilation d'une fermeture utilisée au niveau de l'interprète courant devrait lui aussi être annoté. Il semble difficile de générer automatiquement ces annotations qui sont pourtant requises pour que les résultats de l'évaluation partielle ne contiennent pas trop d'expressions résiduelles [5].

Ces difficultés rendent difficile l'utilisation de ces deux évaluateurs partiels dans le cadre d'un interprète réflexif. Aussi nous nous sommes tournés vers un autre évaluateur partiel qui répond mieux à nos critères, Fuse.

Fuse est le résultat de recherches sur l'évaluation partielle. Un des critères de ces recherches était la qualité des résultats obtenus par l'évaluation partielle plutôt que la rapidité d'obtention de ces résultats comme dans les évaluateurs «offline». Fuse appartient à la famille des évaluateurs partiels dits «online» qui prennent leur décision d'évaluer ou non une expression au cours de la phase d'évaluation. Le contexte dans lequel l'évaluation doit avoir lieu étant mieux connu dans cette phase, le nombre d'expressions réductibles (i.e. qui sont évaluables) y est plus important.

Cet évaluateur présente une caractéristique importante pour notre domaine d'utilisation: il est automatique, l'algorithme de terminaison se basant uniquement sur le programme fourni, sans qu'aucune annotation supplémentaire ne soit nécessaire, pour prendre ses décisions. Le résultat de l'évaluation partielle d'une fermeture reste utilisable par l'évaluateur partiel avec des chances d'obtenir des résultats de qualité identique aux fermetures prédéfinies dans l'implémentation.

Fuse n'étant pas disponible, nous avons dû l'implémenter en reprenant les informations fournies dans la thèse de Ruf ainsi que dans les articles y ayant trait. Cette démarche nous a permis de mieux maîtriser les mécanismes de l'évaluation partielle ainsi que les détails de l'implémentation. Ce dernier point a facilité l'interfacage de l'évaluateur partiel avec l'interprète réflexif. Cependant notre objectif n'étant pas de fournir un interprète traitant l'ensemble du langage Scheme, l'évaluateur partiel a été simplifié et ne traite qu'un sous-ensemble purement fonctionnel de Scheme.

L'algorithme de terminaison utilisé est celui de [Weise 91]. Cet algorithme qui se base sur l'inspection d'une pile des appels des fermetures pour détecter les appels récursifs termine trop tôt pour donner de bons résultats dans le cadre de son utilisation selon la première des projections de Futamura [Futamura 71]. Un interprète pour un langage de type Scheme étant naturellement organisé autour de la fonction d'évaluation qui s'appelle indirectement de façon récursive, l'évaluateur partiel a tendance à terminer sur ces appels récursifs en produisant une version spécialisée de *evaluate*. L'ensemble des expressions à évaluer n'est pas examiné par l'évaluateur partiel. Pour obtenir la tentative d'évaluation de l'ensemble des expressions à évaluer, l'algorithme de terminaison est étendu selon les prescriptions données dans [Weise 91]. Les expressions données en entrée sont marquées, permettant à l'évaluateur partiel de ne pas terminer lorsqu'elles sont présentes dans les paramètres d'un appel de fonction. Ces expressions étant en nombre fini, la terminaison de l'évaluation partielle n'est pas mise en cause par cette modification.

5. Les expressions résiduelles sont celles laissées non évaluées par l'évaluateur partiel qui ne dispose pas des informations nécessaires pour le faire. Elles ne peuvent l'être que dans leur contexte normal d'utilisation.

L'objet de cette section n'est cependant pas l'évaluation partielle mais comment elle peut être utilisée pour compiler un interprète réflexif. Le reste de cette section traite de cette question.

5.4.2 Première projection de Futamura appliquée aux compounds

La compilation d'un compound à l'aide d'un évaluateur partiel est obtenue par application de la première projection de Futumara [Futamura 71]. Une utilisation directe de cette projection à partir d'un évaluateur partiel non modifié produit des résultats qui ne sont pas directement exploitables par notre interprète. Dans cette section, nous identifions les problèmes posés par une utilisation directe de l'évaluation partielle et nous montrons comment nous avons adapté l'évaluateur partiel afin qu'il produise des résultats directement utilisables par notre interprète.

L'application directe de l'évaluation partielle à un applicateur dont l'un des paramètres est la fonction à compiler ne donne pas le résultat escompté. L'évaluateur partiel produit une version spécialisée de l'applicateur en fonction des paramètres spécifiés et non une version spécialisée de la fonction à compiler. Les paramètres traités par le produit de l'évaluation partielle sont ceux de l'applicateur et non ceux de la fonction. Nous allons montrer dans cette section comment l'évaluateur partiel a été modifié de manière à produire une version compilée qui traite directement les paramètres de la fonction à compiler.

En l'appliquant à un applicateur, l'évaluateur partiel fournit par conséquent une version spécialisée de l'applicateur. Soit *f* une fermeture qui calcule la somme de ses deux paramètres et *default-applier* son applicateur, le résultat de l'évaluation partielle est:

```
(set! f (lambda default-applier (a b) (+ a b))
```

```
; Compilation directe de f
(pe default-applier
    (list f *unknown* (make-continuation)) ...)
```

```
; Résultat
(define (default-applier-10 proc args k)
    (+ (car args) (car (cdr args))))
```

L'évaluation partielle est appliquée de telle façon que le résultat obtenu puisse être utilisé directement et en particulier, retourne directement le résultat de l'appel de la fermeture. Cette approche permet un appel direct des fermetures entre elles sans nécessiter d'évaluation. La valeur du deuxième paramètre de l'applicateur est spécifiée comme étant inconnue, les valeurs des paramètres passés à la fermeture *f* n'étant déterminées qu'au moment de l'appel.

La fonction Scheme *make-continuation* construit une continuation sous forme de fermeture qui retourne immédiatement le résultat passé.

```
(define (make-continuation)
  (list 'compound
```

```
        '(v)
        'v
        '()
        default-applier
        (lambda (v) v)))
```

Le résultat attendu de l'évaluation partielle est une fonction Scheme dont un seul des paramètres est resté inconnu, le second paramètre de *default-applier*.

La fonction Scheme *default-applier-10* produite par l'évaluateur partiel ne peut être appelée directement par notre interprète car les paramètres de cette fonction dépendent de *default-applier* et non de f. Le produit de l'évaluation partielle peut être rendu plus conforme au résultat escompté en modifiant légèrement le point d'entrée dans l'évaluateur partiel. Ce point d'entrée force désormais la production d'une lambda Scheme dont les paramètres sont définis à partir de ceux de la fermeture compilée plutôt que de ceux de son applicateur. Une forme simplifiée de cette modification peut s'écrire:

```
(define (comp compound)
  `(lambda ,(compound-parameters compound)
     (papply (compound-applier compound)
             (list compound
                   (make-parameters (compound-parameters))
                   (make-continuation))
             ...)
```

Le second paramètre de l'applicateur est ici construit de façon plus élaborée comme une liste dont les éléments sont au nombre des paramètres de la fermeture compilée, et dont la valeur des éléments est inconnue bien qu'étant désignés comme des variables dont le nom est fourni par la liste des paramètres. Ceci est rendu possible par l'utilisation des valeurs symboliques [Weise 90] qui sont des structures permettant la description complète des objets du langage traité par l'évaluateur partiel.

```
(define (make-parameters parms)
  (if (null? parms)
      '()
      (cons (make-symbolic-value *a-value* (car parms))
            (make-parameters (cdr parms)))))
```

Ainsi modifié, l'évaluateur partiel produit des résultats plus conformes à ceux attendus. L'application de *comp* à f donne:

```
(lambda (a b)
  (+ a b))
```

Le résultat produit ne permettant pas toujours la suppression des premiers et derniers paramètres de l'applicateur, le produit de l'évaluation partielle est évalué dans un environnement contenant la définition de ces paramètres. Les fonctions détaillées de l'évaluateur partiel sont plus complexes pour des raisons techniques qui ne font pas partie du sujet de ce chapitre. La présentation simplifiée de ces fonctions permet de se faire une idée des problèmes essentiels à notre propos sans pour autant les obscurcir par des considérations trop techniques.

5.4.3 Modifications de l'évaluateur partiel

Le portage de Fuse sur notre système s'est traduit par une intégration fine de l'évaluateur partiel avec l'interprète réflexif. L'évaluateur partiel a été adapté pour exploiter directement les informations contenues dans l'environnement global. Il a été pourvu d'une phase de «post-processing» destinée à forcer la compilation de tout compound qui échappe à la première phase de l'évaluation partielle. Pour finir, le traitement des appels de fonctions a été revu pour assurer un compromis entre spécialisation et occupation de la mémoire.

L'environnement global

L'évaluateur partiel prend en entrée le texte du programme à traiter. Ce texte n'est pas directement disponible dans notre interprète mais peut être extrait de l'environnement global de l'interprète. Toute modification de l'environnement global est exclue durant l'évaluation partielle, l'environnement global de l'interprète peut être utilisé par l'évaluateur partiel moyennant sa traduction dans la représentation utilisée par l'évaluateur partiel.

Une autre difficulté technique provient de différences de représentation entre les objets appartenant au programme partiellement évalué et les objets manipulés par ce programme. Lors de l'évaluation partielle d'un interprète qui manipule une expression, trois niveaux d'interprétation ayant chacun leur propre représentation coexistent. Le premier niveau est constitué par l'évaluateur partiel, le second par l'interprète qu'il manipule et le troisième par l'expression calculée par l'interprète. À chacun des deux derniers niveaux correspond un environnement global. Ces environnements sont dans notre interprète confondus en un seul. Les différents niveaux de représentation utilisés par l'évaluateur partiel exigent la séparation des environnements en deux. Cette séparation est rendue possible par la garantie que l'environnement global ne sera pas modifié durant l'évaluation partielle.

Post-processing des compounds

L'évaluation partielle produit parfois des compounds qui ne sont pas compilés. Ce comportement est dû à la création de compounds qui ne sont pas appelés depuis le code traité, mais qui le seront dans un autre contexte. La compilation de ces compounds est néanmoins souhaitable pour améliorer l'efficacité de l'exécution.

Les fonctions Scheme traitées par Fuse [Ruf 93b] peuvent aussi ne pas être spécialisées lors de la phase d'évaluation partielle à proprement parler. L'évaluateur partiel est constitué de plusieurs modules le premier s'occupant de l'évaluation partielle et le troisième de générer le code final. Le second module traite le cas des lambdas qui n'ont pas été spécialisées. Pour être correcte, la génération du code final nécessite la spécialisation de toutes les fonctions Scheme.

Cette seconde phase est utilisée pour forcer la compilation des compounds. Lorsqu'un compound non compilé est détecté, l'évaluation partielle du compound est forcée.

Factorisation du code des compounds

L'évaluation partielle est un processus assez coûteux, tant par le temps requis pour son exécution, que par la quantité de code qu'il produit. Une tendance naturelle d'un évaluateur partiel comme Fuse [Ruf 93a] est de dupliquer certaines portions de code, en particulier de substituer l'appel à une fermeture par une duplication du code de la fermeture. En laissant l'évaluateur partiel procéder inconsidérément à des duplications de code, l'interprète risque de saturer rapidement la mémoire disponible. Un compromis entre l'efficacité de l'exécution du code produit et sa taille est souhaitable.

Les compounds appelés par l'interprète doivent être distingués de ceux appartenant au niveau en cours de compilation. La duplication du code intervient principalement lors des appels aux compounds du niveau en cours de compilation, la tendance naturelle des compounds appelés par l'interprète étant d'être éliminés. Pour éviter la duplication de ces compounds, il est essentiel de remarquer que leur forme compilée constitue déjà une optimisation. Dès lors, plutôt que de dérouler leur code, l'évaluateur partiel peut utiliser cette connaissance pour générer un appel, sous la forme d'un *call-compound*, à cette forme compilée. Ce compromis permet d'obtenir une bonne efficacité au niveau de l'exécution du code tout en optimisant l'occupation de la mémoire.

Pour obtenir un tel comportement de l'évaluateur partiel, il faut être capable de distinguer les appels des compounds de l'interprète entre eux, des appels des compounds du niveau compilé. Si les premiers sont le résultat d'un *call-compound*, les derniers sont systématiquement traités par la fonction *apply-procedure* de notre interprète. À ce niveau de l'interprète, la distinction entre les différents types d'appels, en particulier entre les compounds classiques et les compounds réflexifs, a déjà eu lieu. Ces derniers doivent impérativement être considérés par l'évaluateur partiel comme partie intégrante de l'interprète. Dans le cas contraire, la compilation ne serait pas aussi performante et perdrait même une partie de son sens en ne supprimant pas une partie significative de l'interprétation. Pour permettre la réutilisation de la forme compilée des compounds du niveau utilisateur, la forme compilée d'*apply-procedure* est modifiée:

```
(set! apply-procedure
  (list 'compound
        '(proc args k)
        '(if (compound? proc)
           ((lambda default-applier (compound-proc)
              (if (procedure? compound-proc)
                  (k (call compound-proc args))
                  ((compound-applier proc) proc args k)))
            (compound-procedure compound))
           (k (apply-primitive (primitive-name proc))))
```

```
'()
default-applier
(lambda (proc args k)
  (if (call-compound 'compound? proc)
    (let ((compound-procedure (list-ref proc 5)))
      (if (procedure? compound-procedure)
        (call-compound k (apply calling-compound
                                compound-procedure args))
        (call-compound (list-ref procedure 4) proc args k)))
    (call-compound k (call-compound 'apply-primitive (car (cdr proc)) args))))))
```

où *calling-compound* est défini comme un synonyme de *call-compound*. La différence entre ces deux fonctions Scheme réside dans leur traitement par l'évaluation partielle. La première sert à conserver les appels entre les fonctions d'un même niveau, tandis que la seconde va dérouler à chaque fois que c'est possible les appels des fonctions de l'interprète.

5.4.4 Traitement des primitives *call-compound*, *calling-compound* et *call-reifier*

Les fonctions Scheme *call-compound*, *calling-compound* et *call-reifier* différencient les appels des compounds selon leurs natures depuis une forme compilée. Elles sont traitées par l'évaluateur partiel comme des primitives. Le traitement qu'effectue l'évaluateur partiel sur ces fonctions permet de résoudre les changements de niveau, d'obtenir un code efficace, tout en conservant un certain degré de factorisation de ce dernier.

Le *call-compound*

La fonction Scheme *call-compound* gère les appels des compounds depuis une forme compilée en jouant le rôle d'une interface entre Scheme et notre interprète. L'écriture spécifique de la forme compilée de *apply-procedure* permet d'en réserver l'usage à l'appel entre elles des fonctions d'un même niveau d'interprétation. Le premier paramètre de *call-compound* sera toujours par construction soit un symbole vers la définition d'une fonction contenue dans l'environnement global, soit un compound, représentant une continuation du calcul. Dans les deux cas, et afin d'obtenir une suppression effective du niveau d'interprétation courant, l'évaluateur partiel doit essayer d'effectuer ces appels. *call-compound* traduit donc ces appels en des appels de fonctions traités directement par l'évaluateur partiel.

```
si proc est un symbole
  recherche de sa valeur dans l'environnement global
  et application
sinon proc est un compound (il s'agit des continuations)
  si le compound n'est pas compilé, compilation du compound
  application de la forme compilée du compound
```

Le *calling-compound*

L'objectif de *calling-compound* est de conserver les appels des fonctions de l'environnement global en empêchant l'évaluateur partiel d'intégrer le code des fonctions dans la forme compilée de la fonction en cours de compilation. Il s'agit bien sûr des fonctions du même niveau que celui de la fonction en cours de compilation, les fonctions exécutées au niveau de l'interprète de cette fonction étant elles effectivement appelées. La fonction *calling-compound* traduit un appel par l'interprète à la primitive *call*. D'après la sémantique que nous lui avons donnée, le premier paramètre doit être une fonction Scheme représentant la forme compilée d'un compound. Lorsqu'elle est partiellement évaluée, cette primitive se résidualise systématiquement en un *call-compound*. Le code de la fonction appelée n'est pas ainsi intégré au code de la fonction en cours de compilation. Si maintenant, la forme compilée résultante est appelée au niveau d'un interprète, le *call-compound* sera lui vu comme un appel à son premier paramètre par l'évaluateur partiel qui essaiera alors de le réduire.

Le premier paramètre de *calling-compound* étant une procédure Scheme, il faut être capable de retrouver le compound correspondant. La forme compilée d'un compound étant unique et propre à ce compound, une équivalence entre un compound et sa forme compilée peut être établie. La recherche du compound à partir de la connaissance de sa forme compilée est donc possible.

Le *call-reifier*

Un appel à une lambda réflexive se traduit par deux choses: la non évaluation des paramètres que lui passe l'utilisateur, et un changement de niveau d'interprétation. Le premier de ces mécanismes est géré directement par l'interprète. Le deuxième l'est par l'implémentation sous-jacente i.e. par la fonction Scheme *call-reifier*. Cette fonction ne se charge que de la partie «changement de niveau» de l'appel d'une lambda réflexive.

La spécificité de la forme compilée d'un compound réflexif est qu'elle peut s'exécuter directement en Scheme sans qu'il soit nécessaire d'ajouter un nouvel interprète pour en interpréter l'appel. Cette observation guide l'algorithme de l'évaluation partielle du *call-reifier*. L'appel est toujours effectué directement au niveau courant, la compilation de la lambda réflexive étant préalablement forcée lorsqu'elle ne contient pas de forme compilée.

> *si le reifier n'est pas compilé*
>   *compilation de la* **lambda** *réflexive*
> *application de la forme compilé de la* **lambda** *réflexive*

Les lambdas réflexives, sauf lorsqu'elles font l'objet d'une compilation, sont toujours des éléments du niveau de l'interprétation. Là encore, pour obtenir la réduction du niveau d'interprétation, leur appel doit être déroulé par l'évaluateur partiel.

5.5 Exemples d'utilisation

5.5.1 Une fonction simple, la factorielle

L'utilisation de l'évaluation partielle pour compiler l'interprète montre déjà son utilité avec de simples fermetures lexicales, plus particulièrement lorsqu'elles sont récursives. Un exemple d'une telle fermeture est la factorielle.

```
(set! fact (lambda default-applier (n)
             (if (< n 2)
               1
               (* n (fact (- n 1))))))
```

L'évaluation partielle de la factorielle a pour résultat la forme compilée suivante:

```
(lambda (n)
  (letrec ((fact-1
      (lambda (n k)
        (if (< n 2)
          (call-compound k 1)
          (fact-1 (- n 1)
                  (list 'compound
                        '(v)
                        '(evaluate-operands ...)
                        (let ((var-17 (list (cons 'n n))))
                          (cons (cons 'k
                                      (list 'compound ...))))
                        default-applier
                        (lambda (v)
                          (call-compound k (* n v)))))))))
    (fact-1 n (list 'compound
                    '(v)
                    'v
                    default-applier
                    (lambda (v) v)))))
```

Sur une simple fermeture lexicale, la compilation se manifeste par la suppression d'un niveau d'interprétation. Même si des constructions de haut niveau, comme la construction des compounds, restent présentes au sein du code, l'interprète n'apparaît plus dans le résultat de la compilation. L'interprète appelant directement la forme compilée d'un compound, l'exécution se passe ici directement en Scheme entraînant une efficacité incomparable avec un compound qui serait interprété.

Le langage utilisé par l'évaluateur partiel étant un sous-ensemble de Scheme plus important que celui traité par notre interprète, le code de la forme compilée utilise certaines constructions qui ne font pas partie de notre langage. En particulier, un **letrec** qui permet l'intégration au sein de la forme compilée de la récursivité directe de *fact*. Le calcul du résultat de la factorielle s'effectue au niveau Scheme sans qu'il y ait de passage par l'interprète lors des appels de la factorielle à elle même.

La factorielle se montre 20% plus lente qu'avec Scheme. Cette perte est due à la construction des compounds à chaque appel. Par contre elle est 60 fois plus rapide que sous I_{r^∞}. L'évaluation partielle permet donc un gain significatif.

5.5.2 Le *call/cc*

Le *call/cc* de Scheme permet d'obtenir la continuation du calcul. Son utilisation la plus courante est la construction de nouvelles structures de contrôle.

```
(set! call/cc
  (compound-to-reifier
    (lambda default-applier (e1 r1
                      (evaluate (car e1) r1
                        (lambda default-applier (closure)
                          (apply-procedure closure
                            (list (compound-to-reifier
                                    (lambda default-applier (e2 r2 k2)
                                      (evaluate (car e2) r2
                            k1)))))
```

L'obtention de la continuation du calcul ne peut se faire qu'en utilisant une lambda réflexive. Le *call/cc* s'écrit comme une lambda réflexive dont l'objectif principal est la capture de la continuation courant du calcul. L'utilisation d'une lambda réflexive nous contraint à évaluer le premier paramètre du *call/cc* afin d'obtenir la valeur de la procédure qui lui est passée. C'est à cette procédure que la continuation du calcul doit être passée.

Le dernier des paramètres passés à une lambda réflexive est la continuation du calcul au sein de l'interprète en cours d'exécution. Cette continuation constitue un transfert de contrôle à l'intérieur de l'interprète et non, comme dans les interprètes réflexifs basés sur la notion de méta-continuation, la composition d'un transfert de contrôle et d'un changement de niveau. Pour la rendre utilisable au niveau où l'appel au *call/cc* a lieu, il faut la composer avec un changement de niveau destiné à restituer le niveau correct d'interprétation. Ce changement de niveau est obtenu en enveloppant l'appel à la continuation dans une seconde lambda réflexive appelée depuis le même niveau que le *call/cc*.

La continuation passée en argument de la procédure passée au *call/cc* s'utilise comme une procédure à un paramètre. Ce paramètre constitue le résultat du calcul passé à cette continuation. Les paramètres passés aux lambdas réflexives n'étant pas évalués, l'autre tâche de la lambda réflexive enveloppant l'appel à la continuation de l'interprète est d'évaluer cet argument. La combinaison de cette évaluation et de l'appel à la continuation de l'interprète est réalisée par un appel à la fonction *evaluate* de notre interprète qui prend alors comme paramètres l'expression servant à calculer la valeur du résultat, suivi de l'environnement local dans lequel l'appel à la continuation du calcul a été effectué, suivi de la continuation de l'interprète capturée lors de la construction du *call/cc*.

La compilation du *call/cc* donne le résultat suivant:

```
(lambda (e1 r1 k1)
  (call-compound 'evaluate (car e1) r1
    (let ((var-8 (list (cons 'k1 k1)
                       (cons 'r1 r1)
                       (cons 'e1 e1))))
      (list 'compound
            '(closure)
            '(apply-procedure closure
               (list (compound-to-reifier
                       (lambda default-applier (e2 r2 k2)
                         (evaluate (car e2) r2 k1)))) k1)
            var-8
            default-applier
            (lambda (closure)
              (call-compound 'apply-procedure closure
                (list (call-compound 'compound-to-reifier
                        (list 'compound
                              '(e2 r2 k2)
                              '(evaluate (car e2) r2 k1)
                              (cons (cons 'closure closure) var-8)
                              default-applier
                              (lambda (e2 r2 k2)
                                (call-compound 'evaluate (car e2) r2 k1)))))
                k1))))))
```

Les principaux éléments de la définition du *call/cc* se retrouvent quasiment directement dans le corps de sa forme compilée. Le gain réalisé par la compilation de cette lambda réflexive semble se limiter à la suppression de son interprétation. Mais n'oublions pas le caractère réflexif de cette lambda: la compilation des compounds qui l'utilisent peut bénéficier de l'intégration au sein de leur forme compilée du résultat de l'appel au *call/cc*.

L'exemple qui suit ne s'illustre pas par l'intérêt qu'il présente d'un point de vu algorithmique. Sa qualité principale réside dans l'utilisation interne qu'il fait du *call/cc*, les conséquences de son utilisation étant ainsi circonscrites au corps du compound donné en exemple.

```
(set! test-call/cc
  (lambda default-applier (a b)
    (+ a (call/cc (lambda default-applier (cc)
                    (+ a (if (< a b) (cc b) b)))))))
```

La compilation d'un tel compound donne le résultat suivant:

```
(lambda (a b)
  (if (< a b) (+ a b) (+ a (+ a b))))
```

C'est ce résultat qui fait l'intérêt de l'exemple donné: toute référence directe ou indirecte à l'appel à *call/cc* disparaît dans la forme compilée, sa présence ne se traduisant que par les différences dans les expressions des deux branches du **if**. La suppression du *call/cc* est à assimiler à celle d'un niveau d'interprétation. En effet, le *call/cc* implémenté à l'aide d'une lambda réflexive s'exécute au même niveau que l'interprète en le modifiant dynamiquement.

L'évaluateur partiel applique au *call/cc* par conséquent le même traitement que celui des fonctions de l'interprète.

Ce résultat permet d'obtenir une fonction plus rapide que son équivalent Scheme. Le *call/cc* est une instruction lourde qui suppose une recopie de la pile d'exécution. Sa suppression dans le code final permet à notre interprète réflexif d'être plus rapide que l'interprète qui sert à son implémentation.

La nécessité, pour obtenir un tel gain, d'utiliser un exemple dans lequel les conséquences du *call/cc* sont circonscrites à l'intérieur du corps du compound trouve sa raison dans les implications de la méthode utilisée pour factoriser le code compilé au sein d'un même niveau d'interprétation. Cette méthode basée sur la résidualisation des appels des compounds globaux d'un même niveau entre eux s'oppose à la propagation lors de la compilation des modifications induites par le *call/cc* d'un compound à un autre.

Le calcul de la longueur d'une liste par une procédure acceptant n'importe quel type de valeur, exemple présenté dans [Abelson 91], illustre cet aspect. L'implémentation présentée fait appel à un **letrec** qui n'ayant pas d'équivalent dans notre langage doit être traduit par un découpage en deux compounds. *list-length* s'écrit alors:

```
(set! compute-list-length
  (lambda default-applier (return obj)
    (if (null? obj)
        0
        (if (pair? obj)
            (+ 1 (compute-list-length return (cdr obj)))
            (return #f))
        (return #f))))
```

```
(set! list-length
  (lambda default-applier (obj)
     (call/cc (lambda default-applier (return)
                  (compute-list-length return obj)))))
```

Le résultat de la compilation de *list-length* est:

```
(lambda (obj)
  (call-compound 'compute-length
    (let ((var-1 (list 'compound '(v) 'v '() default-applier
                       (lambda (v) v))))
       (list 'reifier
             '(e2 r2 k2)
             '(evaluate (car e2) r2 k1)
             (let ((var-2 (list (cons 'obj obj))))
               (cons (cons 'closure
                        (list 'compound
                              '(return)
                              '(compute-length return obj)
                              var-2
                              default-applier
                              (lambda (return)
```

```
                                    (call-compound 'compute-length return obj))))
                        (cons (cons 'k1 var-1)
                              (cons (cons 'r1 var-2)
                                    '((e1 (lambda default-applier (return)
                                            (compute-length return obj))))))))
              default-applier
              (lambda (e2 r2 k2)
                  (call-compound 'evaluate (car e2) r2 var-1))))
      obj))
```

Le contexte dans lequel la compilation de *compute-list-length* a lieu ne permet aucune optimisation basée sur la nature de **return**, sa valeur étant alors inconnue. Quant à la forme compilée de *list-length* elle doit se contenter de construire la continuation, ne sachant pas quelle en sera son utilisation.

En choisissant de ne pas factoriser les formes compilées au sein d'un même niveau d'interprétation, l'évaluation partielle aurait pu propager la connaissance qu'elle a du **return** de *list-length* sur l'appel à *compute-list-length*. En y ajoutant sa connaissance du résultat de la valeur passée à **return**, l'évaluateur partiel aurait pu supprimer totalement le niveau d'interprétation contenu dans la forme compilée de *list-length*.

5.6 Extensions possibles de l'interprète

L'extension la plus évidente est d'étendre le langage traité. Le sous-ensemble de Scheme que constitue le langage pris en compte par notre interprète est particulièrement limité. Une des limitations les plus importantes est l'absence totale de l'affectation. Le traitement des effets de bords, et plus particulièrement de l'affectation, par l'évaluation partielle est délicat et non trivial. Si les effets de bord comme la lecture d'un fichier ne peuvent, le plus souvent, qu'être laissés sous forme résiduelle, l'affectation peut, moyennant un traitement approprié, être parfois évaluée. Cependant, il faut remarquer qu'un évaluateur partiel comme Similix qui traite l'affectation de variables globales indirectement au travers de primitives, laisse les expressions les contenant résiduelles. Ce comportement risque de provoquer l'accumulation d'expressions résiduelles au cours de l'évaluation partielle des différents niveaux d'interprétation résultant en une forme compilée moins efficace. Comme toujours, la mise en œuvre de l'évaluation partielle reste une opération délicate.

L'évaluation partielle est habituellement conçue comme un pré-processeur de programmes [Ruf 93a]. L'utilisation en cours d'exécution du programme qui en est faite ici nécessite des adaptions et l'évaluateur partiel n'utilise pas directement les informations qui sont à sa disposition. Cet usage de l'évaluation partielle se traduit par un coût important. Il serait sans doute souhaitable de mieux intégrer l'évaluation partielle au sein de l'interprète, lui permettant ainsi de bénéficier de sa connaissance. L'utilisation de l'évaluation partielle dans un contexte dynamique a fait l'objet de travaux qui devrait pouvoir être mis à profit [Consel 96].

Une extension souhaitable pour donner une autre dimension a cet interprète est de lui donner la possibilité de modifier les composantes globales de chaque niveau d'interprétation. La réflexivité dans les langages fonctionnels ne devient vraiment intéressante que si elle intègre cette caractéristique. Des applications comme la modification de l'accès à l'environnement, ou encore de trace des expressions évaluées sont alors envisageables. Cependant, la mise en place de la réflexivité globale est rendue délicate par les effets de bords qu'elle suppose. L'évaluation partielle ne traite pas directement la modification de variables globales. Elle sait juste résidualiser les expressions qui lisent ou écrivent des variables globales. Or la modification d'une des définitions de l'interprète doit se traduire par une modification de l'ensemble des expressions qu'il interprète. Pour concilier les limitations de l'évaluation partielle avec la réflexivité globale, une forme linguistique spécifique pourrait être introduite. Les modifications des définitions de l'interprète seraient alors clairement identifiées et l'évaluateur partiel pourrait les traiter en conséquence.

5.7 Conclusion

Ce travail constitue à notre connaissance le premier travail sur la réduction d'une tour virtuellement infinie d'interprètes à l'aide de l'évaluation partielle dans un langage fonctionnel. L'objet de ces recherches était de montrer le gain performance que permet d'obtenir l'évaluation partielle lorsqu'elle est utilisée pour éliminer les différents niveaux d'interprétation induits par l'utilisation d'une implémentation basée sur la notion de tour virtuelle. Partant d'une discussion générale du problème, nous avons abouti à l'implémentation d'un interprète qui bien que limité démontre l'intérêt de l'utilisation de l'évaluation partielle dans le cadre de la réflexivité.

Mais ce travail nous a permis de mettre en lumière les problèmes relatifs à l'utilisation de cette technique. Certains de ces problèmes sont issus de la simple volonté de compiler: quelle que soit la technique employée, la compilation nécessite de fixer statiquement certaines informations et entraîne toujours la perte d'une partie de ces informations. L'élimination totale d'un niveau d'interprétation se traduit ainsi par la suppression de toute référence directe à l'interprète. La prise en compte par les fonctions compilées d'une modification même légère de leur interprète nécessiterait leur recompilation. D'autres problèmes sont plus spécifiques à l'évaluation partielle. Cette technique encore jeune n'a bien été développé que dans le cadre des langages fonctionnels. L'obtention d'interprètes plus réalistes passe nécessairement par son développement vers le traitement des cas moins «fonctionnels», en particulier par le traitement efficace des entrées-sorties et de l'affectation.

La réduction de la tour virtuelle par l'évaluation partielle n'élimine pourtant pas l'intérêt des techniques plus traditionnelles. Certaines applications nécessitent une réflexivité plus dynamique que celle offerte par cette technique. En particulier pour la mise en œuvre d'un environnement de déverminage d'un langage réflexif [Pazzaglia 97], la possibilité de modifier dynamiquement les niveaux méta représente un outil essentiel.

Deuxième partie

Définition et implémentation d'un modèle de réflexivité pour un langage à objets concurrents

Chapitre 6

Réflexivité dans les langages à objets

Par objet, nous entendons ici une entité qui réalise l'encapsulation des données et qui communique avec ses semblables par des envois de message. Cette définition assez générale permet de regrouper sous le vocable de «langage à objets» des familles aussi diverses, et parfois opposées, telles que les langages de classes, les langages à prototypes et les langages à objets concurrents. C'est dans cette dernière famille, qui est celle qui nous intéresse le plus dans la suite de cette thèse, que se situent les langages d'acteurs.

Dans ce chapitre, un certain nombre d'interprètes réflexifs sont examinés afin d'en déterminer les caractéristiques. Il ne s'agit pas ici de faire une étude exhaustive des interprètes réflexifs mais, à travers l'examen d'un certain nombre d'entre eux, de déterminer quelles sont leurs caractéristiques.

6.1 Réflexivité dans les langages de classes

Les langages de classes se caractérisent par la présence de deux entités, la classe et l'objet, la première servant de moule à la création de la seconde. Le grand succès de ces langages, y compris dans le monde de l'industrie, provient du concept d'héritage. De ce concept découlent les notions de réutilisation et de composant logiciel très prisées des industriels.

La réflexivité dans les langages de classes est organisée autour de la notion de méta-classe. La méta-classe est à la classe ce que la classe est à l'objet. Elle est directement issue de l'uniformisation du concept d'objet et de classe: lorsque la classe est traitée comme un objet, la classe d'une classe, sa méta-classe, naît naturellement du prolongement du concept d'instanciation de l'objet à la classe. La classe ne se différencie alors guère plus de l'objet que par sa capacité à créer des instances.

Ces langages fondent la relation méta/dénotation sur la relation d'instanciation. Ainsi, de même que la classe est le moule de plusieurs objets, la méta-classes est la classe de plusieurs classes. Il ressort de ce modèle qu'il est difficile de modifier les mécanismes d'un objet en

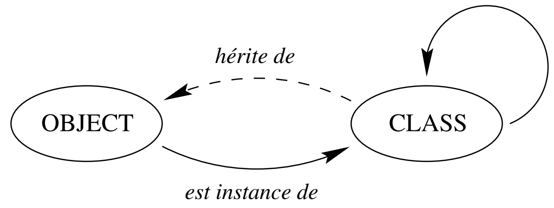

FIG. 6.1 – *Les classes CLASS et OBJECT d'ObjVlisp*

particulier.

6.1.1 ObjVlisp

ObjVlisp [Cointe 84, Cointe 87b, Cointe 87a] se distingue de tous les autres langages de classes par sa simplicité et la rationalité de son approche. Inspiré de Smalltalk-76 [Ingalls 78], il unifie les notions de méta-classe, classe et objet en exploitant le caractère uniforme du langage. Tout y est objet [1] et les méta-classes qui n'ont pas de statut spécifique comme dans Smalltalk-80 [Goldberg 83] sont librement accessibles à l'utilisateur.

L'interprète ObjVlisp se construit à partir de deux classes:

1. La classe CLASS est la racine de l'arbre d'instanciation. Elle est instance d'elle même et hérite de la classe OBJECT.

2. La classe OBJECT est la racine du graphe d'héritage. Elle est instance de CLASS.

La figure (Fig. 6.1) décrit les relations qui lient ces deux classes. Ces classes se différencient par leur méthode d'instanciation: la méthode **new** de CLASS crée une classe tandis que celle de OBJECT crée un objet.

La programmation réflexive est basée sur la création de nouvelles méta-classes par instanciation et héritage de CLASS. Ce procédé permet de créer autant de niveaux de méta-classes que souhaité.

6.1.2 Le MOP de Clos

Clos est l'extension vers le monde des objets de Common Lisp [Steele jr 90]. Il est caractérisé par son modèle étendu d'envoi de message basé sur la notion de fonction générique.

La réflexivité dans Clos est fondée sur la notion de méta-classe. Le système est lui même implémenté en se servant de cette notion. Le MOP [Kiczales 91], i.e. le «MetaObject Protocol», définit les méta-classes et objets du méta accessibles à l'utilisateur. La modification de ces derniers permet d'introduire de nouveaux mécanismes.

1. Ce «tout objet» là s'arrête aux trois entités précédemment citées. Les messages, les structures de contrôle et autres objets du langage ne sont pas des objets (au sens des langages à objets).

Le MOP de Clos est le plus reconnu des interprètes réflexifs. Sa notoriété est en grande partie due à la disponibilité d'une définition claire du MOP. Par cette démarche, Clos démontre qu'un interprète réflexif doit disposer d'une solide documentation pour être réellement utilisé.

La mise en place de la méta-classe d'une classe est assurée par le champ :metaclass associé à la définition de chaque classe. Bien que Clos autorise un objet à changer de classe, le modèle prône une association statique d'une classe à sa méta-classe.

6.2 Réflexivité dans les langages à prototypes

Par opposition aux langages de classes, les langages à prototypes ne différencient pas la classe de l'objet. Tout objet est susceptible de servir de modèle à une instance et la notion d'objet terminal est impropre à ces langages.

La réflexivité dans ces langages se distingue par une relation méta/dénotation établie sur une base individuelle. À chaque objet le modèle de réflexivité fait correspondre un méta-objet. Le méta-objet étant lui aussi un objet, il peut avoir un méta-objet. Une tour de méta-objet, virtuellement infinie, naît sur la base de chaque objet.

Le méta-objet représente les aspects structuraux et comportementaux de sa dénotation. Il est l'interprète des mécanismes de sa dénotation. La plupart du temps, pour des raisons de simplicité, la connexion causale est réalisée en donnant au méta-objet le statut d'implémentation de sa dénotation.

6.2.1 3-KRS

3-KRS [Maes 87] est un interprète réflexif basé sur KRS, langage à objet pour la représentation des connaissances [Steels 86]. Organisé autour de la notion d'objet, ce langage utilise un modèle de calcul basé sur l'envoi de message et la création d'objets. Une de ses particularités est que tout objet peut servir de modèle à un autre objet i.e. de prototype pour reprendre le terme de P. Maes (cf. page 127 de [Maes 87]).

Le modèle de réflexivité de 3-KRS est basé sur la notion de méta-objet. Tout objet de 3-KRS a un méta-objet. Cette notion est renforcée par l'uniformité du langage: c'est le langage du tout objet. Même les programmes y sont représentés par des objets.

La réflexivité dans 3-KRS est non seulement locale mais aussi globale[2] i.e. les objets qui composent le système sont eux aussi modifiables par l'utilisateur et causalement connectés au système.

Les nombreuses régressions infinies introduites par le modèle sont résolues par un mécanisme d'évaluation paresseuse.

2. P. MAES qualifie un système capable de réflexivité locale un système muni d'une architecture réflexive et un système capable de réflexivité locale et globale, un langage réflexif. Ce dernier terme ne nous convient pas dans le sens où ce n'est pas le langage mais le système qui l'implémente qui est réflexif.

La programmation réflexive passe dans 3-KRS par la spécification du méta-objet d'un objet à la création de cet objet (slot *meta*) (réflexivité locale) et par la substitution d'objets définis par l'utilisateur aux objets qui décrivent le système (réflexivité globale).

Les méta-objets introduits par P. Maes présentent de multiples avantages sur la notion de méta-classe: d'une part ils offrent une manipulation aisée des mécanismes individuels de chaque objet et d'autre part, n'étant pas basés sur l'utilisation extensive de la notion d'instanciation, ils offrent une séparation claire de la programmation réflexive et de la programmation usuelle.

6.3 Réflexivité dans les langages à objets concurrents

Les langages à objets concurrents intègrent la notion de concurrence aux objets. L'objet devient alors actif et dispose de sa propre capacité de calcul. L'envoi de message passe de l'attachement procédural à un véritable envoi de message dans lequel le message transmis à l'objet est stocké dans une boîte aux lettres. C'est le destinataire du message qui décide de son traitement et non plus l'expéditeur.

Les modèles de réflexivité mis en œuvre dans ces langages sont multiples. Ils vont de l'architecture individuelle à base de méta-objets à la réflexivité de groupe en passant par des modèles hybrides qui combinent les deux premiers.

6.3.1 ABCL/R

ABCL/R [Watanabe 88] est le premier des langages réflexifs de la famille ABCL. Implémenté au dessus d'ABCL/1 [Yonezawa 86b, Yonezawa 86a], un langage à objets concurrents basé sur le modèle acteur d'Hewit [Hewit 73], il inaugure l'introduction de la réflexivité dans les langages à objets concurrents.

La réflexivité dans ABCL/R est basée sur la notion de méta-objet, un objet qui implémente un autre objet, sa dénotation. Les aspects structuraux de la dénotation y sont décrits par les variables d'état du méta-objet tandis que ses méthodes décrivent les aspects comportementaux. En tant qu'implémentation de sa dénotation, le méta-objet est causalement connecté à sa dénotation. La syntaxe d'un méta-objet (extraite de [Watanabe 88]) est la suivante:

```
; Un méta-objet
[object
  (state ; Variables d'état du méta-objet = aspect structuraux de la dénotation
    [queue := queue des messages de la dénotation]
    [state := Variables d'état de la dénotation]
    [scripset := liste des méthodes de la dénotation]
    [evaluator := évaluateur rattaché à la dénotation]
    [mode := mode dans lequel se trouve le méta-objet :dormant ou :active])
  (script ; Méthodes du méta-objet = aspects comportementaux
    (=> [:message Message Reply-Dest Sender] ; arrivée et réception des messages
         ... )
    (=> :begin ; acceptation et exécution du script
```

```
        ... )
    (=> :end ; Terminaison de l'exécution
        ... ))]
```

À chaque méta-objet est attaché un évaluateur local. Cet évaluateur est un objet du méta chargé de l'évaluation des instructions contenues dans le script de la dénotation.

Par défaut, tout objet dispose d'un méta-objet. Les méta-objets qui sont eux aussi des objets ont donc un méta-objet. La régression infinie du modèle est résolue par un système de construction paresseuse des méta-objets

L'envoi d'un message m d'un objet O_1 à un objet O_2 se traduit par l'envoi d'un message [:message *m reply-dest* O_1] du méta-objet de O_1 au méta-objet de O_2.

L'installation d'un méta-objet défini par l'utilisateur se fait lors de la déclaration de la dénotation grâce au champ *meta*. La syntaxe d'un objet qui déclare un méta-objet est donc la suivante:

```
[object
  (meta le méta-objet)
  (state ... )
  (script ... )]
```

Cette construction linguistique qui est la seule à permettre l'installation d'un méta-objet spécifique induit un modèle de programmation réflexive dans lequel l'utilisation de la réflexivité doit être pensée avant la création d'un objet.

Implémenté au dessus d'ABCL/1, ABCL/R souffre de l'existence de deux niveaux d'interprétation.

6.3.2 ABCL/R2

ABCL/R2 [Masuhara 92, Matsuoka 91] a bénéficié de l'expérience acquise avec ABCL/R et ACT/R. Il tente de conjuguer les avantages de ces deux modèles de réflexivité à travers une approche hybride qui combine les méta-objets avec la réflexivité de groupe. Le premier de ces modèles s'attache à modéliser les propriétés individuelles de chaque objet alors que le second modélise les ressources partagées par un groupe d'objets. Par conséquent, deux types de tours coexistent, l'une basée sur les méta-objets et l'autre sur les groupes.

ABCL/R2 innove aussi par son implémentation.La grande attention apportée à cette dernière s'est traduite par un interprète significativement plus rapide qu'ABCL/R. Parmi les techniques employées, notons la compilation partielle des scripts, l'introduction des objets légers, un mécanisme de création des tours individuelles basé sur la «progression dynamique de la réflexivité» et la compilation des objets non-réifiables.

6.3.3 ABCL/R3

Avec ABCL/R3 [Masuhara 94, Masuhara 95], le langage de base passe d'ABCL/1 à un langage à objets concurrents , ABCL/*f* [Taura 94], qui fait des futures, l'envoi de message de base. Le modèle de réflexivité d'ABCL/R3 reste pourtant le même que celui d'ABCL/R2.

La principale innovation d'ABCL/R3 réside dans son implémentation. Elle fait appel à l'évaluation partielle pour améliorer l'efficacité de l'exécution du système. Dans le modèle de réflexivité mis en jeu, la principale source d'inefficacité est l'évaluation des instructions du langage. L'évaluation partielle est donc utilisée pour compiler le couple (script, évaluateur associé) en un programme en ABCL/*f* dans lequel les constructions réflexives ont pour la plupart disparues.

La compilation d'un programme ABCL/R3 se découpe en trois phases:

1. Un pré-processeur transforme l'évaluateur, un objet d'ABCL/R3, en un ensemble équivalent de fonctions en Common Lisp écrites en continuation explicite.

2. L'évaluateur équivalent écrit en Common Lisp est spécialisé en fonction de chacune des méthodes qu'il doit évaluer. Le résultat de chaque évaluation partielle est un programme en Common Lisp.

3. Un post-processeur transforme le résultat de l'évaluation partielle en un programme équivalent en ABCL/*f*.

L'évaluation partielle a donc pour résultat d'éliminer le niveau d'interprétation correspondant à l'évaluation par un objet du méta, l'évaluateur associé à un groupe, des méthodes des objets dénotés. Le gain obtenu semble réel. Cependant la technique n'est pas étendue à l'ensemble des tours qui peuvent exister dans ABCL/R3. En particulier, les tours de méta-objets ne semblent pas en bénéficier.

6.3.4 ACT/R

ACT/R [Watanabe 90] introduit la notion de réflexivité de groupe. Basé sur le modèle acteur défini par G. Agha [Agha 86], il n'établit pas de connexion causale entre un acteur et son méta-acteur, mais entre un groupe d'acteurs et un autre groupe d'acteurs qui décrit le premier au niveau méta.

Les acteurs de niveau méta décrivent le calcul de la dénotation, le groupe d'acteurs qu'ils représentent. Ils décrivent aussi les interactions avec les éléments externes au groupe. La programmation réflexive consiste alors à modifier les composants du méta.

L'utilisation d'un groupe d'acteurs pour en décrire un autre présente avantages et inconvénients. L'esprit qui sous-tend ce modèle de réflexivité le rapproche du modèle fonctionnel pour lequel chaque niveau est représenté par l'interprète de la dénotation. Enfin, ce modèle de réflexivité, si il prend mieux en compte les ressources disponibles, pose des problèmes identiques au méta-classes: il rend plus difficiles la description et la manipulation individuelle des acteurs.

6.3.5 ReActalk

ReActalk [Giroux 93] est une extension vers la réflexivité de Actalk [Briot 89], un ensemble de classes qui étendent Smalltalk vers le monde des acteurs. L'introduction d'objets actifs communicant par envois de message asynchrone, les acteurs, est obtenue par l'encapsulation d'objets de Smalltalk dans des sérialiseurs.

Fortement influencé par les problèmes d'organisations, le modèle de réflexivité de ReActalk est construit autour de la notion d'usine. À chaque acteur, ReActalk associe un méta-acteur dont les diverses activités sont déléguées à des acteurs spécialisés. L'ensemble méta-acteur et acteurs spécialisés forme ce que S. Giroux nomme une usine. Cette approche apporte une souplesse intéressante à la programmation de la réflexivité: la modification du mécanisme de réception des messages s'effectue en substituant à l'acteur spécialisé dans cette tâche un acteur ad hoc. Il n'est donc plus nécessaire de modifier l'intégralité du méta-acteur.

Un acteur de ReActalk dispose de deux autres descriptions au niveau méta: sa classe, description héritée de Smalltalk, et son écosystème. L'écosystème d'un acteur régit ses interactions avec les autres acteurs, son comportement «social». La réification de ce dernier au méta-méta-niveau est réalisée par une organisation d'agents.

6.3.6 GMAL

GMAL [Hassas 93] est un interprète réflexif pour un langage d'acteurs. Le modèle de calcul utilisé ici est celui défini par G. Agha [Agha 86].

Il existe deux versions de GMAL. La première n'est pas réflexive et est construite au dessus de Common Lisp. La seconde, réflexive est construite au dessus de la première. L'approche utilisée ici est donc celle d'ABCL/R.

Le système GMAL est modélisé par quatre acteurs qui représentent les fonctions essentielles d'un système d'acteurs. Ces acteurs sont l'acteur *BEHAVIOR*, l'acteur *MESSAGE*, l'acteur *ACTORENTITY* et l'acteur *REPLACEMENT*. Ils définissent les comportements de base du système et sont tous instance de l'acteur *BEHAVIOR* (qui est donc sa propre instance). Notons que les messages de GMAL sont des messages actifs i.e. des messages qui «se transmettent» eux mêmes.

L'ensemble des instructions «acteurs» de GMAL s'expriment à l'aide d'envois de messages asynchrones. La création d'un acteur et le changement de comportement sont obtenus par l'envoi d'un message, respectivement *CREATE* et *BECOME*. Tout envoi de message se traduit par la création d'un acteur message qui s'expédie à son destinataire.

Tout acteur de GMAL est implémenté par un méta-acteur instance de *ACTORENTITY*.

Le modèle mis en place dans GMAL se singularise par sa fidélité au modèle de calcul défini par Agha. Cependant le nombre de niveaux méta est limité à deux. GMAL se rapproche donc plus d'un méta-système que d'un système réflexif.

L'utilisation de la première version de GMAL pour implémenter la version réflexive est un sérieux handicap.

6.3.7 La réflexivité selon Tanaka

T. Tanaka [Tanaka 90] propose un langage d'acteurs, basé sur le modèle de calcul défini par G. Agha [Agha 86] dans lequel la réflexivité n'est pas basée sur des méta-acteurs mais sur des messages réifiants et réfléchissants. Les premiers de ces messages correspondent à une réification des données qui décrivent l'acteur. Il s'agit du filtre des messages acceptés par l'acteur[3], de son script, composé d'une seule commande, de ses accointances (nommées variables d'états) et de la queue des messages en attente de traitement. Un message réifiant, construit par la fonction *make-reify-message*, comporte une continuation et un paramètre qui permet de geler l'acteur réifié i.e. de bloquer l'acteur jusqu'à ce que ses données soient réinstallées par un message réfléchissant. La continuation est l'adresse de l'acteur qui va recevoir la forme réifiée et la traiter. Cet acteur peut installer de nouvelles données à l'aide d'un message réfléchissant.

Le mode de calcul introduit par les messages réifiants et réfléchissants diffère fortement du modèle à base de méta-acteurs. La relation entre une dénotation et un méta y est éphémère. Si ce modèle de réflexivité accède aux données de l'acteur, il ne fait pas de l'acteur qui joue le rôle de méta l'interprète des mécanismes de sa dénotation.

À notre sens, les deux modèles ne sont pas équivalents. L'argument de T. Tanaka selon lequel les messages réifiant et réfléchissant pourraient se substituer dans tous les cas aux méta-acteurs n'est pas fondé. L'exemple qui le soutient, la trace de message, démontre justement la différence essentielle entre les deux modèles: ses messages ne permettent pas une programmation des mécanismes de niveau méta. La trace est installée par une modification de la dénotation à la manière de ce qui peut être fait en Lisp pour tracer les appels à une fonction. Avec les méta-acteurs, ce sont les mécanismes de l'acteur qui sont alors modifiés et non ses composantes. Le modèle de T. Tanaka ne dissocie pas clairement la programmation réflexive de la programmation usuelle. Il introduit donc une confusion entre ce qui relève du méta et ce qui appartient au niveau utilisateur.

Malgré tout, l'approche reste intéressante par les alternatives qu'elle suggère. Le style de programmation réflexive qu'elle suppose reste à explorer. Et l'implémentation de la réflexivité qu'elle sous-tend est probablement moins lourde et plus efficace que celle des méta-acteurs.

6.3.8 Mering IV

Mering IV [Ferber 89, Ferber 91, Carle 92] est par essence un langage hybride qui unifie acteurs, objets et agents. Les acteurs constituent les entités de base de Mering IV, entités sur lesquelles sont construites les objets. L'utilisation du modèle acteur comme modèle de calcul fait de ce langage un langage concurrent. Le modèle objet de Mering IV s'inspire de ObjVlisp.

3. Les acteurs de T. Tanaka n'ont qu'un seul filtre.

Chaque acteur de Mering IV peut avoir un méta-acteur instance d'une classe, son *smodel*. Les objets, instances de classes, peuvent eux aussi avoir un méta-objet instance d'un smodel. Chaque classe, en tant qu'instance d'une autre classe, dispose d'une méta-classe, appelée *hyperclasse*. Ces deux axes de réflexivité orthogonaux sont complémentaires, chacun d'entre eux décrivant des aspects complémentaires d'une même entité. Au sommet de la hiérarchie des classes et des objets se trouve le duo bien connu d'ObjVLisp, les classes *Class* et *Object*.

Mering IV se singularise par un modèle de réflexivité opératoire fondé sur la compilation de programmes réflexifs. Ces derniers modifient les règles d'exécution des programmes utilisateurs. Cette technique est basée sur l'utilisation d'un compilateur spécifique dédié à la compilation des programmes réflexifs. Le respect de la connexion causale nécessite la recompilation de l'ensemble des expressions dont les règles d'exécution sont modifiées. Le compilateur des expressions du langage tient compte des règles modifiées pour générer le code final.

Ainsi que P. Carle [Carle 92] le fait remarquer, le modèle de réflexivité opératoire employé par Mering IV se caractérise par son efficacité. L'exécution du code final ne souffre pas des multiples niveaux d'interprétation que supposent les modèles interprétatifs. La recompilation systématique des expressions concernées en est l'inconvénient majeur.

Par son modèle de réflexivité opératoire, Mering IV se rapproche des systèmes à implémentation ouverte préconisés par G. Kiczales [Lamping 92, Kiczales 93].

6.4 Analyse de l'existant et objectifs

L'objectif de nos travaux est de définir et d'implémenter un modèle de réflexivité pour un langage d'acteurs «à la»Agha [Agha 86]. Pour les recherches effectuées dans notre équipe, nous devions disposer d'un interprète réflexif qui soit simple à utiliser, avec un large champ d'application et une mise en place dynamique de la réflexivité. La disponibilité de l'interprète et ses aspects pratiques étaient des critères primordiaux.

Trois des interprètes examinés utilisent, plus ou moins directement, le modèle de calcul de G. Agha: ACT/R, GMAL et ReActalk. Des trois interprètes précédents, seul ReActalk était disponible. Nous allons examiner ici plus en détail ces interprètes par rapport à nos objectifs.

La réflexivité de groupe propre à ACT/R [Watanabe 90], si elle convient bien au partage de caractéristiques entre différents acteurs, se révèle moins adaptée à la modification des mécanismes qui régissent un seul individu. Par exemple, il rend difficile la modification des envois de messages effectués par un seul individu. L'évolution de la famille ABCL vers ABCL/R2 [Masuhara 92] est à ce sujet éloquente: ce dernier concilie l'approche individuelle, basée sur les méta-objets, avec la réflexivité de groupe afin de pallier aux carences des deux modèles. Par ailleurs, pour nos recherches, la possibilité d'intervenir au niveau d'un individu est plus importante que les mécanismes mis en valeurs par la réflexivité de groupe.

GMAL [Hassas 93] n'est, à notre connaissance, pas disponible. Le modèle de réflexivité s'intègre parfaitement au monde des acteurs. L'efficacité de l'exécution n'est cependant pas prise en compte par S. Hassas. L'utilisation de GMAL pour implémenter l'interprète réflexif ne peut que conduire à des performances insuffisantes pour notre propos. GMAL est lui-même implémenté sur une machine virtuelle, la machine de Gosseyn, qui utilise comme modèle de calcul, le calcul des prédicats. Cette accumulation de différents niveaux d'interprétation a du mal à se concilier avec notre contrainte sur l'efficacité.

ReActalk [Giroux 93] met l'accent sur la réflexivité organisationnelle. Il offre une décomposition du méta sous forme d'agents spécialisés dans une tache précise et qui communiquent entre eux. L'ensemble des agents qui réifient un acteur est lui-même réifié vers une société d'agents qui en reflète la structure ainsi que les interactions entre agents. Ces descriptions du niveau méta et du niveau méta-méta sont régies par des préoccupations axées sur les aspects organisationnels. L'utilisation d'une description à un niveau et d'une autre au niveau au dessus va à l'encontre du caractère récursif propre à toute tour d'interprètes. Or, l'un des axes de recherche de notre équipe est le déverminage de programmes réflexifs. Dans ce cadre, il faut pouvoir intervenir à n'importe quel niveau d'interprétation afin de déverminer le niveau qui se trouve juste en dessous.

Si Actalk [Briot 89] constitue une plate-forme idéale pour l'expérimentation des objets concurrents, son extension vers la réflexivité souffre de la multiplicité des points de vue. L'utilisation de la réflexivité dans ReActalk suppose la maîtrise de nombreux concepts. Elle est à la fois basée sur des concepts propres à Smalltalk, comme l'utilisation des classes pour réifier les aspects structurels, et des concepts introduits par ReActalk. Cette multiplication des concepts nous semble aller à l'encontre de la simplicité que nous recherchons.

Deux des caractéristiques recherchées sont ignorées par ces trois interprètes: la mise en place dynamique de la réflexivité et l'efficacité de l'exécution.

La première d'entre elle, la mise en place dynamique de la réflexivité, est souhaitable dans le cadre du déverminage [Pazzaglia 97]. Lorsqu'une erreur est signalée, il faut pouvoir extraire des informations auxquelles seule la réflexivité donne accès. La mise en place des éléments réflexifs qui permettent d'accéder à ces informations avant de lancer le programme fautif dénature son fonctionnement et risque de faire disparaître certaines erreurs. La mise en place dynamique de la réflexivité permet de réifier les éléments fautifs qu'après que l'erreur se soit produit.

L'efficacité des systèmes réflexifs est un paramètre qui conditionne fortement leur utilisation. Comme nous l'avons vu dans la première partie de cette thèse, un système «tout réflexif» qui permet entre autres la réflexivité globale se traduit par des performances en contradiction avec une utilisation journalière. Les recherches effectuées dans notre équipe nécessitent un interprète dont l'efficacité soit compatible avec une utilisation quotidienne. Aussi, si notre besoin de réflexivité était manifeste, le traitement de l'efficacité primait sur la beauté du modèle.

LActE/R est né de l'absence d'interprète réflexif qui réponde à l'ensemble de nos attentes. Et, parce que ces attentes étaient avant tout d'ordre pratique, nous l'avons défini en tenant compte des demandes d'un utilisateur averti, J. C. Pazzaglia, avec pour objectif d'obtenir un interprète dont le modèle de réflexivité soit simple à mettre en œuvre, dynamique et efficace. LActE/R est donc le résultat d'une approche pragmatique de la réflexivité.

6.5 Conclusion

Les modèles de réflexivité des langages à objets sont très nombreux. Cette multiplicité est la traduction non seulement des différentes familles de langages, mais aussi de conceptions. Elle montre les différences d'appréciations dans ce que doit être l'introduction de la réflexivité dans le monde des objets.

Ces interprètes réflexifs s'accordent pourtant sur un point: la distribution de la relation méta/dénotation sur les objets. Alors que les langages fonctionnels réifient l'ensemble de l'interprète vers un méta-interprète, les langages à objets établissent une relation dans laquelle le méta n'est méta que d'un sous-ensemble des objets du système.

Dans la suite de cette seconde partie, nous allons montrer comment cette distribution de la relation méta/dénotation peut contribuer significativement à l'amélioration des performances d'un interprète réflexif.

Chapitre 7

LActE/R, une approche pragmatique de la réflexivité

LActE/R, acronyme de «Lisp Actor Environment / Reflective», résulte d'une approche pragmatique de la réflexivité introduite dans un interprète préexistant, LActE. Il s'agissait pour nous d'allier des contraintes sur l'efficacité du modèle avec d'autre concernant le dynamisme du langage et des aspects réflexifs.

L'introduction de la réflexivité dans LActE est la conséquence de l'analyse des besoins survenus lors de l'utilisation du langage. Le développement d'applications écrites à partir de ce langage a mis en évidence ses carences. LActE est en effet basé sur une approche minimaliste du modèle acteur telle qu'elle est proposée dans la thèse de G. Agha [Agha 86], approche enrichie d'une typologie complète d'envois de message. Cette approche, qui a ses mérites, présente vite ses limites lors de l'écriture d'applications d'une certaine taille (cf. le développement de Demact ([Michel 90], [Michel 92] et [Michel 93])). Pour pallier aux limites de LActE, l'utilisateur devait passer par une modification directe de l'implémentation, ce qui n'était rendu possible que par la connaissance de cette dernière.

La réflexivité, elle, rend possible la modification de différents aspects du langage sans passer par une modification directe de son implémentation. Elle confère au langage une souplesse qu'aucun autre paradigme n'a encore égalée en permettant à l'utilisateur de modifier et enrichir les mécanismes du langage. Au regard de ces avantages, l'introduction de la réflexivité dans LActE nous a paru être la solution à son manque de souplesse.

LActE/R est le résultat de cette démarche visant à doter LActE de capacités réflexives, le rendant par là-même apte à répondre aux besoins d'évolutivité et de souplesse qui lui ont fait jusqu'alors défaut.

Dans ce premier chapitre, nous allons décrire le modèle LActE/R. Le chapitre suivant en décrit l'implémentation tandis que le troisième illustre l'utilisation de LActE/R à l'aide d'exemples.

Afin que le lecteur ait une meilleure compréhension de LActE/R, nous commençons par donner un aperçu de LActE avant d'aborder le modèle de réflexivité de LActE/R.

7.1 LActE, une couche acteur au dessus de Common Lisp

Construit comme une couche au dessus de Common Lisp [Steele jr 90], LActE [Regourd 94] s'appuie sur un modèle de calcul bien défini, le modèle acteurs, qu'il enrichi d'une typologie complète d'envois de messages et d'acteurs sérialisés et non sérialisés. Nous présentons ici ses caractéristiques essentielles afin que le lecteur puisse mieux comprendre les différentes articulations entre l'implémentation classique de LActE et la version réflexive qu'est LActE/R.

7.1.1 Le modèle de calcul

La notion d'acteurs fut introduite par C. Hewit dès le début des années 70 afin de distribuer le raisonnement d'un robot [Hewit 69]. En s'inscrivant dans la mouvance des adeptes d'une intelligence artificielle distribuée, par opposition à la vision centralisée des systèmes classiques, Hewit a défini un modèle de calcul distribué et asynchrone basé sur une notion d'objets autonomes et actifs [Hewit 73], disposant de leur propre capacité de calcul, et qui communiquent entre eux par envois de message asynchrones et unidirectionnels. Comme l'objet, l'acteur réalise l'encapsulation des données et dispose d'une connaissance finie sous la forme de la liste de ses accointances et d'un comportement qui définit les actions dont il est capable.

Le modèle de calcul adopté par LActE est celui défini par G. Agha dans sa thèse [Agha 86] dans laquelle il formalise le concept d'acteur. Pour lui, la machine acteur est composée d'une boîte aux lettres à la capacité virtuellement infinie, de la liste de ses accointances ainsi que d'un comportement (Fig. 7.1 page suivante). L'activité de l'acteur consiste alors à sélectionner l'un des messages contenus dans sa boîte aux lettres, puis à générer une tâche support de l'activité de l'acteur traitant un message. Le traitement du message est fonction du comportement de l'acteur et se compose d'un ensemble d'actions qui sont de trois types :

1. L'*envoi d'un message* à un autre acteur qu'il connaît soit qu'il appartienne à la liste de ses accointances, soit que cette connaissance lui ait été transmise par le message qu'il traite. Cet envoi est asynchrone et unidirectionnel, le traitement d'une éventuelle réponse devant se faire de manière explicite.

2. La *spécification d'un comportement de remplacement.* L'acteur a alors la possibilité de modifier les valeurs de ses accointances et, ou, d'adopter un nouveau comportement. La modification de comportements est un trait caractéristique de ce modèle qui en explique le caractère dynamique. De plus, dès lors qu'il a spécifié un comportement de remplacement, l'acteur devient capable de traiter un autre message reçu même si le traitement en cours n'est pas encore terminé. Le programmeur a donc les moyens d'effectuer un contrôle fin de la granularité du parallélisme.

3. La *création d'un autre acteur.* Cette possibilité renforce encore le caractère dynamique

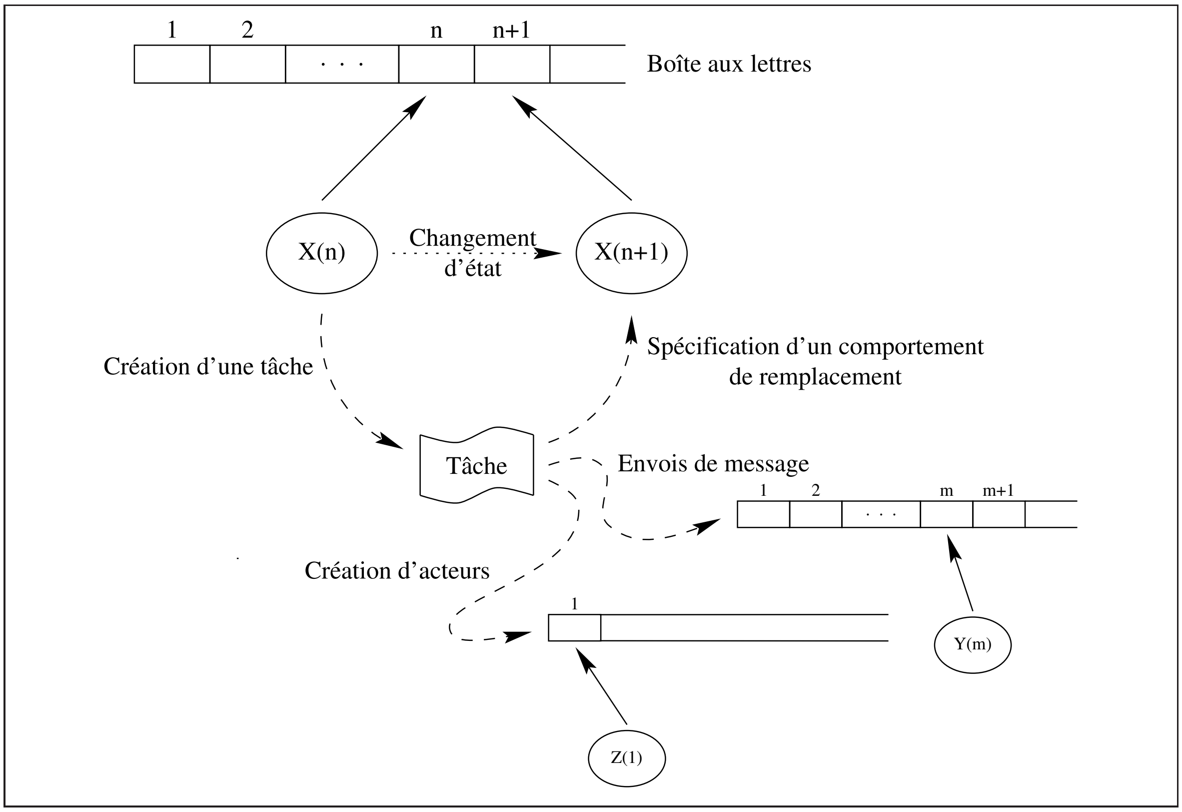

FIG. 7.1 – *La machine acteur selon G. Agha*

du modèle en permettant à un acteur d'étendre dynamiquement l'ensemble des acteurs qu'il connaît.

Ces trois actions sont le support du calcul au sein d'un système d'acteurs. Par leur sémantique, le dynamisme et le parallélisme du modèle de calcul sont mis en avant. En particulier, elles confèrent à un système d'acteurs une topologie dynamique.

Le fonctionnement du modèle s'appuie sur deux principes:

1. La *garantie de délivrance des messages*: Tout message expédié sera reçu par son destinataire. Ce principe est nécessaire au bon déroulement du calcul. Par contre, rien n'est dit quant au délai de délivrance des messages et à l'ordre de leur réception, un message pouvant être reçu avant un autre message pourtant expédié avant le premier.

2. L'*égalité de tous les acteurs devant l'attribution des ressources* disponibles. Ce principe évite les problèmes de famine et assure qu'un calcul sera effectif, aucun acteur ne se trouvant bloqué par accaparement des ressources du système par un autre acteur.

Ces deux principes permettent de garantir qu'un calcul se fera, pour autant que son implémentation soit correcte.

7.1.2 La couche LActE

Le modèle de calcul qui précède est celui adopté par LActE. Cependant, notre couche acteur est un peu moins minimaliste que l'approche prônée par G. Agha. Elle enrichit le modèle défini par G. Agha.

LActE distingue deux types d'acteurs:

1. Les *acteurs sérialisés* qui correspondent aux acteurs tels que les définis G. Agha dans son modèle. Ils utilisent l'instruction **become** pour spécifier un nouveau comportement. Dans le cas où aucun comportement de remplacement n'a été spécifié, la fin du traitement du message courant tient lieu de **become** par défaut, les valeurs des accointances et le comportement courant étant conservés.

2. Les *acteurs non sérialisés*. Ces acteurs se distinguent des premiers par leur incapacité à spécifier un comportement de remplacement. Les valeurs des accointances et le comportement définis à la création de l'acteur restent les mêmes pour toute la durée de vie de l'acteur. L'avantage de ces acteurs est de traiter les messages reçus dès leur réception, l'attente de la spécification d'un comportement de remplacement n'étant plus nécessaire. Un tel acteur a ainsi la possibilité de traiter plusieurs messages en même temps.

Le modèle acteur se base sur un seul type d'envoi de message, l'envoi asynchrone. La notion de réponse à un calcul n'est pas directement prise en compte par cet envoi et son traitement explicite demeure à la charge du programmeur. LActE bénéficie d'une typologie complète d'envois de message issue de ABCL/1 ([Yonezawa 86b], [Yonezawa 86a]). Cette typologie est constituée de trois types de messages (Fig. 7.2 page ci-contre):

1. L'envoi de message *asynchrone* ou **send**. Constituant l'envoi de message de base du modèle acteur, c'est le plus utilisé des envois de messages. Cependant, l'expression de certains calculs en mode purement asynchrone rend parfois la programmation par acteurs un peu absconse. Aussi l'introduction des deux autres types d'envois de messages est complémentaire et facilite l'expression de toutes les formes de calcul.

2. L'envoi de message *synchrone*, ou **ask**, dans lequel le calcul courant est suspendu jusqu'à la réception d'une réponse à la requête émise. Avec cette forme d'envoi de message la demande d'un calcul et le traitement de sa réponse peuvent avoir lieu au sein d'une seule et même méthode.

3. L'envoi de message *asynchrone avec promesse de résultat* ou **future**. Ce type d'envoi de message combine les avantages des deux premiers: le calcul n'est pas suspendu après l'envoi du message, l'acteur conservant ainsi la possibilité d'accomplir d'autres calculs jusqu'au traitement de la réponse. La réponse est reçue par l'intermédiaire d'un objet

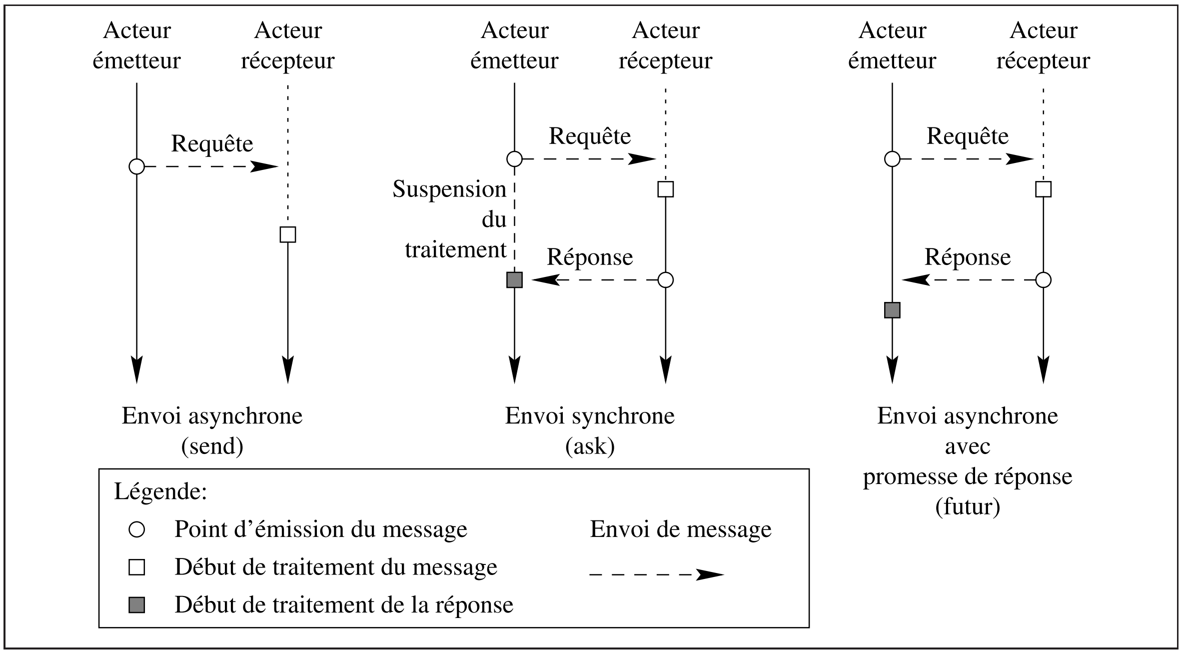

FIG. 7.2 – *Typologie des envois de message dans LActE*

spécifique qui peut être passé à un autre acteur. Une grande souplesse dans la façon de traiter la réponse est offerte par ce biais.

Dans sa thèse, G. Agha [Agha 86] utilise des comportements à la forme épurée qui laisse au programmeur le soin de filtrer explicitement les messages reçus. Bien que ce modèle soit suffisamment expressif, il ne convient pas au développement de programmes de taille importante. Dans LActE la tâche du programmeur est facilitée par la décomposition des comportements en méthodes, associations d'un filtre et d'un corps de méthode. Le filtre sert de support au filtrage des messages, la méthode déclenchée correspondant à la première méthode dont le filtre accepte le message.

Une autre différence entre le modèle originel de G. Agha et LActE est le mode d'exécution des méthodes: les instructions des méthodes de LActE sont exécutées de manière séquentielle contrairement à l'approche parallèle prônée par Agha. Cette simplification se justifie d'une part par le support matériel ciblé par LActE, les réseaux de stations de travail, pour lequel une multiplication inconsidérée du nombre de tâches engendre des phénomènes indésirables pouvant aller jusqu'à l'effondrement du système, et d'autre part, par notre volonté d'obtenir un modèle de parallélisme qui reste facilement manipulable.

LActE est doté d'un mécanisme de partage des connaissances, la délégation. Chaque acteur de LActE est doté de l'adresse d'un autre acteur, son *proxy*, auquel il envoie les messages non reconnus. Le proxy est censé être capable de traiter ces messages. Dans le cas contraire, il les envoie à son propre proxy. Une chaîne d'acteurs qui partagent ainsi leurs connaissances peut être établie.

LActE dispose par ailleurs de nombreuses autres facilités qui vont de la possibilité de créer des acteurs nommés et des acteurs anonymes, de constructions spécifiques qui, en combinant création d'acteur et description de comportement en une seule instruction, rendent aisée l'utilisation des continuations et des jointures [Carle 92]. Une description plus complète de LActE est disponible dans [Regourd 94].

7.1.3 LActE en action

L'exemple de la factorielle choisi pour illustrer le fonctionnement de LActE l'a été afin de montrer la différence d'esprit entre la programmation dans les langages plus classiques comme Scheme [Abelson 91] tels que nous les avons utilisés dans la première partie de cette thèse et la programmation par acteurs. En LActE, l'écriture et l'utilisation d'une factorielle prennent la forme suivante:

```
  ; Une session LActE
  ; Définition du comportement d'une factorielle en LActE
> (def-behavior a-fact() nil
       ((n)
        (if (< n 2)
           (result 1)
           (send [(- n 1)] :to self :forward-to
                 (continuation ((next forward-to))
                        ((value)
                         (send [(* n value)] :to next)))))))
#<BEHAVIOR a-fact: ...>
; Création de l'acteur factorielle
> (new fact a-fact ())
#<ACTOR fact: An unserialized actor>
; Calcul de la factorielle de 5
> (send [5] :to fact)
T
>
  ; Le top level de Lisp est utilisé par défaut comme destinataire
  ; de la réponse au calcul
Top-level: received (120) from CONTINUATION-1541
```

Le comportement défini ici est nommé *a-fact*. La liste qui suit le nom du comportement correspond à la liste des noms des accointances. Pour la factorielle, cette liste est vide, ce comportement ne nécessitant aucune accointance. Cette liste est suivie d'une expression dont la valeur doit être un booléen à vrai dans le cas d'un comportement d'acteur sérialisé et faux pour les non sérialisés.

Le filtre de l'unique méthode de la factorielle est (n). Il filtre tous les messages composés d'un seul et unique élément. Bien que le type attendu de cet élément soit un entier supérieur à zéro, rien n'interdit d'envoyer à la factorielle un objet d'un autre type. Cette liberté est à rapprocher de celle de Common Lisp [Steele jr 90].

Le calcul de la factorielle s'effectue de la manière suivante: si le nombre reçu est inférieur à 2, sa factorielle est 1. Ce résultat est envoyé à la continuation du calcul à l'aide de l'instruction **result** qui équivaut à un envoi asynchrone de la liste de ses arguments à l'adresse contenue dans la variable *forward-to*. Si le nombre reçu est supérieur ou égal à 2, la factorielle s'envoie un message demandant le calcul de $(n-1)!$ en spécifiant comme continuation un acteur chargé d'effectuer le produit du résultat du calcul précédent par n et de renvoyer ce résultat à la continuation courante. La continuation construite ici à l'aide de **continuation** regroupe deux choses: la description d'un comportement ainsi que la création d'un acteur. Son premier argument est une liste de déclarations dont la forme est comparable à celle d'un **let** de Common Lisp. Elle est utilisée ici afin de renommer la variable *forward-to* de la factorielle pour éviter qu'elle soit écrasée par la variable portant le même nom dans la continuation.

En faisant abstraction des envois de messages, cette factorielle présente une forte similitude avec une factorielle Lisp écrite en récursivité terminale. Elle n'utilise pas directement le parallélisme du langage. Le calcul d'une factorielle s'effectue de manière séquentielle bien que plusieurs factorielles puissent être calculées en parallèle.

7.2 Le modèle de réflexivité LActE/R

Le modèle de réflexivité adopté par LActE/R résulte de l'étude approfondie de nombreux langages munis de capacités réflexives, mais aussi de considérations propres à LActE/R.

Le modèle de réflexivité de LActE/R reflète les spécificités de LActE. Construit autour de LActE, avec qui il assure une compatibilité ascendante, LActE/R a d'abord été conçu afin de pallier aux limitations de la couche originelle. Plusieurs travaux ([Michel 90], [Michel 92] et [Michel 93]) ont montré que l'approche minimaliste de LActE présente des faiblesses lors de son utilisation extensive. La nécessité de rendre la couche acteur extensible a conduit à y introduire des capacités réflexives.

LActE/R reprend l'ensemble des caractéristiques de LActE, y compris la compilation des méthodes. L'ensemble des instructions définies dans LActE se retrouvent dans LActE/R et le modèle de calcul est préservé lors du passage à la réflexivité.

7.2.1 Genèse de LActE/R

La réflexivité dans une couche acteurs

LActE est conçu comme une couche au dessus de Common Lisp [Steele jr 90] qui augmente le langage de base de constructions et de mécanismes acteurs. Au sein d'un programme LActE coexistent des constructions acteur avec des constructions issues de Common Lisp.

L'introduction de la réflexivité dans un tel système ne peut se faire qu'en se limitant aux mécanismes acteur introduits par la couche LActE. En effet, l'extension de la réflexivité à Common Lisp supposerait soit la totale maîtrise d'une implémentation de ce langage afin de pouvoir la modifier dans le but d'y introduire les mécanismes nécessaires, soit l'ajout d'un interprète Lisp écrit en langage d'acteurs. Dans les deux cas, l'approche serait trop lourde et ne correspondrait pas aux objectifs que nous nous sommes fixés.

D'autre part, l'intérêt de la réflexivité dans LActE réside principalement dans la possibilité de modifier les mécanismes acteur, la manipulation des aspects liés à Common Lisp restant moindre. La réflexivité porte donc, dans le modèle de LActE/R, uniquement sur les aspects introduits par la couche acteur.

Modèle, implémentation et efficacité

La plupart des études sur la réflexivité portent essentiellement sur le modèle, aboutissant ainsi à des systèmes souvent inefficaces. Notre approche se veut plus pragmatique, et fait donc de l'efficacité du système résultant un de ses objectifs principaux, mais ceci, tout en préservant ses avantages, en particulier, son aspect dynamique. Ainsi, LActE/R conserve de LActE la compilation des méthodes tout en introduisant les mécanismes qui rendent possible la réification des instructions introduites par la couche acteur.

De par sa nature, l'utilisation de la réflexivité a un coût non négligeable. L'explicitation, et donc la manipulation, du langage passe par les outils linguistiques définis dans le langage. L'efficacité de telles constructions, tant au niveau de l'exécution que de l'occupation mémoire, est moindre que celle résultant d'une utilisation directe du langage ayant servi à l'implémentation.

Afin de minimiser le coût de l'utilisation de la réflexivité, le modèle défini suit deux principes: le principe de localisation spatiale, et le principe de localisation temporelle. Ces deux principes visent à minimiser l'impact de la réflexivité sur les performances. Ils servent de principes fondateurs au modèle de réflexivité de LActE/R.

Le *principe de localisation spatiale* consiste à localiser autant que possible les opérations réflexives aux entités qui y font appel. Le respect d'une telle localisation a pour effet de minimiser l'impact de la réflexivité sur les entités qui n'y font pas appel. En effet, si l'implémentation est telle que les portions de code qui ne font pas appel aux capacités réflexives du langage s'exécutent à une vitesse comparable à celle de la version non réflexive, le langage gagne déjà en efficacité. Pour conserver l'intérêt d'une telle approche, il est nécessaire que l'utilisation de la réflexivité ne se traduise pas par une dégradation des performances globales du système. Pour ce faire, il convient de localiser autant que possible les effets de l'utilisation de la réflexivité.

Le *principe de localisation temporelle* consiste à maîtriser dans le temps l'utilisation des capacités réflexives. En effet, l'utilisation de telles capacités est parfois très éphémère. Par exemple, lors d'une phase de mise au point, l'utilisation d'un dévermineur à des fins de localisation de l'origine d'une erreur peut rester localisée aux acteurs susceptibles d'en être à l'origine. De plus cette mise en place se faire dynamiquement, sur demande de l'utilisateur. En dotant le langage d'opérations qui permettent à l'utilisateur de mettre en place et de retirer les constructions réflexives, LActE/R autorise un contrôle précis de l'utilisation dans le temps des constructions réflexives.

Le modèle de réflexivité de LActE/R est basé sur le respect de ces principes en offrant la possibilité à l'utilisateur de maîtriser l'impact de la réflexivité sur l'ensemble du système. Il est conçu afin de permettre d'opérer de façon aussi locale que possible.

Méta-stabilité du système

L'utilisation de la réflexivité pose un problème dés lors que l'on cherche à conserver au système sa stabilité malgré les interventions de l'utilisateur. Un langage réflexif, tel que le définit P. Maes [Maes 87], présente la caractéristique d'être entièrement redéfinissable, que ce soit de façon locale, i.e. par modification d'une des entités de l'utilisateur, ou de façon globale, par modification d'une entité du système (ce qui, si la connexion causale est correctement réalisée, a pour conséquence de modifier le comportement global du système).

Si permettre la modification de l'intégralité des aspects du langage présente un intérêt au niveau du modèle, elle reste secondaire d'un point de vue pratique. En effet la plupart des programmes qui font appel à la réflexivité procèdent par ajout de capacités et non par une modification globale du fonctionnement intrinsèque du langage. D'autre part, permettre ces modifications mets en jeu la méta-stabilité du système, pouvant conduire à des programmes qui mettent le système hors fonction.

Notre modèle ne permet donc pas la modification globale du système. Ceci permet de garantir la stabilité du système, la programmation de la réflexivité procédant par ajout. LActE/R ne peut donc pas être qualifié, au sens de P. Maes [Maes 87], de langage réflexif, mais plutôt, de langage muni de capacités réflexives.

7.2.2 Éléments du modèle de réflexivité

Les principaux éléments de la réflexivité dans LActE/R sont:

- Les *méta-acteurs* qui réifient les mécanismes acteurs de leur dénotation.
- Une manipulation *dynamique* des tours de méta-acteurs basée sur deux opérations explicites, la *réification* et la *réflexion.*
- Une *uniformité linguistique* poussée qui donne aux objets de la couche LActE un statut d'acteur et laisse les acteurs jouer le rôle des objets de la couche.
- Les *méta-acteurs spécialisés* qui réifient les mécanismes propres aux objets de la couche LActE autres que les acteurs.
- Le triplet (*constructeur d'instructions*, *réificateurs*, *méta-instructions*) qui autorise la définition et la redéfinition des instructions ainsi que la réification de l'interprétation de chaque instruction vers une méta-instruction dédiée.

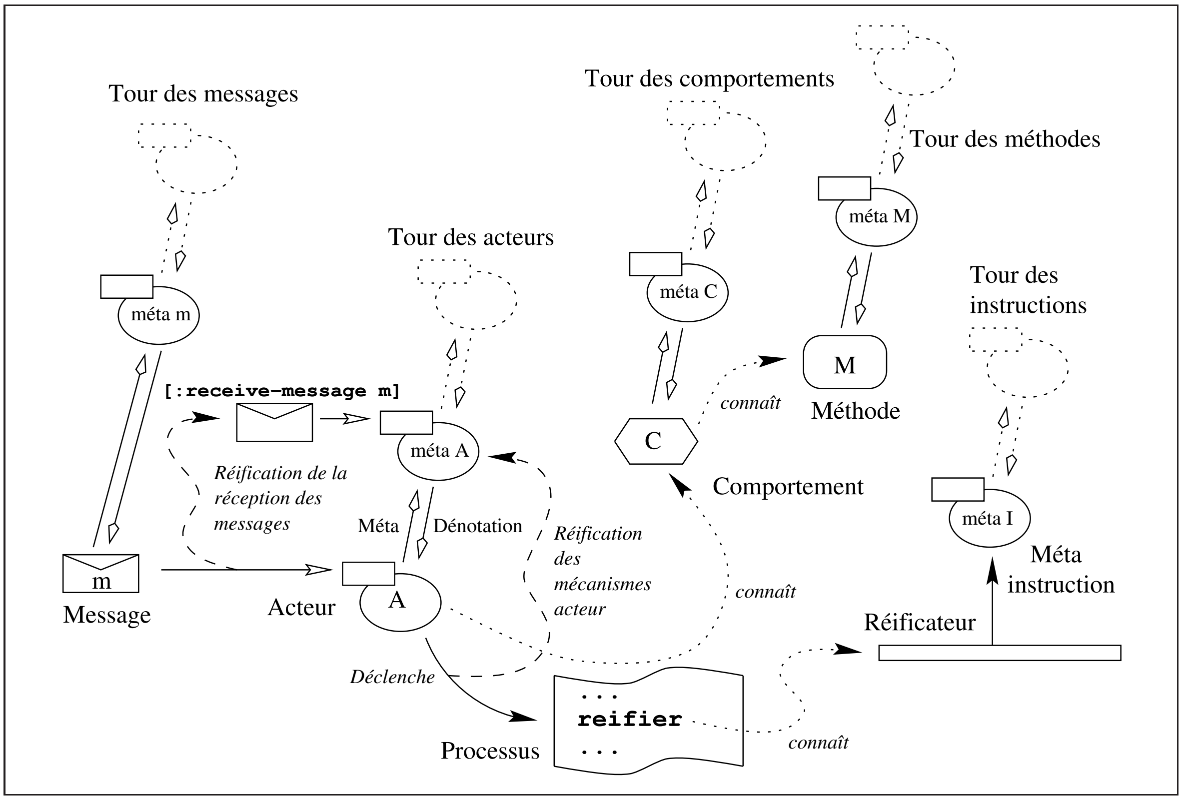

FIG. 7.3 – *Éléments de la réflexivité dans LActE/R*

Les éléments du modèle de réflexivité de LActE/R se retrouvent dans la figure (Fig. 7.3 page suivante).

Chaque objet de la couche LActE est susceptible de servir de base à une tour de méta-acteurs. Les tours de méta-acteurs diffèrent alors par la nature de l'objet qui en constitue la base et la nature du méta-acteur qui constitue le premier niveau. Les autres niveaux de la tour sont toujours constitués de méta-acteurs qui réifient un acteur (lui-même méta-acteur).

La réification des objets de la couche LActE s'opère vers des méta-acteurs spécialisés. L'utilisation de méta-acteurs spécialisés permet de décrire les mécanismes spécifiques à l'objet réifié. Par exemple, un message sera réifié vers un méta-message i.e. un acteur qui décrit les mécanismes d'un message.

Le dialogue entre deux éléments de la tour passe, dans LActE/R, par la dénotation. Contrairement à ABCL/R [Watanabe 88] qui traduit l'envoi d'un message d'une dénotation à une autre par l'envoi d'un message :receive du méta-objet de l'émetteur au méta-objet du destinataire, LActE/R autorise un dialogue direct entre les dénotations. Ce n'est qu'à la réception d'un message par une dénotation que le message est réifié en une activité de réception de message pour son méta-acteur. Ce mécanisme qui ne présuppose pas de l'existence d'un méta-acteur du destinataire du message autorise l'implémentation directe d'un acteur non réifié par l'interprète LActE/R. Il se traduit par un gain de performance notable.

La réification d'un acteur vers un méta-acteur passe par une opération explicite et dynamique. Cette réification peut s'opérer en cours d'exécution. Ce principe s'applique à l'opération inverse, la réflexion. L'utilisateur dispose ainsi d'un contrôle fin de l'utilisation dans le temps des méta-acteurs ainsi que du nombre de niveaux qui composent une tour de méta-acteurs. Ces opérations illustrent le principe de localisation temporelle.

LActE/R ne dispose pas à proprement parler d'un évaluateur des expressions du langage. Seules les instructions introduites par la couche LActE sont réifiables vers des méta-instructions. Ces dernières sont des acteurs qui se comportent comme des interprètes locaux des instructions du langage. Les réificateurs, dont un est attaché à l'évaluation de chaque couple (message, méthode), offrent une description synthétique des instructions. Chaque réificateur spécifie quelles sont les instructions de LActE/R qu'il réifie et vers quelle méta-instruction chaque instruction est réifiée.

Le constructeur d'instructions de LActE/R offre la possibilité de modifier les instructions préexistantes et d'en introduire de nouvelles. Il permet de spécifier le format d'une instruction, son implémentation ainsi que la manière dont une instruction doit être réifiée.

7.3 Les méta-acteurs

Les méta-acteurs jouent un rôle essentiel dans la réflexivité de LActE/R. Leur importance est liée à l'objet du langage dont ils mettent les mécanismes à nu: l'acteur. La manipulation des mécanismes de l'acteur est un des enjeux primordiaux de la réflexivité dans LActE/R. Les autres outils linguistiques qu'introduit la réflexivité sont organisés autour de ce concept.

Un méta-acteur est lui-même un acteur. L'utilisation d'un acteur pour représenter les mécanismes acteurs n'est possible que parce que le pouvoir expressif des langages d'acteurs est suffisamment puissant. Avec ses accointances et ses méthodes, l'acteur est capable de représenter une liste de messages reçus ou encore une activité comme la réception des messages.

Un méta-acteur est l'interprète des mécanismes acteurs de sa dénotation. Il se substitue à l'interprète LActE dans l'interprétation des mécanismes acteurs de sa dénotation. C'est par l'utilisation de comportements appropriés d'un méta-acteur que l'utilisateur a la possibilité de modifier les mécanismes acteurs de la dénotation.

Lorsqu'un acteur dispose d'un méta-acteur, tous les messages qui lui sont envoyés sont déroutés vers le méta-acteur après réification vers un message de réception de messages. Pour chaque message destiné à sa dénotation, le méta-acteur reçoit donc un message de la forme [:receive−message m] où m est le message destiné à la dénotation (Fig. 7.4 page suivante). Cette réification de la réception[1] des messages est à la base de la connexion causale entre le méta-acteur et sa dénotation. C'est d'abord par elle que la prise en charge de la dénotation par son méta-acteur s'opère.

Par conséquent, tout méta-acteur dispose d'une méthode :receive−message qui traite la

1. De la *réception* car elle est décidée sur la base du destinataire et non de l'émetteur du message.

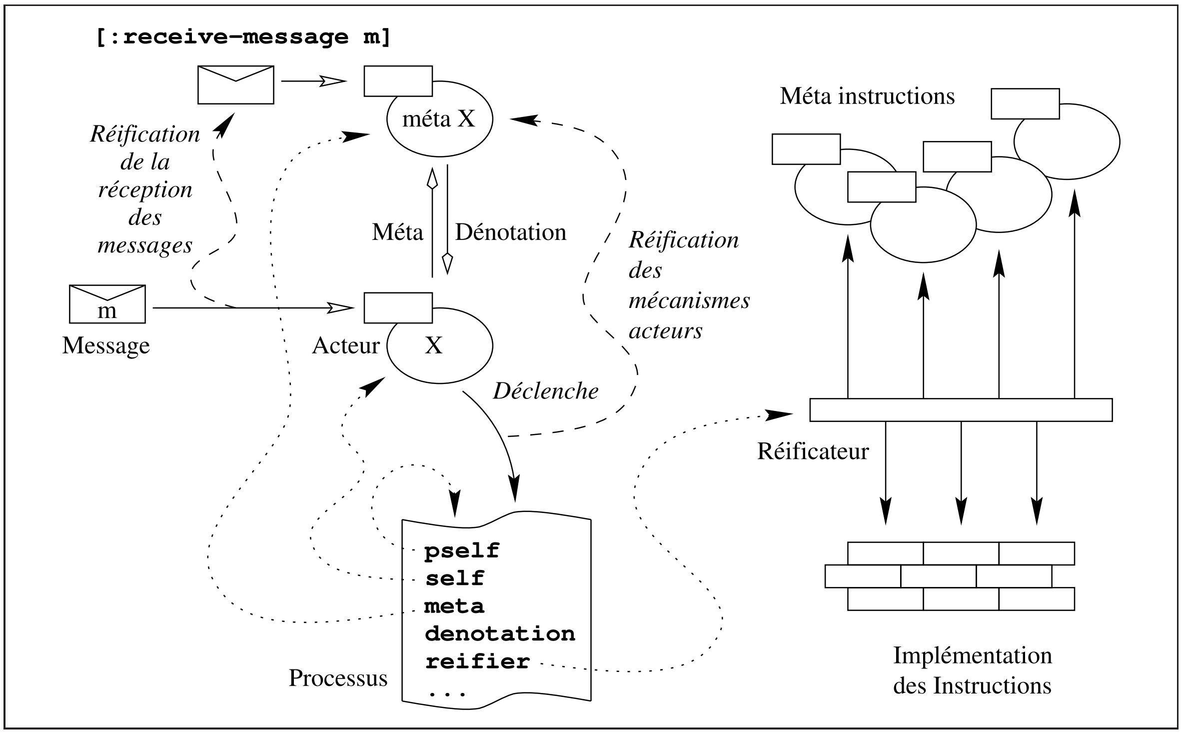

FIG. 7.4 – *Un méta-acteur et son environnement d'évaluation*

réception des messages de sa dénotation et à la suite de laquelle sera généré un processus support de l'évaluation de la méthode de la dénotation qui filtre le message. Les étapes qui mènent de la réception du message à la génération du processus appartiennent aux mécanismes contrôlés par le méta-acteur. Ces étapes, communes aux méta-acteurs d'acteurs sérialisés et d'acteurs non sérialisés, sont:

- La *réception des messages* qui consiste à recevoir un message et à décider si ce message doit être traité immédiatement ou en différé.
- L'*activation des messages* qui consiste à rechercher la méthode qui filtre le message reçu.
- L'*évaluation des méthodes* qui construit l'environnement d'évaluation de la méthode qui filtre le message et génère le processus support de l'activité de la méthode.
- La *gestion des messages non reconnus* qui traite les messages non reconnus en signalant une erreur ou en provoquant leur délégation.
- La *délégation des messages* qui délègue les messages i.e. les envoie au proxy de la dénotation.

Les prérogatives du méta-acteur s'arrêtent à la génération du processus support de l'activité de la méthode qui filtre le message reçu. Ce processus est indépendant du méta-acteur et de sa dénotation. Le méta-acteur n'exerce pas la fonction d'évaluateur des méthodes de sa dénotation. Cependant, il désigne le réificateur(cf. § 7.9.4 page 160) attaché au processus. Ce dernier, comme nous le verrons plus tard, est le moyen qu'offre LActE/R de réifier les instructions identifiées par la couche LActE.

7.3.1 Méta-acteur d'un acteur non sérialisé

Les acteurs non sérialisés se distinguent des acteurs sérialisés par leur incapacité à changer de comportement. Une fois créé, un acteur non sérialisé conserve le comportement qui lui est donné jusqu'à la fin de sa vie. De ce fait, son comportement est connu à tout instant, qu'il traite ou non un message. Les acteurs non sérialisés n'ont donc pas la nécessité d'attendre la spécification du prochain comportement pour traiter un nouveau message. Ils bénéficient ainsi d'un degré de concurrence plus important que celui des acteurs sérialisés.

Cette propriété se traduit au niveau du méta-acteur par l'utilisation d'un acteur non sérialisé pour représenter les mécanismes d'un acteur non sérialisé. Le méta-acteur se rapproche de sa dénotation dans son propre fonctionnement. Cette dernière bénéficie ainsi d'une machine acteur dotée d'un degré de concurrence important.

Le comportement *a-meta-actor-unserialized* définit le comportement standard du méta-acteur d'un acteur non sérialisé.

Le méta-acteur utilise cinq accointances pour représenter les différentes informations attachées à l'acteur. Ces informations correspondent respectivement au nom de l'acteur, à l'adresse de son comportement, à la liste des valeurs de ses accointances, à l'adresse de son proxy et à son réificateur.

```
; Méta-acteur d'un acteur non sérialisé
(def-behavior a-meta-actor-unserialized (actor-name actor-behavior
                                         actor-acquaintances actor-proxy
                                         actor-reifier) ()
```

Le code de la méthode qui implémente l'activité de réception des messages est:

```
  ; Réception des messages
 ((:receive-message a-message)
  (send [:activate-message a-message] :to self))
```

La méthode :receive−message recouvre l'activité de réception des messages. Les messages filtrés par cette méthode sont envoyés par le système lorsque la dénotation doit recevoir un message. La dénotation étant un acteur non sérialisé, le méta-acteur déclenche directement l'activité suivante, :activate−message.

Le code de la méthode qui implémente l'activation des messages est:

```
  ; Activation des messages
 ((:activate-message a-message)
  (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message
                        mesg-take mesg-type)
                       (ask [:get-values] :to a-message)                (i)
```

```
      (let ((a-actor-method
               (ask [:get-actor-method mesg-message :take mesg-take]
                    :to actor-behavior)))                                (ii)
        (if a-actor-method
            (send [:evaluate a-actor-method a-message] :to self)
            (send [:no-matching-actor-method a-message] :to self)))))
```

L'activation des messages correspond à la recherche de la méthode qui filtre le message passé en paramètre, *a-message*. La méthode commence par décomposer le message afin d'en extraire les informations utilisées pour effectuer la recherche (i). Ces informations correspondent au contenu du message ainsi qu'au rang de la méthode qui filtre le message.

La recherche du message n'est pas directement effectuée par le méta-acteur mais est déléguée au comportement de la dénotation (ii).

Selon qu'une méthode filtre ou non le message contenu dans *a-message*, la méthode déclenche l'activité suivante (:evaluate) ou l'activité de traitement des messages non reconnus par la dénotation (:delegate).

Le code de la méthode qui implémente l'activité d'évaluation des méthodes est:

```
 ; Évaluation des méthodes
((:evaluate a-actor-method a-message)
 (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message
                        mesg-take mesg-type)
                      (ask [:get-values] :to a-message)
    (let ((actor-ctxt
            (process-actor-ctxt mesg-to mesg-sender mesg-forward-to
                                self actor-proxy
                                mesg-type mesg-take a-actor-method a-message nil
                                actor-acquaintances actor-reifier)))
      (if (ask [:get-compiled?] :to a-actor-method)
          (evaluate-compiled-actor-method a-actor-method
            (process-compiled-actor-method-arguments actor-ctxt mesg-message))
          (evaluate-interpreted-actor-method a-actor-method
            (process-interpreted-actor-method-arguments
               actor-ctxt mesg-message actor-behavior))))))
```

L'évaluation des méthodes consiste à générer un processus support de l'activité d'une méthode après en avoir construit l'environnement d'évaluation. Pour assurer une portabilité minimale de LActE/R, la construction de l'environnement d'évaluation ainsi que du processus[2] support à l'évaluation d'une méthode passent par des fonctions dédiées. Notons que le protocole n'explicite pas la structure de l'environnement d'évaluation des méthodes.

L'environnement d'évaluation des méthodes est construit en deux temps: d'abord les informations en provenance de la dénotation puis celle en provenance du message reçu. L'environnement est légèrement différent selon que la méthode est compilée ou interprétée. Parmi les valeurs qui composent le contexte d'évaluation d'une méthode se trouve le réificateur

2. Les processus utilisés sont ceux de Common Lisp. Chaque implémentation de Common Lisp qui dispose de processus, ou plus précisément, de «threads», adopte une interface spécifique.

(*actor-reifier*). Il n'y a pas dans LActE/R d'interprète au sens d'ABCL/R [Masuhara 92]. Le réificateur est un objet particulier de LActE/R qui permet de réifier chacune des instructions du langage vers son propre interprète.

Les fonctions *evaluate-compiled-actor-method* et *evaluate-interpreted-actor-method* créent le processus support de, respectivement, l'exécution ou l'évaluation d'une méthode. Dans LActE/R, chaque couple (message, méthode) dispose de son propre processus.

Le traitement des messages non reconnus est implémenté par la méthode suivante:

```
 ; Traitement des messages non reconnus
((:no-matching-actor-method a-message)
 (if actor-proxy
    (send [:delegate a-message] :to self)
    (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message)
                         (ask [:get-values] :to a-message)
       (declare (ignore mesg-forward-to))
       (send [:error (format nil "Unrecognize message [~{~a~}] in ~a from ~a: ~a"
                     mesg-message (name-of mesg-to)
                     (name-of mesg-sender) a-message)]
             :to error))))
```

Si la dénotation dispose d'un proxy, le message non reconnu est délégué à ce proxy. Dans le cas contraire, une erreur est signalée.

La méthode suivante prend en charge l'activité de délégation des messages non reconnus:

```
 ; Délégation de messages
((:delegate a-message)
 (ask [:set-to actor-proxy] :to a-message)
 (send-message a-message)))
```

L'activité de délégation consiste à envoyer au proxy le message non reconnu. L'adresse du receveur du message passé en paramètre est modifiée, puis le message est envoyé à l'aide de la primitive *send-message*.

Le comportement qui vient d'être décrit est un comportement standard. Tout l'objet de la programmation réflexive consiste à modifier ce comportement afin d'obtenir des acteurs aux mécanismes particuliers. L'objet de la définition précédente est de servir de modèle et de faciliter la programmation réflexive en utilisant le comportement précédent à l'aide du mécanisme d'inclusion. Il reste bien sur possible de créer ex-nihilo le comportement du méta-acteur d'un acteur non sérialisé.

Tous les éléments de cette définition ne sont pas obligatoires. Pour les méthodes, seule la méthode de réception des messages est requise. Les autres méthodes sont le résultat d'un découpage arbitraire de la machine acteur, mais qui, nous le croyons, clarifie les mécanismes acteurs tels qu'ils sont conçus dans LActE/R. Les accointances ne sont exigées par le système que lors d'une opération de réification ou de réflexion. Entre ces deux opérations, le comportement du méta-acteur peut faire abstraction de la position des différentes accointances ainsi que de leur présence.

Un méta-acteur d'acteur non sérialisé

L'écriture d'un méta-acteur d'un acteur non sérialisé est relativement simple dès que l'inclusion(cf. § 7.6.1 page 144) de comportement est utilisée. Cette facilité linguistique rationalise et simplifie l'écriture des méta-acteurs.

Prenons pour exemple une méta-acteur dont le rôle est de tracer les messages reçus par sa dénotation. Ce méta-acteur se différencie de la définition précédente par une méthode :receive−message qui provoque l'affichage du contenu du message en plus de son activation. En utilisant l'inclusion de comportement, le comportement de ce méta-acteur s'écrit:

```
(def-behavior a-mesg-tracer() ()
  (declare (is-a a-meta-actor-unserialized))          (i)
  ((:receive−message message)
   (send [:get−message] :to message :forward−to screen)
   (send [:activate−message message] :to self)))
```

L'utilisation de l'inclusion de comportement réduit à quelques lignes l'écriture d'un méta-acteur qui trace les messages reçus par sa dénotation. L'expression (i) provoque l'inclusion du comportement *a-meta-unserialized-actor* au sein de notre comportement. L'unique méthode à écrire est :receive−message. Cette méthode envoi le contenu du message à l'écran et provoque l'activation du message.

7.3.2 Méta-acteur d'un acteur sérialisé

Les acteurs sérialisés ont la capacité de changer de comportement à chaque exécution de l'une de leur méthode. Ce n'est qu'une fois le prochain comportement de l'acteur spécifié qu'il lui est possible de traiter le message suivant. La granularité du parallélisme est donc contrôlée par les instructions qui spécifient le comportement suivant de l'acteur.

Lors d'un changement de comportement, un acteur spécifie le comportement qu'il va adopter, l'adresse de son proxy ainsi que les valeurs de ses accointances. Pour pouvoir modéliser ces changements de comportement, le méta-acteur d'un acteur sérialisé est lui-même sérialisé.

Le comportement du méta-acteur d'un acteur sérialisé est plus complexe que celui du méta-acteur d'un acteur non sérialisé. Les cinq méthodes principales s'y retrouvent plus ou moins modifiées. À ces méthodes s'ajoutent trois méthodes supplémentaires qui gèrent les changements de comportement de la dénotation. Ces méthodes sont associées directement aux instructions qui spécifient ces changements de comportement. Elles correspondent respectivement:

- au **become** sans paramètre.
- au **become** avec paramètres.
- à l'instruction **update**.

Elles seront appelées à chaque changement de comportement que les instructions correspondantes soient réifiées ou non. Le méta-acteur a ainsi le contrôle sur les changements de comportement de sa dénotation. LActE/R considère donc que les changements de comportement appartiennent, pour partie tout au moins, aux mécanismes acteurs.

Le comportement *a-meta-actor-serialized* est le comportement standard du méta-acteur d'un acteur sérialisé. Il diffère du comportement précédent par son statut de comportement d'acteur sérialisé, par le traitement des messages reçus ainsi que par ses trois méthodes dédiées au changement de comportement. En plus des cinq accointances communes avec le comportement du méta-acteur d'un acteur non sérialisé, apparaissent deux accointances supplémentaires : *actor-messages* et *actor-active*. La première de ces accointances contient la liste des messages reçus par la dénotation qui sont en attente de traitement. La seconde accointance est un booléen à *t* lorsqu'une méthode qui n'a pas encore spécifié le prochain comportement de la dénotation est en cours d'exécution. Il coordonne les divers mécanismes en jeu, en particulier la réception des messages avec leur activation ou leur mise en attente.

```
; Comportement du méta-acteur d'un acteur sérialisé
(def-behavior a-meta-actor-serialized (actor-name actor-behavior
                                        actor-acquaintances
                                        actor-proxy actor-reifier
                                        actor-messages actor-active) t
```

L'activité de réception des messages est pour les acteur sérialisés très différente des acteurs non sérialisés. Elle décide du traitement d'un message reçu selon l'état de la machine acteur. L'examen du booléen *actor-active* lui permet de savoir si elle peut directement lancer l'activation du message reçu ou si ce message doit être mis en attente. Dans ce dernier cas, le message est mis à la fin de la liste des messages en attente de traitement.

```
  ; Réception des messages
 ((:receive-message a-message)
  (cond (actor-active
          (setf actor-messages (append actor-messages (list a-message)))
          (update))
        (t (setf actor-active t)
          (update)
          (send [:activate-message a-message] :to self))))
```

La méthode :become avec paramètres est appelée lorsqu'une instruction **become** avec paramètres est utilisée au sein d'une méthode. Outre la modification des divers paramètres qui caractérisent l'état d'un acteur, elle permet de transformer un acteur sérialisé en acteur non sérialisé. Lorsqu'une telle modification survient, le méta-acteur vide la liste des messages en attente de traitement en provoquant l'activation de chacun des messages contenus dans cette liste. Il se transforme ensuite en un méta-acteur d'acteur non sérialisé. Notons que cette transformation n'est pas nécessaire au fonctionnement d'un acteur non sérialisé. Ces derniers sont en effet construits pour fonctionner aussi bien en mode non sérialisé que sérialisé. Mais si la transformation du méta-acteur n'est pas effectuée alors la dénotation perdra un peu de sa capacité à traiter plusieurs messages en même temps.

```
 ; Changement de comportement avec spécification d'un nouveau comportement
((:become a-behavior a-proxy some-acquaintances)
 (unless (eq a-behavior :self)
  (setf actor-behavior a-behavior))
 (setf actor-proxy a-proxy)
 (setf actor-acquaintances
       (append some-acquaintances
               (nthcdr (length some-acquaintances) actor-acquaintances)))
 (cond ((ask [:get−serialized] :to behavior)
        (send [:become] :to self :forward−to forward-to)
        (update))
       (t (become a-meta-actor-unserialized proxy
                  actor-name actor-behavior actor-acquaintances
                  actor-proxy actor-reifier)
          (mapcar #'(lambda (mesg)
                      (send [:receive−message mesg] :to self))
                  actor-messages))))
```

La méthode :become sans paramètre est appelée soit lors de l'utilisation d'une instruction **become** sans paramètre au sein d'une méthode, soit par l'une des deux autres méthodes de changement de comportement du méta-acteur. Elle examine la liste des messages en attente de traitement. Lorsque cette liste n'est pas vide, elle extrait le premier message de la liste qu'elle active. Si la liste est vide, elle modifie la valeur de l'accointance *actor-active* afin que le prochain message reçu par la dénotation soit directement traité. L'expression (**result**) qui termine la méthode correspond au résultat retourné à l'instruction de changement de comportement qui a déclenché la méthode.

```
 ; Changement de comportement sans spécification de nouveau comportement
((:become)
 (if actor-messages
     (let ((a-message (first actor-messages)))
       (setf actor-messages (rest actor-messages))
       (send [:activate−message a-message] :to self))
     (setf actor-active nil))
 (update)
 (result t))
```

La méthode :update est la dernière des trois méthodes de changement de comportement. Elle ne permet que la modification des valeurs des accointances de la dénotation. Après mise à jour de la liste des valeurs des accointances, elle appelle la méthode **become** sans paramètre qui achèvera d'effectuer le changement de comportement de la dénotation.

```
 ; Changement de comportement avec modification des accointances
((:update some-acquaintances)
 (setf actor-acquaintances
       (append some-acquaintances
               (nthcdr (length some-acquaintances) actor-acquaintances)))
 (send [:become] :to self :forward−to forward-to)
```

```
    (update))
```

L'évaluation des méthodes suit un schéma identique à celui des acteurs non sérialisés. À l'exception de l'instruction **become** du début de la méthode, et de certaines des valeurs passées aux différentes fonctions, le code reste ligne à ligne identique à celui des acteurs non sérialisés.

```
  ; Évaluation des méthodes
((:evaluate a-actor-method a-message)
 (become)
 (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message
                       mesg-take mesg-type)
                      (ask [:get-values] :to a-message)
     (let ((actor-ctxt (process-actor-ctxt mesg-to mesg-sender mesg-forward-to
                                           self actor-proxy
                                           mesg-type mesg-take
                                           a-actor-method a-message t
                                           actor-acquaintances actor-reifier)))
       (if (ask [:get-compiled?] :to a-actor-method)
           (evaluate-compiled-actor-method a-actor-method
              (process-compiled-actor-method-arguments actor-ctxt mesg-message))
           (evaluate-interpreted-actor-method a-actor-method
              (process-interpreted-actor-method-arguments
                  actor-ctxt mesg-message actor-behavior))))))
```

L'activation des messages est représentée par la méthode suivante identique au changement de comportement près à celle des acteurs non sérialisés.

```
  ; Activation des messages
((:activate-message a-message)
 (become)
 (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message
                       mesg-take mesg-type)
                      (ask [:get-values] :to a-message)
     (let ((a-actor-method
              (ask [:get-actor-method mesg-message :take mesg-take]
                   :to actor-behavior)))
         (if a-actor-method
            (send [:evaluate a-actor-method a-message] :to self)
            (send [:no-matching-actor-method a-message] :to self)))))
```

La méthode de traitement des messages non reconnus est, elle aussi, presque identique à celle des acteurs non sérialisés. Cependant, en plus de l'instruction **become**, elle doit, en l'absence de proxy, appeler la méthode **become** sans paramètre afin de provoquer le traitement des éventuels messages mis en attente.

```
  ; Traitement des messages non reconnus
((:no-matching-actor-method a-message)
 (become)
 (if actor-proxy
    (send [:delegate a-message] :to self)
```

```
      (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message)
                           (ask [:get-values] :to a-message)
        (declare (ignore mesg-forward-to))
        (ask [:become] :to self)
        (send [:error (format nil
                        "Unrecognize message [~{~a~}] in ~a from ~a: ~a"
                        mesg-message (name-of mesg-to) (name-of mesg-sender)
                        a-message)]
              :to error))))
```

La méthode :delegate reprend de la méthode correspondante pour les acteurs non sérialisés avec un appel systématique de la méthode **become** sans paramètre pour provoquer le traitement de messages mis en attente.

```
  ; Délégation de messages
 ((:delegate a-message)
  (become)
  (ask [:become] :to self)
  (ask [:set-to actor-proxy] :to a-message)
  (send-message a-message)))
```

Le comportement *a-meta-actor-serialized* définit le comportement standard du méta-acteur d'un acteur sérialisé. L'objet de la programmation réflexive consiste, comme pour les acteurs non sérialisés, à modifier ce comportement afin d'obtenir des acteurs aux mécanismes particuliers. Cependant, les exigences du système vis à vis des méta-acteurs d'acteurs sérialisés sont plus fortes que pour les acteurs non sérialisés. En effet, le méta-acteur doit être doté non seulement d'une méthode de réception des messages, mais aussi des méthodes de changement de comportement. En l'absence de ces méthodes, les instructions correspondantes ne peuvent correctement fonctionner.

Un méta-acteur d'acteur sérialisé

L'exemple de comportement de méta-acteur d'acteur sérialisé présenté ici est celui d'un compteur des messages reçus par sa dénotation.

```
(def-behavior a-mesg-counter(count) t
   (declare (is-a a-meta-actor-serialized))
   ((:get-count) (become) (result count))
   ((:receive-message message)
    (incf count)
    (update)
    (send [:receive-message message] :to self :take (next-method))))
```

L'accointance *count* est directement manipulée par l'utilisateur. Elle représente le nombre de messages reçus par la dénotation du méta-acteur. La méthode :get−count autorise l'accès en lecture à la valeur de *count*. Le comptage des messages est effectué par la méthode :receive−message. Elle incrémente l'accointance *count* avant de mettre à jour les valeurs des accointances (instruction **update**) et d'appeler la méthode :receive−message qui provient de l'inclusion du comportement *a-meta-actor-serialized*. Le mécanisme utilisé est basé sur le rang de la méthode calculé à l'aide de la macro *next-method*.

7.4 Les tours de LActE/R

Tout acteur de LActE/R peut être doté d'un méta-acteur qui prend en charge l'interprétation de ses mécanismes. Un méta-acteur étant un acteur, il est susceptible d'avoir un méta-acteur. Un méta-méta-acteur, i.e. le méta-acteur d'un méta-acteur, est aussi un acteur et est donc, lui aussi, susceptible d'avoir sont propre méta-acteur. Ce processus qui peut être réitéré à l'infini donne naissance à une tour de méta-acteurs dont la base est un acteur qui ne possède pas de dénotation.

Chaque acteur peut donner naissance à sa propre tour de méta-acteurs. Contrairement aux langages procéduraux de la première partie, ce n'est pas une tour d'interprètes qui peut exister dans LActE/R mais de multiples tours de méta-acteurs. En fait, il peut exister dans le système autant de tours que d'acteurs sans dénotation.

Les tours de méta-acteurs sont «potentiellement» infinies. Elles sont potentiellement et non virtuellement infinies car construites explicitement par l'utilisateur. Un acteur de LActE/R, s'il est susceptible d'avoir un méta-acteur, ne dispose pas de méta-acteur implicite par défaut. Le problème de la régression infinie de la tour de méta-acteurs ne se pose pas dans LActE/R: le sommet d'une tour est toujours un acteur explicitement créé par l'utilisateur.

Dans LActE/R, la relation entre une méta-entité et sa dénotation s'établit explicitement après une opération de réification. Contrairement à la plupart des interprètes réflexifs [Kiczales 91], le méta d'une dénotation n'est pas créé à la création de cette dernière, mais plus tard, par une opération spécifique dite de réification. LActE/R dissocie la création d'un objet du langage de sa réification.

À l'opération de réification, LActE/R associe une opération symétrique dite de réflexion. Alors que la première opération ajoute un niveau à une tour de méta-acteurs, la seconde supprime un niveau en redonnant à un acteur réifié son statut d'acteur autonome directement pris en charge par le système.

Une tour de méta-acteurs change de taille au gré des opérations de réification et de réflexion qui lui sont appliquées. L'utilisateur exerce un contrôle complet sur l'instant où un acteur dispose d'un méta-acteur et sur l'instant où ce méta-acteur lui est retiré. Les opérations de réification et de réflexion offrent donc un moyen de maîtriser l'utilisation dans le temps des capacités réflexives de LActE/R. Elles sont donc la traduction dans LActE/R du principe de localité temporelle.

7.4.1 L'opération de réification

L'opération de réification permet de réifier un acteur vers un méta-acteur i.e. de créer le méta-acteur et d'établir la connexion causale entre le méta-acteur et sa dénotation. Après une telle opération, le méta-acteur se comporte comme l'interprète des mécanismes acteurs de sa dénotation.

À sa création, un objet du langage n'a donc pas de méta. L'envoi d'une requête [:meta] à un tel objet se traduit par une valeur de retour égale à *nil*. Lorsque cette valeur correspond à l'adresse d'un acteur, ce dernier est alors le méta-acteur du récepteur de la requête.

La réification d'un acteur est obtenue par l'envoi d'un message du méta, le message :reify−to, à l'acteur à réifier. Le format de ce message est:

| [:reify−to <*comportement*> <*proxy*> <*valeur*$_1$> ... <*valeur*$_n$>]

où

- <*comportement*> est le comportement du méta-acteur.
- <*proxy*> est le proxy du méta-acteur.
- <*valeur*$_1$> à <*valeur*$_n$> sont les valeurs des accointances du méta-acteur qui ne sont pas spécifiées par le système.

Par exemple, la réification de l'acteur *fact* vers un méta-acteur ayant pour comportement *a-message-tracer* s'obtiendra par l'expression suivante:

| (**send** [:reify−to *a-message-tracer* ()] :to *fact*)

Les paramètres du message qui déclenche l'opération de réification sont similaires à ceux des instructions **new** et **new-anonymous**. Cette similitude est à rapprocher du fait que la réification contient une opération de création d'acteur.

La réification est subordonnée à certaines conditions:

- L'acteur à réifier ne doit pas déjà posséder de méta-acteur. Pour modifier le méta-acteur d'une dénotation, il faut dans un premier temps appliquer une opération de réflexion avant d'installer le nouveau méta-acteur à l'aide d'une opération de réification.
- Les premières accointances définies dans le comportement du méta-acteur doivent correspondre à celles définies dans le protocole. Vers ces accointances sont réifiées les valeurs des champs de la structure qui représente l'acteur. Le seul critère qu'utilise LActE/R pour décider vers quelle accointance doit être réifié tel champ est la position de l'accointance dans la liste des noms des accointances définie par le comportement du méta-acteur.

La réification d'un acteur n'est pas toujours immédiate. Pour assurer la cohérence du système, celui-ci doit être dans un état précis au moment où l'acteur est réifié. En particulier, dans le cas d'un acteur sérialisé, le comportement de l'acteur doit être connu. Il est donc exclu de réifier un acteur sérialisé alors qu'il exécute une méthode qui n'a pas encore spécifié son comportement de remplacement.

L'opération de réification se découpe en plusieurs étapes:

- Le système s'assure d'abord que l'acteur à réifier est dans un état connu et stable. Cet état ne doit plus être sujet à modification durant la réification. En particulier, l'acteur ne peut plus recevoir de message et est, pour ainsi dire, retiré du système. Toutes les interactions avec les autres acteurs du système sont suspendues jusqu'à la fin de l'opération.

- Les champs de la structure qui représente l'acteur sont réifiés pour servir de valeur aux accointances du méta-acteur. Lorsque la représentation interne diffère de la représentation externe, une conversion est effectuée. À cette liste d'accointances, le système ajoute celles fournies par l'utilisateur en paramètre du message réifiant. Les accointances d'un méta-acteur sont donc partiellement fournies par le système, celles qui proviennent de la réification de la dénotation, et partiellement par l'utilisateur, celles qui se trouvent en fin de liste.

- Le méta-acteur est créé avec comme comportement le comportement fourni en paramètre du message réifiant, comme proxy le second paramètre du message réifiant et, comme accointances, la liste des accointances précédemment calculée. Notons que le méta-acteur créé est un acteur anonyme. Cet anonymat renforce le caractère éphémère des constructions réflexives dans LActE/R.

- La connexion causale entre le méta-acteur et sa dénotation est établie. En particulier, le système déroute désormais tout message envoyé à la dénotation vers le méta-acteur après réification vers un message de réception de message (i.e. si le message m est adressé à la dénotation, le méta-acteur recevra le message [:receive−message m]).

- L'acteur est alors rendu au système avec son méta-acteur. Il peut désormais recevoir des messages et interagir avec les autres acteurs du système.

Une fois l'opération de réification accomplie, le méta-acteur devient l'interprète des mécanismes acteurs de sa dénotation. Il se substitue effectivement au système et est causalement connecté à sa dénotation.

En dissociant la réification d'un acteur de sa propre création, LActE/R permet de différer la mise en place de constructions réflexives.

7.4.2 L'opération de réflexion

L'opération de réflexion est symétrique à l'opération de réification: elle supprime le méta-acteur d'une dénotation. À la suite d'une telle opération, les mécanismes acteurs de la dénotation sont directement pris en charge par l'interprète LActE/R.

La réflexion d'un acteur est obtenue par l'envoi du message [:reflect] à l'acteur à réfléchir. Cet acteur doit nécessairement être le méta-acteur d'un autre acteur.

La réflexion est, elle aussi, subordonnée à certaines conditions.

- Le méta-acteur doit posséder une dénotation. Un acteur simple, sans dénotation, ne peut donc être sujet à une réflexion.
- Les premières accointances du méta-acteur doivent correspondre par leur sémantique aux accointances d'un méta-acteur tel qu'il est défini dans le protocole.

Les étapes qui conduisent à la réflexion d'un méta-acteur sont:

- Le système attend que les états du méta-acteur et de sa dénotation soient parfaitement déterminés.
- Les premières accointances du méta-acteur sont réifiées vers les champs de la structure qui représente sa dénotation. De la même manière que la réification établit une correspondance entre les champs de la structure qui représente la dénotation et les accointances du méta-acteur, l'opération de réflexion établit une correspondance entre les accointances du méta-acteur et les champs de la structure qui représente sa dénotation. Par contre, les accointances additionnelles fournies par l'utilisateur ne sont pas réfléchies. Elles sont totalement ignorées par le processus de réflexion.
- La connexion causale entre le méta-acteur et sa dénotation est levée. Les messages émis vers la dénotation seront désormais directement reçus par cette dénotation.
- L'acteur dénotation est rendu au système.

À la suite d'une réification, l'acteur dénotation reprend son statut d'acteur autonome directement pris en charge par le système. L'efficacité de l'exécution d'un tel acteur est identique à celle d'un acteur n'ayant pas été réifié.

7.4.3 D'une réification à une réflexion

Les contraintes imposées sur le comportement du méta-acteur lors d'une opération de réification ou réflexion sont assez fortes. Elles limitent un peu ce qu'il est possible de faire avec un méta-acteur. Cependant ces contraintes n'ont de raison d'être que lors de l'application de l'une ou de l'autre de ces opérations. Entre une opération de réification et une opération de réflexion, le comportement d'un méta-acteur a toute liberté d'évoluer.

Par exemple, le méta-acteur d'un acteur non sérialisé peut s'écrire:

```
(def-behavior another-meta-actor-unserialized(actor-behavior actor-acquaintances) ()
  ((:receive-message a-message)
   (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg-message
                                  mesg-take mesg-type)
                        (ask [:get-values] :to a-message)
      (let ((a-method (ask [:get-method mesg-message :take mesg-take]
```

```
                    :to actor-behavior)))
      (if a-method
         (let ((actor-ctxt (process-actor-ctxt mesg-to mesg-sender mesg-forward-to
                                               self nil
                                               mesg-type mesg-take a-method
                                               a-message nil
                                               actor-acquaintances *default-reifier*)))
            (if (ask [:get-compiled?] :to a-method)
               (evaluate-compiled-method a-method
                  (process-compiled-method-arguments actor-ctxt mesg-message))
               (evaluate-interpreted-method a-method
                  (process-interpreted-method-arguments actor-ctxt
                                    mesg-message actor-behavior))))
      (send [:error (format nil "Unrecognize message [~{~a~}] in ~a from ~a: ~a"
                              mesg-message (name-of mesg-to)
                              (name-of mesg-sender) a-message)]
            :to error))))))
```

Cette version simplifiée du comportement du méta-acteur utilise un nombre restreint d'accointances, *actor-behavior* et *actor-acquaintances*, qui contiennent respectivement l'adresse du comportement de la dénotation et la liste des valeurs des accointances de la dénotation. Les autres accointances habituelles à un méta-acteur, le nom de l'acteur, l'adresse de son proxy ainsi que son réificateur, n'apparaissent pas. Le méta-acteur utilise des valeurs par défaut pour ces dernières.

Le comportement utilise une méthode unique, :receive−message. C'est pour un méta-acteur d'acteur non sérialisé l'unique méthode requise pour se conformer au protocole. L'ensemble des mécanismes acteurs sont regroupés au sein de cette méthode. Par contre, en l'absence de proxy, le méta-acteur suit une version simplifiée du traitement des messages non reconnus: il génère une erreur.

La réification d'un acteur vers un méta-acteur qui adopte ce comportement ne peut se faire directement. Une étape intermédiaire est requise.

7.5 Messages et méthodes du méta

Les opérations relatives à la réflexivité, comme les messages réifiant et réfléchissant de la section précédente, ont un statut spécifique. Elles sont toutes implémentées à l'aide de messages dits messages du méta[3]. Leur implémentation est réalisée à l'aide de méthodes, dites méthodes du méta[4], qui sont incluses dans la liste des méthodes de chaque comportement. Ces messages et méthodes sont dits «du méta» car ils définissent les opérations qui permettent de contrôler l'implémentation d'un acteur.

3. Par opposition aux méta-messages qui sont aux messages ce que le méta-acteur est à l'acteur.

4. Comme pour les méta-messages, les méta-méthodes sont des acteurs vers lesquels les méthodes sont réifiées.

Les messages et méthodes du méta sont:

- :reify−to qui réifie le destinataire du message vers un méta-acteur dont le comportement est un des paramètres du message.
- :reflect qui réfléchit le destinataire du message, un méta-acteur. La dénotation du destinataire reprend ainsi son statut d'acteur directement implémenté par le système.
- :meta qui retourne l'adresse du méta-acteur du destinataire s'il existe. Dans le cas contraire *nil* est retourné.
- :denotation qui retourne l'adresse de la dénotation du destinataire ou *nil* s'il n'a pas de dénotation.
- :set−reifier qui permet de modifier le réificateur courant du destinataire.

Tout acteur de LActE/R dispose de ces cinq méthodes sans lesquelles la manipulation des aspects réflexifs de LActE/R est impossible. Pour éviter que ces méthodes ne soient cachées par les méthodes de l'utilisateur, elles sont toujours placées en première position dans la liste des méthodes.

Au statut particulier de ces méthodes correspond une implémentation adaptée et cachée à l'utilisateur. Pour assurer le fonctionnement continu de la réflexivité, ces méthodes sont implémentées comme des méthodes compilées dont la forme interprétée n'est pas disponible pour l'utilisateur. Il peut néanmoins les retirer de la liste des méthodes et obtenir ainsi un acteur qui ne soit pas réifiable.

7.6 Définition et création de comportement:

La définition de comportement est un des actes essentiels de la programmation dans le modèle de calcul de LActE/R. Sans son comportement, un acteur ne peut être créé et aucun calcul ne peut avoir lieu. Contrôler la création d'un comportement, c'est donc contrôler les tout premiers instants du calcul.

L'introduction de la réflexivité dans LActE ne pouvait aller sans la possibilité pour l'utilisateur de modifier le processus de création des comportements. De plus, pour faciliter l'utilisation de la réflexivité un mécanisme d'inclusion de comportement a été ajouté.

7.6.1 Extension de la définition de comportement

Pour permettre de détourner la création d'un comportement du système vers un autre acteur défini par l'utilisateur et d'inclure un comportement dans un autre comportement, la définition de comportement de LActE a dû être étendue. Dans LActE/R, la définition de comportement observe la syntaxe suivante:

```
(def-behavior <nom> <accointances> <sérialisation>
    {<déclarations>}\textrm{?}
    <script>)
```

À l'exception des déclarations, optionnelles, la définition d'un comportement en LActE/R ne diffère pas de celle de LActE. La compatibilité ascendante des deux systèmes permet donc à LActE/R de traiter du code LActE.

Les déclarations observent la syntaxe suivante:

```
(declare (is-a <comportement>) (constructor <acteur>))
```

Le **is-a** correspond à une *inclusion de comportement* et le **constructor** désigne un *constructeur de comportements* différent du constructeur par défaut. La première de ces déclarations constitue une facilité syntaxique destinée à simplifier la tâche du programmeur. La seconde appartient au modèle de réflexivité pour lequel elle introduit la possibilité de modifier le protocole de création du comportement.

L'inclusion de comportement

L'inclusion de comportement a été introduite dans LActE/R afin de faciliter la programmation d'applications réflexives. La programmation d'un méta-acteur nécessite la spécification de la totalité des accointances et méthodes définies dans le protocole alors que la différence entre le méta-acteur programmé et le comportement par défaut d'un méta-acteur peut se réduire à une seule méthode.

Par exemple, la trace des messages reçus par un acteur utilise un méta-acteur dont seule la méthode :receive−message diffère du protocole. L'utilisation de l'inclusion de comportement réduit l'écriture de son comportement à la définition de cette méthode[5]:

```
; Un méta-acteur qui trace les messages reçus
(def-behavior a-trace-message() ()
  (declare (is-a a-meta-actor-unserialized))
  ((:receive-message mesg)
   ; Envoi de la trace à l'écran
   (send [:get-message] :to mesg :forward-to screen)
   ; Activation du message
   (send [:activate-message mesg] :to self)))
```

Outre la brièveté du code, l'inclusion de comportement apporte clarté et sécurité. Elle souligne clairement les différences entre le comportement défini et le comportement d'un méta-acteur classique. En minimisant la quantité de code à écrire, elle réduit les sources d'erreurs. En particulier, il n'est plus nécessaire de se souvenir de l'intégralité du protocole pour définir un méta-acteur.

5. L'exemple présenté est celui du méta-acteur d'un acteur non-sérialisé.

Mécanisme d'inclusion de comportement

L'inclusion de comportement fonctionne à la manière du :include du **defstruct** de Common Lisp. Par analogie avec l'héritage des langages à objets [Masini 89], il s'apparente à un héritage simple et statique. Un héritage est simple car l'inclusion n'opère que sur un seul et unique comportement et statique, car il ne s'agit pas d'établir un pointeur sur le comportement inclus mais d'en copier les informations.

L'ensemble des informations qui proviennent du comportement inclus[6] sont copiées dans le comportement incluant[7]. Après création du comportement incluant, plus aucun lien ne subsiste avec le comportement inclus.

Règles d'inclusion de comportement

Les règles d'inclusion de comportement définissent le rapport entre les éléments qui proviennent du comportement inclus et les éléments définis par le comportement incluant. Elles régissent la fusion des accointances, des booléens de sérialisation et des méthodes.

- *Règle d'inclusion des accointances:* Les accointances incluses précèdent les accointances déclarées dans le comportement incluant. Si le comportement inclus a pour accointances $(a_1^1 \ldots a_n^1)$ et le comportement incluant déclare $(a_1^2 \ldots a_m^2)$ alors le comportement incluant aura pour accointances $(a_1^1 \ldots a_n^1 a_1^2 \ldots a_m^2)$. Ce choix s'explique par les modifications des mécanismes de changement de comportement induites par l'inclusion de comportement(cf. § 7.6.1 page suivante).

- *Règle de fusion des booléen de sérialisation:* Si l'un des comportements, inclus ou incluant, est sérialisé alors le comportement résultant est sérialisé. En effet, un comportement sérialisé impose une contrainte plus forte qu'un comportement non sérialisé. Les méthodes issues d'un comportement sérialisé contiennent des spécifications de changement de comportement qui nécessitent des mécanismes de synchronisation que les acteurs non sérialisés ne mettent pas en jeu. Par ailleurs, les acteurs non sérialisés peuvent être vus comme un cas particulier des acteurs sérialisés dans lequel les méthodes débuteraient par une instruction **become**.

- *Règle d'inclusion des méthodes:* Les méthodes incluses sont ajoutées à la fin de la liste des méthodes du comportement incluant. Les méthodes incluses sont ainsi redéfinissables dans le comportement incluant, les méthodes de ce dernier cachant les méthodes du premier.

L'ensemble de ces règles contribue à rendre transparent et déterministe le mécanisme d'inclusion de comportement.

6. Le comportement passé en paramètre de **is-a**.
7. Le comportement en cours de création et qui utilise une déclaration **is-a**.

Changement de comportement et inclusion de comportement

L'inclusion de comportement soulève de nombreux problèmes quant à la sémantique du changement de comportement. Si le **become** sans paramètre fonctionne correctement que ce soit avec une méthode incluse ou non, le **become** avec paramètres ne continue pas à fonctionner correctement lorsqu'il conserve sa sémantique.

Un **become** avec paramètres spécifie le nom du comportement de remplacement, l'adresse du proxy ainsi que les valeurs des accointances. Ces valeurs, lorsqu'elles proviennent d'une méthode incluse, ne correspondent qu'aux accointances incluses. Les accointances définies par le comportement incluant n'ont alors pas de valeur.

L'observation d'un changement de comportement au sein d'une méthode incluse suggère que les accointances ajoutées par l'utilisateur conservent leurs valeurs. Les calculs qu'effectuent ces méthodes ne portent que sur les accointances incluses.

La sémantique du **become** est adaptée à l'inclusion de comportement. Les valeurs spécifiées modifient uniquement les premières accointances. Le reste des accointances conserve ses valeurs. Il devient ainsi possible, pour un comportement ayant $(a_1 \ldots a_n)$ comme accointances, de ne spécifier que les valeurs de $a_1 \ldots a_m$ avec $m<n$. Les accointances $a_{m+1} \ldots a_n$ conservent alors leurs valeurs lors du changement de comportement.

Cette sémantique du **become** avec paramètres est en parfait accord avec la règle d'inclusion des accointances(cf. § 7.6.1 page ci-contre).

Afin de faciliter la tâche du programmeur, une instruction supplémentaire de changement de comportement a été introduite, le **update**. Cette instruction sans paramètre modifie les valeurs des accointances à partir de leurs valeurs dans le contexte lexical courant. Le comportement courant de l'acteur ainsi que son proxy ne sont pas modifiés.

L'instruction **update** facilite l'utilisation de LActE/R.

Accès aux méthodes cachées

La règle d'inclusion des méthodes conduit dans certains cas à cacher des méthodes incluses. Si une méthode définie dans le comportement incluant utilise le même filtre de message que l'une des méthodes incluses, alors il devient impossible d'utiliser la méthode incluse.

À la manière des langages à objets, LActE/R dispose d'un mécanisme qui permet l'accès aux méthodes cachées. Ce mécanisme est basé sur le rang des méthodes qui filtrent un même message. La première de ces méthodes a, par convention, le rang 1, la seconde, le rang 2, et ainsi de suite jusqu'à la dernière.

Cette numérotation des méthodes est dynamique. Elle s'effectue lors de la recherche de la méthode qui filtre le message reçu.

Les trois instructions d'envoi de message sont étendues afin d'accepter un mot-clé supplémentaire, `:take`. Ce mot-clé doit être suivi du rang de la méthode qui filtre le message. Aussi, par défaut, sa valeur est 1, soit le rang de la première méthode qui filtre le message.

Dans un processus de spécialisation de méthode, l'appel de la méthode cachée par la méthode la spécialisant ne peut se faire directement par ce biais. L'utilisation du nombre 2 comme valeur de :take répondra aux attentes du programmeur tant que le comportement incluant ne sera pas utilisé comme comportement inclus. L'inclusion de ce comportement provoquerait une boucle infinie sur la méthode spécialisante.

Pour pallier à ce défaut, LActE/R utilise la macro *next-actor-method* qui calcule le rang de la méthode suivante à partir du rang de la méthode courante.

Bien que plus rudimentaire que le mécanisme de super-méthode des langages à objets, le système de rang de méthode mis en place dans LActE/R s'est révélé être suffisant dans la plupart des cas.

Anomalies dues à l'inclusion de comportement

L'inclusion de comportement, si elle facilite l'utilisation de la réflexivité, reste délicate à mettre en œuvre et demande toute l'attention du programmeur. Une utilisation inconsidérée de ce mécanisme conduit dans certains cas à des anomalies de fonctionnement.

La première de ces anomalies apparaît lors de l'inclusion d'un comportement sérialisé. Elle est liée à l'utilisation de l'instruction **become** avec paramètres, dont le premier paramètre, le comportement, selon sa valeur, provoque un changement de comportement inattendu.

Prenons pour exemple une cellule i.e. un acteur qui joue le rôle d'une variable globale. Son comportement est:

```
(def-behavior a-cell(value) t
        ((:get) (become) (result t))
        ((:set new-value)
         (become a-cell proxy new-value)
         (result t)))
```

La méthode :set de ce comportement contient une instruction **become** qui spécifie comme nouveau comportement *a-cell*. Lors de la réception d'une nouvelle valeur de *value*, l'acteur se transformera donc en un acteur de comportement *a-cell* dont la valeur de la première accointance sera *new-value*. C'est là le comportement attendu d'une cellule.

L'utilisation de l'inclusion de comportement pour construire une cellule qui mémorise sa valeur précédente conduit à une anomalie. Le comportement d'une telle cellule est:

```
(def-behavior a-mem-cell(previous-value) t
        ; Inclusion du comportement d'une cellule
        (declare (is-a a-cell))
        ; Modification de la méthode :set
        ((:set new-value)
         (become a-mem-cell proxy value value)
         (send [:set] :to self :forward−to forward-to :take (next-method)))
        ...)
```

La méthode :set est modifiée pour sauvegarder la valeur courante de *value* dans l'accointance *previous-value*. Cependant, elle appelle l'ancienne méthode qui, lorsqu'elle sera activée, effectuera un (**become** *a-cell* ...). L'acteur cellule mémorisant sa valeur précédente perdra alors cette capacité à mémoriser en devenant un simple acteur cellule !

Ce comportement n'est pas le comportement attendu. Dans ce cas précis, le comportement souhaité peut être obtenu de trois façons:

1. En réécrivant la méthode :set lors de l'inclusion au lieu de faire appel à la méthode incluse. Bien que l'approche soit un peu lourde, elle est acceptable dans ce cas précis. Mais, pour une méthode plus longue et plus complexe, les autres solutions lui sont préférables.

2. En utilisant l'instruction **update** au lieu du **become** dans la méthode :set de *a-cell*. La seule modification est le changement de la valeur de *value*. Le comportement reste donc identique. Après inclusion, le fonctionnement sera conservé i.e. une cellule qui mémorise sa valeur précédente restera une cellule qui mémorise sa valeur précédente.

3. La troisième solution s'apparente à la seconde. Elle consiste à utiliser le mot clé :self comme nom de comportement. Ce dernier désigne le comportement courant qui devient *a-mem-cell* dans le comportement incluant.

L'ensemble de ces solutions sont bien adaptées au cas présent: l'inclusion des méthodes d'un comportement sérialisé qui spécifient un changement de comportement qui ne modifie que le proxy ou les accointances. Elles restent inefficaces si le comportement est modifié. Prenons le cas de l'inclusion dans un comportement C_1^{inc} d'un comportement C_1 dont l'une des méthodes spécifie un nouveau comportement C_2. Sur l'ensemble des solutions décrites, les deux dernières sont inadaptées: elles ne conviennent que pour $C_1 = C_2$. La première solution peut encore servir ici. Elle reste lourde à utiliser et démontre que changement de comportement et inclusion de comportement coexistent difficilement.

La seconde de ces anomalies est liée au mécanisme de filtrage des messages. Prenons le cas d'un compteur dont le comportement est le suivant:

```
(def-behavior a-counter(value) t
        ((:inc) (incf value) (update) (result value))
        ((:get) (become) (result value)))
```

et construisons un compteur plus élaboré par inclusion de ce comportement:

```
(def-behavior another-counter() t
        (declare (is-a a-counter))
        ((:inc)
         (become)
         (send [value] :to screen)
         (send [:inc] :to self :forward−to forward-to
                 ; Calcul du rang de la méthode incluse
                 :take (next-method))
```

```
        ((value)
         (update)))
```

La dernière méthode autorise la modification de la valeur courante du compteur par l'envoi d'un message qui contient cette valeur. Le filtre de cette méthode accepte tous les messages de longueur 1, y compris [:inc]. Le rang de la méthode incluse :inc est ainsi modifié de 2 à 3 et le calcul effectué par (*next-method*) erroné. La méthode :inc du comportement incluant n'appellera donc pas la méthode :inc du comportement inclus mais la méthode *value* du comportement incluant.

L'unique solution à ce problème repose intégralement sur le programmeur. Pour appeler la méthode désirée, le calcul du rang de la méthode doit être modifié en (*1+* (*next-method*)).

La réorganisation des méthodes i.e. la déclaration de la méthode *value* avant la méthode :inc ne résout pas le problème. Elle ne fait qu'en soulever d'autres en déclenchant la méthode *value* en lieu et place de la méthode :inc lors de la réception d'un message [:inc].

Cette anomalie est issue de la conjonction de deux choix dans l'implémentation de l'interprète LActE/R:

1. Un algorithme de pattern matching laxiste qui présente aussi bien des avantages que des inconvénients. Le filtrage des messages de LActE/R peut, par exemple, être utilisé pour écrire une factorielle qui n'utilise pas le **if** de Lisp.

2. Un système de désignation des méthodes par leur rang qui semble un peu primitif pour son usage.

LActE/R n'est pas le seul langage à souffrir d'anomalies liées à l'utilisation de l'héritage. Les langages à objets concurrents sont connus pour les problèmes qu'ils soulèvent lors de l'utilisation de l'héritage sur des classes faisant appel à certains mécanismes de synchronisation [Briot 87, Matsuoka 90]. Des solutions plus ou moins performantes ont été proposées pour ces langages. L'adaptation d'une de ces solutions à LActE/R, ou le développement d'une solution propre reste une des tâches à accomplir pour notre interprète.

Il faut cependant noter que l'inclusion de comportement dans LActE/R n'a pas le même objectif que l'héritage des langages à objets concurrents. Elle pallie à la complexité de la programmation réflexive mais en conservant l'autonomie des comportements. La recopie des informations d'un comportement à un autre fait du mécanisme d'inclusion un mécanisme qui ne doit pas être utilisé à l'infini sous peine d'obtenir des comportements trop lourds en occupation mémoire. La connaissance de la chaîne d'inclusion par l'utilisateur est donc un minimum requis pour l'utilisation de ce mécanisme.

De ces anomalies, il résulte que, si l'inclusion facilite la mise en œuvre des capacités réflexives de LActE/R, il convient de l'utiliser avec circonspection.

7.6.2 Protocole de création des comportements

La déclaration d'un constructeur de comportements dans un **def-behavior** rend possible la modification du processus de création d'un comportement. En présence d'une telle déclaration, la commande **def-behavior** délègue la création du comportement à l'acteur dont l'adresse est spécifiée. Cet acteur, nommé *constructeur de comportements*, reçoit du système une requête qui lui demande la construction d'un comportement et à laquelle il doit répondre en retournant l'adresse du comportement créé.

Ce protocole ne laisse à la charge du constructeur de comportements que la création du comportement. Son nommage dans l'environnement global reste du domaine du système. La gestion de l'environnement n'est donc pas rendue explicite par la réflexivité.

Le constructeur reçoit un message qui a la forme suivante:

```
(:build-behavior <name> <acquaintances> <serialized> <declarations> <script>)
```

où

- <*name*> est le nom du comportement à créer.
- <*acquaintances*> est la liste des noms des accointances du comportement.
- <*serialized*> est la valeur du booléen de sérialisation évalué dans l'environnement lexical de déclaration du comportement.
- <*declarations*> est la liste des déclarations.
- <*script*> est la liste non évaluée des méthodes du comportement. Elle correspond au texte des méthodes.

Le protocole ne fixe que le format du message et la nature de la réponse.

Exemple de constructeur de comportements

Un constructeur de comportements est un acteur dont l'une des méthodes filtre le message :build−behavior. À l'aide des paramètres du message, il construit un comportement dont l'adresse est envoyée à la continuation du calcul (contenue dans *forward-to*). À l'instar des comportements de méta-acteurs, le comportement présenté ici n'est qu'un guide à la conception de constructeurs de comportements. Mais, contrairement à ces premiers, il n'est pas inclus dans le système. L'écriture d'un constructeur de comportements ne peut donc s'appuyer sur l'inclusion de comportement.

Notre constructeur de comportements est un acteur non sérialisé, la totalité des informations nécessaires à la construction d'un comportement étant transmise par le message. Il se divise en deux méthodes. La première construit le comportement alors que la seconde construit la liste des méthodes du comportement. Le comportement du constructeur de comportements est:

```
;; A behavior constructor example
(def-behavior a-constructor-example() ()
  ((:build-behavior name acquaintances serialized declarations script)
   (let* ((included-behavior (get-included-behavior declarations))
          (acquaintances (append (when included-behavior
                                       (ask [:get-acquaintances] :to included-behavior))
                                 acquaintances))
          (serialized (or serialized
                          (when included-behavior
                                (ask [:get-serialized] :to included-behavior))))
          (methods (append
                     (make-meta-methods serialized)
                     (ask [:make-methods serialized acquaintances script ()] :to self)
                     (when included-behavior
                           (remove-meta-methods
                             (ask [:get-methods] :to included-behavior))))))
     (result (new-anonymous a-behavior () name acquaintances serialized methods))))
  ((:make-methods serialized acquaintances script methods-list)
   (if script
     (send [:make-methods serialized acquaintances (rest script)
                          (append methods-list
                                  (list (new-anonymous a-compiled-method ()
                                                       serialized acquaintances
                                                       (first script))))]
           :to self :forward-to forward-to)
     (result methods-list))))
```

La méthode :build−behavior, appelée par le système lors d'un **def-behavior**, construit le comportement requis. Un comportement est caractérisé par son nom, la liste des noms de ses accointances, un booléen de sérialisation ainsi que la liste de ses méthodes. La première action de la méthode consiste à extraire des déclarations l'adresse d'un éventuel comportement inclus. Les valeurs associées à chaque champ d'une déclaration sont évaluées dans le contexte lexical de la déclaration avant d'être transmises au constructeur. Le résultat de la fonction *get-included-behavior* est donc soit l'adresse d'un comportement, soit la constante *nil*. Le calcul des accointances du comportement en cours de construction est conditionné par cette valeur.

La liste des accointances est ensuite calculée. Cette liste est soit la liste des accointances déclarées dans le **def-behavior**, soit le résultat de la concaténation de la liste des accointances du comportement inclus et des accointances déclarées. Les accointances du comportement inclus se trouvent donc systématiquement en début de liste. Le positionnement de ces accointances doit être respecté pour assurer le fonctionnement correct de l'instruction **become** avec paramètres qui apparaît dans les méthodes incluses.

Le booléen de sérialisation est calculé en suivant les règles d'inclusion de comportement. Sa valeur est *t* lorsqu'au moins l'un des comportements est sérialisé. Si aucun comportement n'est inclus, sa valeur est alors celle déclarée dans le **def-behavior**.

La liste des méthodes ne peut être construite qu'après le calcul des deux accointances précédentes. La compilation d'une méthode utilise ces deux informations. La liste des méthodes d'un comportement est le résultat de la concaténation de trois listes:

1. La liste des méthodes du méta constituée des méthodes qui traitent les messages du méta (les messages :reify–to, :reflect, :meta, :denotation etc. ...).

2. La liste des méthodes déclarées dans le **def-behavior**.

3. La liste des méthodes incluses privées de leurs méthodes du méta.

Cette construction laisse apparaître d'une part le traitement spécifique des méthodes du méta et, d'autre part, le positionnement en fin de liste des méthodes incluses. Les méthodes du méta sont mises en début de liste afin qu'aucune méthode, déclarée ou incluse, ne vienne les cacher.

Le calcul des différentes accointances du comportement effectué, ce dernier est construit à l'aide d'un **new-anonymous** (le système se chargera ensuite de le nommer dans l'environnement global) et son adresse retournée comme résultat de la requête initiale.

La seconde méthode, :make–methods, construit la liste des méthodes déclarées à partir du texte de leurs déclarations. Elle effectue un parcours récursif de la liste des déclarations de méthodes en construisant chaque méthode à l'aide d'un **new-anonymous**. Les méthodes construites sont des méthodes compilées, les méthodes utilisées par défaut dans LActE/R.

Le comportement *a-constructor-example* peut alors servir à la création d'un acteur constructeur de comportements et ce dernier déclaré comme constructeur de comportements d'un autre comportement. C'est ce que le code suivant fait:

```
(new constructor-example a-constructor-example ())
```

```
(def-behavior a-fact() ()
  (declare (constructor constructor-example))
  ((n) (if (< n 2)
           (result 1)
          (send [(1- n)] :to self :forward-to
                (continuation ((cont forward-to))
                              ((value) (send [(* value n)] :to cont)))))))
```

```
(new fact a-fact ())
```

L'acteur *fact* créé à partir du comportement *a-fact* lui-même construit par le constructeur *constructor-example* est en tout point identique à un acteur construit selon la procédure standard.

L'exemple de cette section ne démontre pas l'utilité des constructeurs de comportements. Il ne fait qu'en montrer le fonctionnement. Le chapitre sur les exemples d'utilisation de LActE/R fait lui appel à cette possibilité pour construire des acteurs aux comportements spécifiques.

7.7 L'uniformité linguistique

L'uniformité linguistique a été l'un des critères de conception du modèle de réflexivité de LActE/R. L'application de ce critère aux objets identifiés par la couche LActE a conduit à traiter les objets du langage comme des acteurs et à rendre possible l'utilisation d'acteurs comme objets du langage.

7.7.1 Vision «acteur» des objets du langage

L'ensemble des objets identifiés par la couche acteur peuvent être traités comme des acteurs: les instructions de création d'acteurs leurs sont applicables et ils peuvent recevoir et traiter des messages. Par exemple, l'émetteur d'un message est obtenu en envoyant à ce message la requête [:get−sender].

Bien que dotés de ces capacités d'acteurs, les objets du langage conservent une implémentation spécifique[8]. Implémentés comme des acteurs légers comparables aux objets légers d'ABCL/R2 [Masuhara 92], ils apparaissent comme étant des acteurs à l'utilisateur. En effet, vu de l'extérieur, un acteur ne se différencie d'un autre objet que par sa capacité à recevoir des messages et à les traiter. En dotant ces objets d'une boîte aux lettres tout en préservant leurs spécificités, LActE/R allie les avantages d'une uniformité linguistique étendue à l'ensemble des objets de la couche LActE avec une efficacité de l'exécution proche de l'interprète originel.

7.7.2 Des acteurs comme objets du langage

Si les objets du langage peuvent être vus dans LActE/R comme des acteurs, l'inverse est aussi vrai : des acteurs peuvent jouer le rôle d'objet du langage. Ainsi LActE/R autorise l'envoi d'un message qui soit un acteur créé par l'utilisateur pour autant que cet acteur respecte le protocole de LActE/R.

La capacité à utiliser des acteurs comme objet du langage introduit une certaine liberté dans la programmation réflexive. Par exemple, l'ajout d'une information supplémentaire dans un message est très simple à réaliser. Le comportement qui suit est celui d'un message auquel a été ajoutée une accointance qui contient la liste des acteurs ayant accédé aux autres accointances du message.

```
(def-behavior a-message-tracer(actors) t
   (declare (is-a a-message))
   ((:get−actors) (become) (result actors))
   (request
      (unless (member sender actors)
         (push sender actors))
      (update)
      (send request :to self :forward−to forward-to
```

8. L'implémentation des objets de la couche LActE est décrite dans le chapitre suivant.

```
                          :sender sender :take (next-method))))
```

L'utilisation d'un acteur en lieu et place d'un objet du langage ne relève pas de la réification. L'acteur qui joue un tel rôle n'est pas le méta d'une quelconque entité mais implémente directement un objet du langage. Ce niveau d'action est différent de l'utilisation de méta-acteurs mais complémentaire à ce dernier. Il étend les possibilités de la programmation réflexive en laissant le choix d'utiliser des métas ou des acteurs classiques pour modifier les objets du langage. Cependant, un objet du langage implémenté par un acteur conservera cette implémentation jusqu'à la fin de sa vie. Il n'est pas possible de lui appliquer une opération de réflexion pour le transformer en objet du langage.

7.8 Réification des autres objets de la couche LActE/R

La réflexivité touche, dans LActE/R, l'ensemble des objets de la couche LActE. Les messages, les comportements, les méthodes et les futures qui appartiennent tous à cette couche sont eux aussi réifiables.

La réification des objets du langage s'opère vers des méta-acteurs spécialisés. Les méta-acteurs spécialisés ne reflètent pas la nature «acteur» de ces objets du langage mais leurs mécanismes spécifiques. Ainsi un message n'est pas réifié vers un méta-acteur mais vers un méta-message. Ce dernier interprète les mécanismes propres au message et non les mécanismes «acteurs» du message.

Ce choix semble être en contradiction avec la nature «acteur» des objets du langage (cf. § 7.7.1 page précédente) et la possibilité d'utiliser des acteurs définis par l'utilisateur pour remplir la fonction de l'un de ces objets (cf. § 7.7.2 page ci-contre). Il introduit deux types de méta-acteurs pour une même fonction: un message créé par le système est réifié vers un méta-acteur spécialisé alors qu'un message créé par l'utilisateur est réifié vers un méta-acteur classique.

Pourtant, ce choix se révèle être justifié dès que la programmation réflexive est prise en compte en conjugaison avec une aptitude particulière à LActE/R, la capacité de réifier dynamiquement un objet du langage. Lorsque l'utilisateur de LActE/R désire un message au comportement particulier, il crée ce message par instanciation d'un comportement qu'il a défini. Ce message est alors un acteur qui remplit le rôle de message. Étant donné que l'utilisateur a défini le comportement de ce message, il s'attend, lorsqu'il le réifie, à en contrôler les mécanismes acteurs. Cependant, un message créé par le système suit un comportement standard. L'objectif d'une réification d'un tel message n'est donc pas d'en contrôler les mécanismes acteurs mais les mécanismes propres aux messages. La réification d'un message créé par le système vers un méta-acteur spécialisé correspond donc à l'attente du programmeur. Elle facilite des opérations comme la modification des valeurs des accointances d'un objet du langage: cette opération ne nécessite plus que la réification de l'objet vers un méta-acteur ad hoc, la modification de la valeur souhaitée, puis une opération de réflexion. L'utilisation de méta-acteur courant aurait, elle aussi, permis une telle opération mais au prix d'une complexité accrue.

Les méta-acteurs spécialisés suivent, pour chaque objet du langage, un protocole diffèrent. Ce protocole est plus contraignant que celui des méta-acteurs car l'ensemble des méthodes qui y sont décrites doivent apparaître dans tout méta-acteur spécialisé.

Prenons pour exemple le comportement d'un méta-message. Ce comportement est le suivant:

```
; Comportement d'un méta-message
(def-behavior a-meta-message(mesg-to mesg-sender mesg-forward-to mesg-content
                             mesg-take mesg-type) t
  ((:get−values) (become)
   (result mesg-to mesg-sender mesg-forward-to mesg-content mesg-take mesg-type))
  ((:get−to) (become) (result mesg-to))
  ((:get−sender) (become) (result mesg-sender))
  ((:get−forward−to) (become) (result mesg-foward-to))
  ((:get−message) (become) (result mesg-content))
  ((:get−take) (become) (result mesg-take))
  ((:get−type) (become) (result mesg-type))
  ((:set−to mesg-to) (update) (result mesg-to)))
```

Les accointances d'un méta-message reflètent les différents attributs de sa dénotation: le destinataire du message, son expéditeur, l'adresse éventuelle de la réponse, le contenu du message, le rang de la méthode qui filtre le message ainsi que le type du message.

Le méta-message est un acteur sérialisé. Il doit être capable de modifier l'adresse du destinataire du message contenue dans l'accointance *mesg-to*. Cette modification est rendue possible par la méthode :set−method. Les autres méthodes permettent de lire les valeurs des attributs du message.

Tout méta-message devra pouvoir répondre à l'ensemble des messages prévus par le protocole.

7.9 Instructions, méta-instructions et réificateurs

Avec les méta-acteurs, l'utilisateur accède aux seuls mécanismes d'un acteur. Ces mécanismes n'incluent pas l'interprétation des instructions contenues dans le corps d'une méthode. Les outils décrits dans cette section étendent la réflexivité à la manipulation des instructions de LActE/R.

7.9.1 Le cas particulier d'une couche acteur

Le caractère de couche de LActE/R impose certaines limitations sur le type de réflexivité relative à l'interprétation des instructions du langage. Si un modèle tel que ABCL/R [Watanabe 88] fournit un interprète complet et entièrement manipulable par l'utilisateur, il le doit à son caractère de langage à part entière et à l'abandon de la notion d'efficacité. Le contexte et les objectifs spécifiques de LActE/R ont conduit à une solution différente.

Implémenté au dessus de Common Lisp, LActE/R ne dispose pas d'un interprète complet qui regrouperait les instructions du langage support de l'implémentation et celles introduites par la couche. Une telle approche impliquerait l'écriture d'un interprète complet pour Common Lisp en langage d'acteurs qui rendrait difficile l'obtention d'un système efficace. Or, outre la contrainte de compatibilité de LActE/R avec LActE, l'efficacité est un objectif essentiel dans la conception de LActE/R.

Aussi notre couche acteur utilise-t-elle un modèle plus approprié de réflexivité.

Plutôt que d'utiliser un interprète unique pour l'ensemble des instructions d'une méthode, LActE/R distribue l'interprète sur les instructions. Chaque instruction dispose alors de son propre interprète réifiable individuellement. Ce découpage rapproche le modèle de réflexivité propre à LActE/R de la philosophie acteur.

7.9.2 Un modèle de réflexivité basé sur les instructions

La restriction des aspects réflexifs de LActE/R aux éléments introduits par la couche acteur a induit un modèle de réflexivité organisé autour de la notion d'instruction. Ce recentrage s'effectue à l'aide de trois outils:

1. Un constructeur d'instructions: le **def-instruction**.

2. Une abstraction de la notion d'interprète: les *réificateurs*.

3. Des micro-interprètes attachés à chacune des instructions: les *méta-instructions*.

Chacun de ces outils contribue, à son niveau, à rendre manipulables les instructions introduites par la couche acteur.

7.9.3 La définition des instructions

Le constructeur d'instructions observe la syntaxe suivante:

```
(def-instruction <nom>
   :syntax <instruction-lambda-list>
   :compilation <compiled-form>
   :lambda-list <implementation-lambda-list>
   :implementation <implementation-function>
   :reified-form <meta-instruction-request>)
```

Les différents paramètres du constructeur d'instructions ont la signification suivante:

– *<nom>* est le nom donné à l'instruction. Toute référence future à l'instruction se fera à l'aide de son nom.

– *<instruction-lambda-list>* correspond à la liste des paramètres de l'instruction. Les mots clés y sont acceptés.

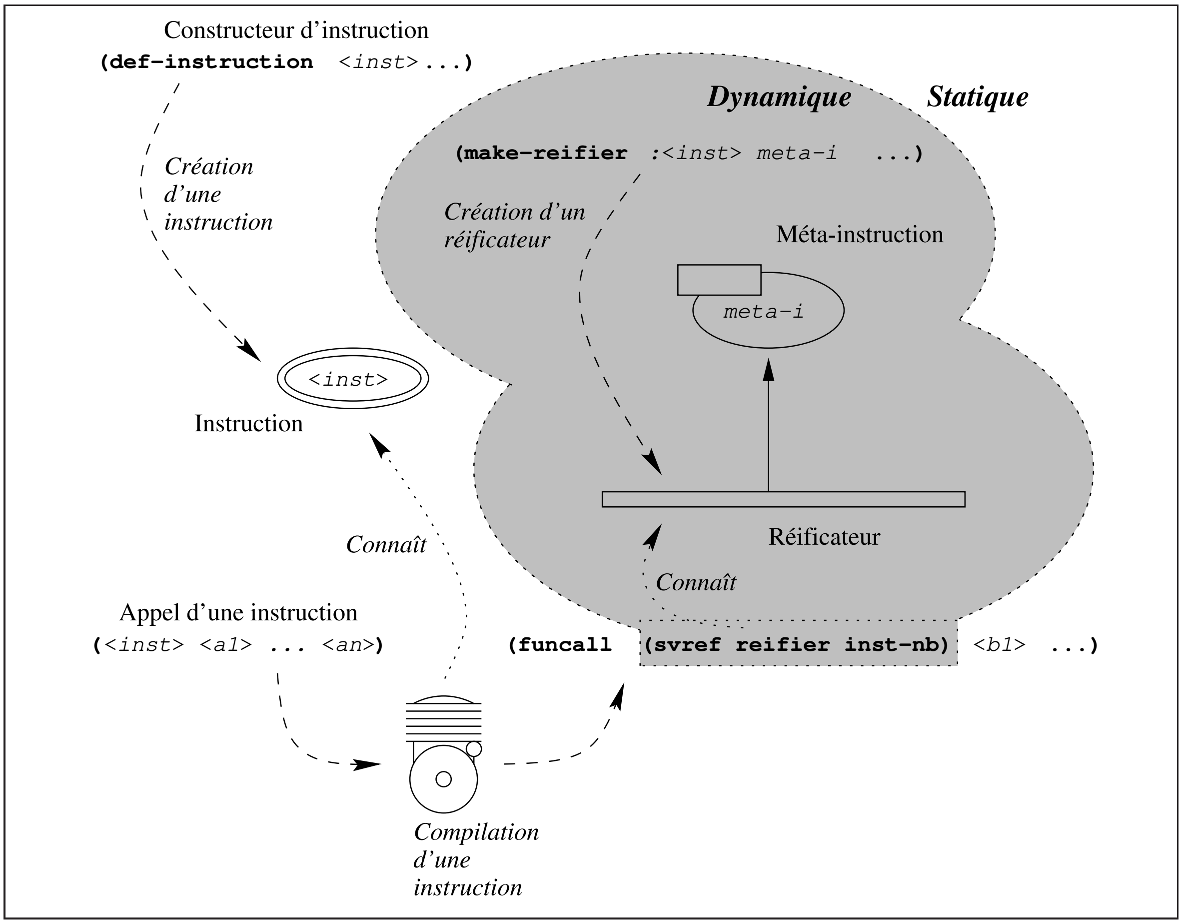

FIG. 7.5 – *Instructions et réflexivité*

- <*compiled-form*> correspond à la forme compilée de l'instruction. La forme compilée de l'instruction, équivalente au corps d'une macro dans lequel l'appel à l'implémentation de l'instruction est remplacé par le symbole **call-implementation**. L'utilisation de cette forme intermédiaire entre l'appel de l'instruction et l'appel de son implémentation autorise la récupération d'informations lexicales.

- <*implementation-lambda-list*> correspond à la liste des paramètres passés à la fonction qui implémente l'instruction ou à la fonction qui construit et envoi la requête de demande d'interprétation de l'instruction à sa méta-instruction.

- <*implementation-function*> est l'adresse de la fonction Common Lisp qui implémente l'instruction. C'est cette fonction qui est appelée lorsque l'instruction n'est pas réifiée.

- <*meta-instruction-request*> défini la forme du message transmis à la méta-instruction par l'instruction réifiée, forme nommée par la suite *forme réifiée*. Cette forme est évaluée dans un environnement défini par <*implementation-lambda-list*>.

La définition d'une instruction nécessite la spécification de la totalité des champs du constructeur.

La forme compilée

La *forme compilée* de l'instruction (ou <*compiled-form*>) est la face la plus obscure d'une instruction. Elle est cependant une des pièces maîtresses de la définition des instructions.

Au premier abord, elle semble inutile. Il est tout à fait concevable de supprimer ce champ du constructeur d'instructions et d'obtenir des fonctionnalités similaires. Les cinq autres champs suffisent à générer les éléments nécessaires à l'utilisation d'une instruction. Par exemple, l'appel (<*inst*> $< a_1 > \ldots < a_n >$) est directement convertible en:

> (*funcall* (*svref reifier* (*get-instruction-nb* <*inst*>))
> $< a_1 > \ldots < a_n >$)

L'indirection introduite permet alors de choisir de réifier ou non une instruction après que la s-expression précédente ait été compilée.

La forme compilée d'une instruction se révèle être indispensable dès lors que l'environnement lexical est pris en compte.

L'utilisation de la compilation des méthodes et l'objectif d'efficacité se sont traduits par une absence de modélisation de l'environnement lexical. Les méta-instructions, ainsi que les fonctions qui implémentent les instructions, n'ont pas accès à l'environnement lexical dans lequel les instructions sont appelées. La forme compilée autorise la récupération d'informations lexicales et leurs transmissions sous forme de paramètres à l'implémentation de l'instruction.

La forme compilée se présente sous la forme d'un corps de macro dont l'expansion à lieu dans le contexte lexical de l'appel de l'instruction. Elle donne ainsi accès à l'environnement lexical.

Par exemple, l'instruction **result** prend comme paramètre un ensemble de résultats qu'elle envoie à la continuation du calcul dont l'adresse se trouve contenue dans la variable *forward-to*. C'est donc la forme compilée qui se charge de récupérer cette information avant de la transmettre à l'implémentation. Pour l'instruction **result**, cette forme compilée est la suivante:

```
`(call-implementation (list ,@the-result) forward-to)
```

où le **call-implementation** est la forme abstraite de l'appel à l'implémentation, l'expression (*list* ,@*the-result*) force l'évaluation des paramètres de l'instruction dans l'environnement lexical courant, et *forward-to* permet d'obtenir l'adresse de la continuation.

Cet exemple montre aussi un autre rôle de la forme compilée: elle permet d'ajuster les paramètres de l'instruction à ceux de l'implémentation. Ces paramètres ne sont pas toujours identiques et nécessitent parfois des aménagements.

7.9.4 Les réificateurs

Les réificateurs offrent une abstraction de la notion d'interprète. Ils regroupent en un seul objet des références sur chacune des instructions de la couche LActE/R. À chaque couple (<message reçu>, <méthode>) est associé un réificateur qui détermine quelle va être l'interprétation des instructions LActE/R contenues dans le corps de la méthode. En particulier, il détermine quelles sont les instructions réifiées et quelles sont les méta-instructions de chacune des instructions réifiées. C'est de cet aspect que les réificateurs tirent leur nom.

Les réificateurs doivent leur existence au compromis entre efficacité et dynamisme. Ils permettent de réifier une instruction en cours d'exécution bien que les méthodes aient été compilées. La compilation des méthodes assure l'efficacité du système et les réificateurs son dynamisme.

Création des réificateurs

Les réificateurs sont créés par duplication d'un autre réificateur. Cet autre réificateur est soit le réificateur par défaut, lors de l'utilisation de la fonction *make-reifier*, soit un réificateur explicitement désigné, lors de l'utilisation de la fonction *duplicate-reifier*. Les paramètres passés à l'une des fonctions de création désignent les instructions réifiées, chaque mot-clé qui nomme une instruction étant suivi de l'adresse de la méta-instruction correspondante. Par exemple, (*make-reifier* :send *meta-send*) crée un réificateur pour lequel l'instruction **send** est réifiée et dont la méta-instruction est l'acteur *meta-send*.

Lors de la duplication d'un réificateur, et sauf mention explicite dans les paramètres de la fonction, ses caractéristiques sont conservées. Si le réificateur du paragraphe précédent est sauvegardé dans la variable *a-reifier*, l'expression (*duplicate-reifier a-reifier*:ask *meta-ask*) aura pour résultat la création d'un réificateur dont les instructions **ask** et **send** seront réifiées. La première doit sa réification aux paramètres passés à la fonction, tandis que la seconde le doit au fait d'être déjà réifiée dans le réificateur dupliqué.

Partage et réificateurs

Un même réificateur est partageable entre plusieurs acteurs, le réificateur associé à un acteur étant partagé par défaut entre tous les couples (<message reçu>, <méthode>) qu'il génère. Si l'utilisateur ne crée par de réificateur, un réificateur par défaut est utilisé. Ce réificateur ne réifie aucune de ses instructions. Le comportement obtenu est celui d'un acteur qui ne fait pas appel à la réification pour l'interprétation de ses instructions i.e. le comportement d'un acteur de LActE.

Une autre forme de partage est permise par les réificateurs: le partage des méta-instructions. Plusieurs instructions sont réifiables vers une même méta-instruction. Ceci peut bien sûr s'appliquer à l'ensemble des instructions. Dans ce dernier cas, la méta-instruction utilisée est comparable à cet unique interprète propre aux autres modèles de réflexivité (ex.

[Watanabe 88]).

Le partage des méta-instructions ne permet pas seulement de réduire l'occupation mémoire d'un système d'acteurs. Il facilite aussi un partage des informations encapsulées au sein de la méta-instruction.

Statut des réificateurs

Bien qu'objets de la couche LActE/R, les réificateurs ne sont pas des acteurs. Cette rupture de l'uniformité linguistique provient de la volonté de conserver son efficacité au système. L'utilisation d'acteurs pour implémenter les réificateurs aurait introduit de nombreux échanges de messages à chaque interprétation de l'une des instructions de LActE/R, que l'instruction soit réifiée ou non. En donnant aux réificateurs un statut spécifique, le coût de l'appel d'une instruction non réifiée se résume à une indirection. Les acteurs qui ne font pas usage des capacités réflexives ne sont ainsi pas trop pénalisés par la présence de la réflexivité.

7.9.5 Les méta-instructions

Une méta-instruction n'est autre qu'un acteur chargé de l'interprétation d'une instruction. Elle agit comme un interprète local d'une instruction de la couche LActE.

Le lien entre instruction et méta-instruction est établi par le réificateur. C'est lors de la création d'un réificateur que l'utilisateur de LActE/R désigne la méta-instruction d'une instruction. L'existence de la méta-instruction et de l'instruction est une condition préalable à l'établissement de ce lien.

L'acteur méta-instruction ne doit son statut de «méta-instruction» qu'à l'existence d'un tel lien.

Outre le réificateur, l'instruction participe à la définition de ce lien. Lors de la définition d'une instruction, le champ `:reified-form` spécifie la forme de la requête envoyée à la méta-instruction par le système.

De part son statut d'acteur, une méta-instruction est réifiable. La réification d'une méta-instruction s'opère vers un méta-acteur, lui aussi réifiable. Chaque méta-instruction est ainsi susceptible de servir de base à une tour potentiellement infinie de méta-acteurs.

7.10 Comparaison avec d'autres travaux

C'est avec le modèle de réflexivité d'ABCL/R [Watanabe 88] que LActE/R présente le plus d'analogies. Tous deux basent la réflexivité sur la notion de méta-objet. Et le découpage de la machine acteur par LActE/R n'est pas très éloigné de celui effectué dans ABCL/R. Cependant la démarche adoptée dans les diverses implémentations d'ABCL/R se caractérise par un effort prononcé sur les techniques d'implémentation et d'optimisation d'un modèle défini sans que la question des performances ne soit prise en compte. LActE se singularise par un modèle fortement influencé par cette question.

De 3-Lisp [Maes 87], LActE/R reprend les méta-objets spécialisés. Notre interprète ne va pourtant pas aussi loin que 3-Lisp dans l'uniformité linguistique. Ce dernier ne souffre pas de l'approche «couche» propre à LActE/R.

Les messages :reify et :reflect présentent des analogies avec ceux de T. Tanaka [Tanaka 90]. Dans les deux cas ils permettent une utilisation de la réflexivité «à la demande» synonyme de dynamisme. Ils s'opposent donc à la propension au statique prônée par les autres langages. Cependant, ces deux approches se distinguent par la nature des opérations qu'elles impliquent. Le modèle de T. Tanaka ne permet que la réification des données de l'acteur alors que notre modèle utilise les messages réifiant pour installer les méta-acteurs.

7.11 Conclusion

Tout modèle de réflexivité implique des choix. Le modèle de réflexivité de LActE/R résulte de nombreux choix dont certains sont parfaitement rationnels et d'autres un peu plus arbitraires. Aussi LActE/R ne prétend pas à la perfection. Mais, ayant bénéficié d'une interaction soutenue avec un utilisateur, J.C. Pazzaglia [Pazzaglia 97], nous avons pu affiner notre modèle jusqu'à le rendre assez complet pour couvrir la plupart des besoins.

Le caractère de couche de LActE/R ainsi que la prise en compte dès la conception de l'efficacité de l'exécution ont fortement influencés le modèle de réflexivité. L'absence d'un interprète complet attaché à chaque acteur comme dans ABCL/R [Watanabe 88] dans LActE/R résulte de ces contraintes. Pour autant, le triplet (**def-instruction**, réificateur, méta-instruction) introduit un modèle de l'interprétation plus proche de la philosophie acteur qui respecte une certaine efficacité du système et qui reste compatible avec la compilation des méthodes.

Enfin, LActE/R se distingue des autres modèles par son explicitation des opérations de réification et de réflexion. Elles dotent LActE/R de tours de méta-acteurs dynamiques et introduisent dans la programmation réflexive une maîtrise fine de l'utilisation de la réflexivité.

Chapitre 8

Implémentation de LActE/R

L'implémentation de LActE/R reprend pour une large part celle de la version non réflexive. La réflexivité y est introduite en modifiant les définitions de certaines structures de données ainsi que les fonctions qui les gèrent. Le propos de ce chapitre n'est pas de décrire la totalité de l'implémentation de LActE/R mais d'éclaircir les points les plus importants de l'introduction de la réflexivité dans LActE/R.

8.1 De LActE à LActE/R

LActE/R n'est pas une réalisation ex-nihilo: il est le résultat de modifications de l'interprète LActE. De ce dernier, il reprend l'essentiel de l'implémentation.

Les aspects essentiels de l'implémentation de LActE sont les suivants:

- Des structures de Lisp spécifiques représentent chaque objet de la couche LActE. Un acteur utilise une structure différente de celle d'un message.
- Les instructions sont implémentées par des fonctions Lisp.
- La concurrence de LActE s'appuie sur les extensions vers les processus des Common Lisp utilisés (principalement Allegro Common Lisp). Chaque exécution de méthode se traduit par la création d'un processus qui devient le support de l'activité de la méthode. Par contraste, l'activité de la machine acteur n'utilise pas de processus spécifique. Elle est due au processus déclencheur i.e. celui depuis lequel un message est envoyé ou celui depuis lequel le changement de comportement de l'acteur est spécifié, ses instructions se chargeant aussi de l'activation de la machine acteur. Le nombre de processus nécessaire au fonctionnement du système est ainsi minimisé[1]
- Le contrôle des accès en concurrence à la boîte aux lettres d'un acteur passe par la pose de locks. Afin de garantir un haut degré de concurrence, ces locks n'entourent que les

1. Les processus de Common Lisp nommés à tord «processus légers» ont un coût non négligeable. La réduction du nombre de processus nécessaire à un calcul se traduit donc par une augmentation des performances.

portions de code qui le nécessitent. Aussi plusieurs envois de message vers un même acteur peuvent avoir lieu en concurrence.

L'implémentation de LActE/R s'appuie sur celle de LActE comme support au modèle de calcul.

L'introduction de la réflexivité dans l'interprète LActE se traduit par:

- Un redécoupage des fonctions de l'interprète afin de les mettre en adéquation avec le modèle de réflexivité. En particulier, ce redécoupage résulte de l'introduction des instructions au sens de LActE/R.
- L'enrichissement des structures qui représentent les objets du langage pour qu'elles puissent décrire les informations utilisées par la réflexivité, en particulier le lien méta/-dénotation.
- L'extension de ces structures pour qu'elles se comportent en acteur lorsqu'un message leur est envoyé.
- L'introduction d'indirections ou de conditionnelles qui assurent la prise en compte des opérations réifiées.
- L'ajout de l'ensemble des comportements par défaut des méta-acteurs.
- L'utilisation systématique du **def-instruction** pour définir les instructions de LActE/R.
- La création d'acteurs de méta-niveau qui offrent une interface uniformisée avec certains éléments du système.

Dans la suite de ce chapitre nous allons décrire en détail les points les plus délicats de l'implémentation de LActE/R.

8.2 L'uniformité linguistique

L'une des caractéristiques de LActE/R est l'uniformité linguistique. Par cela il est fait allusion à la capacité de tous les objets de la couche acteur de recevoir et de traiter des messages. Si, dans la version non réflexive de LActE, les acteurs sont les seuls à recevoir des messages, les autres objets qui utilisent pour leur implémentation des structures spécifiques n'y ont pas la même facilité.

Lors du passage à la version réflexive de LActE/R, les objets du langage ont gagné la capacité de recevoir des messages et adopté pour l'utilisateur une vision acteur. Ils partagent désormais cela avec les acteurs tout en conservant leur structure spécifique. Les objets du langage adoptent donc un traitement particulier de la réception des messages.

L'implémentation du traitement de la réception d'un message par un acteur ou par un objet du langage ne doit pas faire ressortir les différences entre ces entités. L'utilisateur lorsqu'il dialogue avec ces différents objets utilise des envois de message pour lesquels les différences d'implémentation entre objets du langage et acteurs ne transparaissent pas. L'implémentation de LActE/R unifie la réception et le traitement des messages à travers une seule et même interface.

Tous les objets du langage sont implémentés par des structures Lisp qui intègrent une autre structure détentrice des éléments nécessaires à la réception des messages. Cette intégration est réalisée à l'aide du :include des structures de Common Lisp qui permet d'inclure les champs d'une structure au sein d'une autre réalisant ainsi une forme simplifiée d'héritage. Les champs inclus sont alors accessibles à l'aide des mêmes fonctions d'accès pour toutes les instances de la structure incluse et de la structure incluante.

Cette structure minimale nommée *mailbox* est formée de deux champs. Le premier de ces champs est un pointeur vers une fonction à trois arguments, le message à recevoir, le destinataire du message et le contenu du message à recevoir. Cette fonction implémente la réception des messages pour l'objet en question. Le second champ est un lock qui permet de contrôler la lecture et l'application de la fonction de réception des messages. Ce lock est nécessaire afin d'exclure toute activité de réception de message lors de la modification de la fonction de réception des messages due à certaines opérations comme la réification. La définition de la structure *mailbox* est la suivante:

```
(defstruct mailbox
          (put-message       #'default-put-message)
          (put-message-lock (make-lacte-lock)))
```

L'envoi d'un message à un tel objet, décrit par une structure qui inclut la structure précédente, peut alors s'écrire:

```
(defun send-message(message)
  (let ((mesg-to      (message-to message))
        (mesg-message (message-message message)))
    (process-lacte-lock-might-lock (mailbox-put-message-lock mesg-to))
    (funcall (mailbox-put-message mesg-to) message mesg-to mesg-message)))
```

La fonction *send-message* est accessible à l'utilisateur de LActE/R. Elle se contente de poser le lock sur la fonction de réception des messages du destinataire et d'appeler cette fonction en lui passant le message, le destinataire du message ainsi que le contenu du message. Les locks utilisés ici sont des locks aux fonctionnalités étendues qui seront décrits plus précisément par la suite. Notons que la fonction *send-message* ne fait que poser le lock qui devra être levé par la fonction de réception des messages. Cette dernière pouvant appeler d'autres fonctions qui n'ont pas à être protégées par le lock, le *send-message* ne peut se charger de retirer le lock après l'appel de la fonction.

L'utilisation d'un lock justifie la présence des deux derniers paramètres des fonctions de réceptions de message. Sa présence interdit les interactions par envois de message avec les autres objets du système durant la réception d'un message. De telles interactions pourraient conduire indirectement à l'envoi d'un message vers l'entité qui traite la réception d'un message. Si la fonction de réception de message est alors lockée, il en résultera un blocage du système (deadlock). En passant les deux derniers paramètres à la fonction de réception des messages, cette dernière est capable d'effectuer un traitement du message reçu sans avoir à interagir dans la zone contrôlée par le lock avec le message reçu qui peut être un acteur. Cette propriété est utilisée pour l'implémentation des objets du langage.

Le type d'envoi de message n'influe pas sur la réception des messages. Seul l'émetteur est affecté par les différents types d'envois de message. Par exemple, lors de l'envoi d'un message synchrone, le processus depuis lequel le message est émis est suspendu jusqu'à la réception d'une réponse. Le récepteur de la requête ne perçoit le caractère synchrone de l'envoi du message reçu qu'à travers le type du message et la continuation qu'il spécifie. Toutes les instructions d'envoi de message utilisent d'ailleurs la même primitive *send-message* pour effectuer cet envoi.

Toute structure Lisp incluant la structure *mailbox* est susceptible d'être récepteur de message. La nature de l'objet qu'elle représente n'est pas prise en compte lors d'un envoi de message. C'est la fonction de réception de message, particulière à chaque type d'objet, qui tient compte de la nature de l'objet receveur.

Des objets particuliers peuvent ainsi être introduits dans le système. Un trou noir qui avale tous les messages qu'il reçoit peut être défini de la manière suivante:

```
(setf black-hole
      (make-mailbox
         :put-message #'(lambda (message self content)
                            (process-lacte-lock-might-unlock
                               (mailbox-put-message-lock self)))))
```

Ce schéma d'implémentation autorise la coexistence d'objets de nature différente au sein de LActE tout en dialoguant de manière uniforme entre eux. Il rend aussi possible la modification à tout instant du mode de réception des messages d'un objet. Cette dernière propriété est indispensable au fonctionnement dynamique du processus de réification dans LActE/R.

8.3 Implémentation des messages et méthodes du méta

Les messages du méta (:reify-to, :reflect, :meta, ...) présentent une caractéristique qui les différencie notablement des autres messages acceptés par un acteur: leur interprétation est fonction du statut de l'acteur dans le contexte de la réflexivité. Par exemple, la valeur retournée par :meta varie au cours du temps en fonction de la présence ou de l'absence d'un méta-acteur qui interprète l'acteur en question. Les opérations correspondant aux autres messages du méta voient, elles aussi, leur sémantique varier en fonction des modifications du statut de l'acteur qui les reçoit. Réifier un acteur qui ne possède pas de méta-acteur n'a pas,

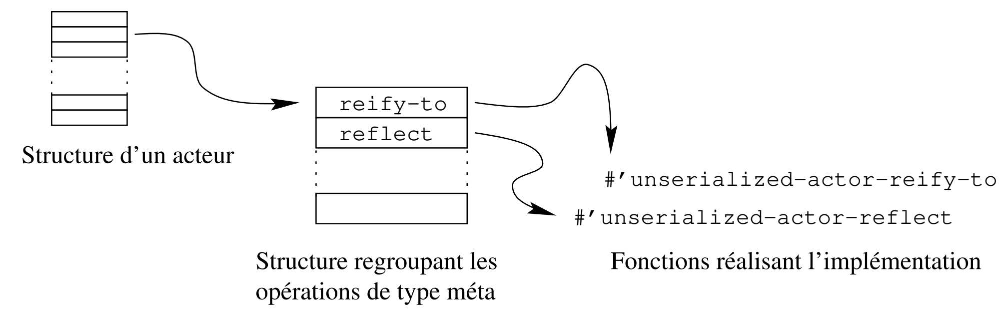

FIG. 8.1 – *Structure des pointeurs vers les fonctions des messages du méta*

en LActE/R, la même sémantique que réifier un acteur qui possède déjà un méta-acteur.

La spécifité de ces opérations relativement à l'acteur auquel elles sont appliquées demande une implémentation différente des autres méthodes. En particulier, l'implémentation des méthodes :reify−to et :reflect ne doit pas être attachée au comportement courant de l'acteur, comportement qui peut être commun à un ensemble d'acteurs et dont l'acteur peut changer lors de l'exécution d'une méthode, mais à l'acteur lui même, qui est détenteur de son statut au sein de la tour à laquelle il appartient. Donc si les méthodes correspondant à ces messages du méta doivent être présentes au niveau du comportement afin de leur assurer leur statut de méthode, leur implémentation doit, elle, se trouver au niveau de l'acteur.

LActE/R résout ce problème en attachant une structure spécifique à chaque objet du langage. Cette structure contient des pointeurs vers les différentes fonctions qui implémentent les opérations correspondant aux messages du méta dont la sémantique change significativement avec la modification du statut de l'acteur (Fig. 8.1). D'autre part tous les comportements contiennent les méthodes qui acceptent ces messages. Leur statut spécifique font d'elles des méthodes dont le code n'est pas explicitement disponible à l'utilisateur. Par contre, la forme compilée de ces méthodes, forme qui n'est pas lisible par l'utilisateur, contient le corps qui les implémente. Ce corps est composé d'une partie commune aux opérations qu'il implémente et d'un appel de la partie spécifique à l'acteur de l'opération. L'appel à la partie spécifique de l'opération passe par la structure *meta-fct-descr* attachée à l'acteur et qui contient les pointeurs vers les fonctions qui réalisent l'implémentation. Par exemple, le corps de la méthode :reify−to est pour les acteurs non sérialisés:

```
  ; Construction de la méthode de réification des acteurs
  ; non sérialisés
  ; Construction d'une méthode qui ne peut pas être réifiée
  ; (Les méthodes des messages du méta ne sont pas réifiables)
(make-actor-method-not-reify
        :pattern '(:reify-to behavior proxy &rest acquaintances)
             ; La forme explicite du corps de la méthode n'est pas
             ; disponible
          :body    '(:hidden-definition)
          :compiled? t
```

```
        :compiled-pattern #'compiled-pattern-reify-to
        :compiled-body    #'unserialized-actor-reify-to)
```

avec

```
; Forme compilée du test de pattern matching
(defun compiled-pattern-reify-to(message-content)
     (and (listp message-content)
          (> (length message-content) 2)
          (eq (first message-content) :reify-to)))
```

et

```
; Forme compilée de la méthode de réification
(defun unserialized-actor-reify-to(pself self sender ...)
   (declare (ignore sender meta ...))
   ; Initialisation des données spécifiques du processus
   (setf (result-mailbox-owner pself) (current-process))
    ; Retour du résultat de l'appel de la fonction
    ; réalisant l'implémentation
   (default-send
     (list
       ; Lock sur la fonction de réception des messages
       (with-process-pseudo-lock-lock
              ((mailbox-put-message-lock self))
               ; Appel de la fonction réalisant l'implémentation
               ; depuis le descripteur de l'acteur récepteur
              (funcall (meta-fct-descr-reify                          (i)
                          (actor-meta-fct-descr (mailbox-actor self)))
                     self meta-behavior meta-proxy
                     meta-acquaintances)))
     forward-to self self 1 :send))
```

L'ensemble des comportements d'acteurs non sérialisés incorpore automatiquement la méthode précédente ainsi que les autres méthodes relatives aux traitements des messages du méta. Le statut spécifique de ces méthodes, en particulier le fait qu'elles ne soient pas réifiables, est garanti par l'utilisation d'un constructeur dédié qui les équipe d'opérations de réification et de réflexion vide.

L'implémentation de la méthode :reify–to, disponible uniquement sous sa forme compilée, consiste à retourner le résultat de l'appel à la fonction qui implémente l'opération de réification pour l'acteur adoptant un tel comportement ((i) grisé dans de code). L'appel de cette fonction est réalisé par un *funcall* sur le contenu de la structure *meta-fct-descr* désignée par le champ *actor-meta-fct-descr* de l'acteur. C'est ainsi que l'implémentation des opérations relatives aux messages du méta passe par des fonctions rattachées aux acteurs, tout en respectant la présence des méthodes correspondantes qui, elles, sont rattachées aux comportements. La valeur de ces fonctions dépend du statut de l'acteur dans sa tour de méta-acteurs.

Cette implémentation, qui est reprise par l'ensemble des messages du méta et pour tous les objets de la couche LActE/R, permet de modifier l'implémentation des différents messages du méta lorsque le statut de l'acteur est modifié, de conserver cette implémentation lorsque l'acteur change de comportement, et de respecter le modèle de réflexivité de LActE/R au niveau de la gestion de ces méthodes.

8.4 Implémentation des opérations de réification et de réflexion

Parmi les mécanismes qui participent à l'introduction de la réflexivité dans LActE/R, les opérations de réification et de réflexion sont les plus délicates à implémenter. La difficulté essentielle réside dans la cohérence qu'il faut assurer dans le contexte de la concurrence lors de la construction ou la réduction d'une tour. L'entité qui est réifiée ou réfléchie doit l'être alors qu'elle peut être en pleine activité et continuer à recevoir des messages en provenance d'autres objets du langage ainsi que de l'interprète LActE/R.

8.4.1 Activités concurrentes à la modification d'une tour de méta-acteurs

Les opérations de réification et de réflexion se déclenchent par des messages, dits messages du méta, dont le traitement par l'acteur qui les reçoit est identique aux autres messages. La réification d'un acteur ou la réflexion d'un méta acteur interviennent par conséquent dans un contexte similaire à celui d'un autre message. Les activités qui peuvent avoir lieu en concurrence avec ces opérations ne se différencient pas de celles qui entourent le traitement d'un autre type de message.

Les activités concurrentes se divisent en deux branches: les interactions avec l'interprète LActE/R et les interactions avec d'autres acteurs.

Les interactions avec l'interprète LActE/R traduisent le fonctionnement de la machine acteur.

Les interactions avec les autres acteurs du système passent par des envois de message. Au niveau de l'acteur, elles se traduisent par l'activité de réception de message réalisée par la fonction sur laquelle pointe le champ :put−message de la boîte aux lettres de l'acteur.

8.4.2 Implémentation de la première réification

LActE/R distingue, au niveau de l'implémentation, la première réification d'un acteur des réifications suivantes. La raison d'une telle distinction provient de la différence de contexte entre ces opérations. Aussi nous commençons par nous intéresser à la première réification d'un objet de LActE/R, avant d'en décrire la réflexion. Suite à cela, le contexte des réifications suivantes étant clairement défini, la description de l'implémentation des opérations de réification faisant suite à une opération de réflexion pourra avoir lieu.

La réification consiste en LActE/R à substituer à l'interprète LActE/R un méta-acteur chargé de l'interprétation des mécanismes propres à l'entité réifiée. Dès lors, toutes les activités «acteur» de l'acteur réifié seront prises en charge par le méta-acteur et non plus par l'interprète LActE/R.

La première réification s'opère sur un acteur parfaitement autonome, directement manipulé par l'interprète LActE/R, et qui ne possède pas, et n'a encore jamais eu de méta-acteur. La création du méta-acteur sera l'une des tâches à effectuer lors de la réification.

Contexte de la première réification

L'opération de réification est une opération dynamique qui intervient alors que l'acteur à réifier est en pleine activité. En particulier, d'autres méthodes que la méthode :reify−to peuvent être en cours d'exécution. Ceci est vrai non seulement pour les acteurs non sérialisés mais aussi pour les acteurs sérialisés qui acceptent l'exécution concurrente d'une autre méthode dès que le nouveau comportement de l'acteur a été spécifié. Cependant, un acteur sérialisé ne peut avoir plus d'une méthode n'ayant pas encore effectué de **become**.

Dans le cas d'acteurs sérialisés, l'exécution d'une méthode relative à une opération de réification exclut l'activation d'autres méthodes tant qu'elle n'a pas effectué son **become**. L'activité de sa machine acteur est par conséquent réduite à la bufferisation des messages reçus par l'acteur.

Dans le cas des acteurs non sérialisés, la réception d'un message enclenche immédiatement l'activation de la méthode correspondante. La machine acteur peut donc traiter plusieurs messages alors que la méthode :reify−to est en cours d'exécution.

Outre les activités issues de l'acteur, une autre activité concurrente qui modifie l'état d'un acteur existe dans LActE/R: la réception des messages. Afin de conserver un haut niveau de concurrence, la réception d'un message peut se faire en parallèle à la réception d'un autre message. Cette caractéristique minimise les attentes lors des interactions entre les entités du langage. Dans le cadre de la réification, il faut cependant être capable d'interrompre cette activité pour pouvoir en modifier la sémantique.

Les méthodes en cours d'exécution, l'activité de la machine acteur ainsi que l'arrivée de messages en concurrence constituent les activités concurrentes à la réification.

Les pseudo-locks de LActE/R

La réification d'un acteur nécessite de bloquer certaines des activités concurrentes d'un acteur. La réception des messages ne peut continuer à avoir lieu alors que l'opération de réification va nécessiter de redéfinir cette activité. L'activité de réception de message d'un objet de LActE/R est réalisée par une fonction pointée par le champ :put−message de sa boîte aux lettres.

Pour s'assurer qu'une fonction Lisp n'est pas exécutée à un instant donné, l'extension multithread du Common Lisp utilisé fournit un système de locks. L'exécution d'une fonction Lisp peut être contrôlée en enveloppant l'appel de la fonction concernée par un lock posé

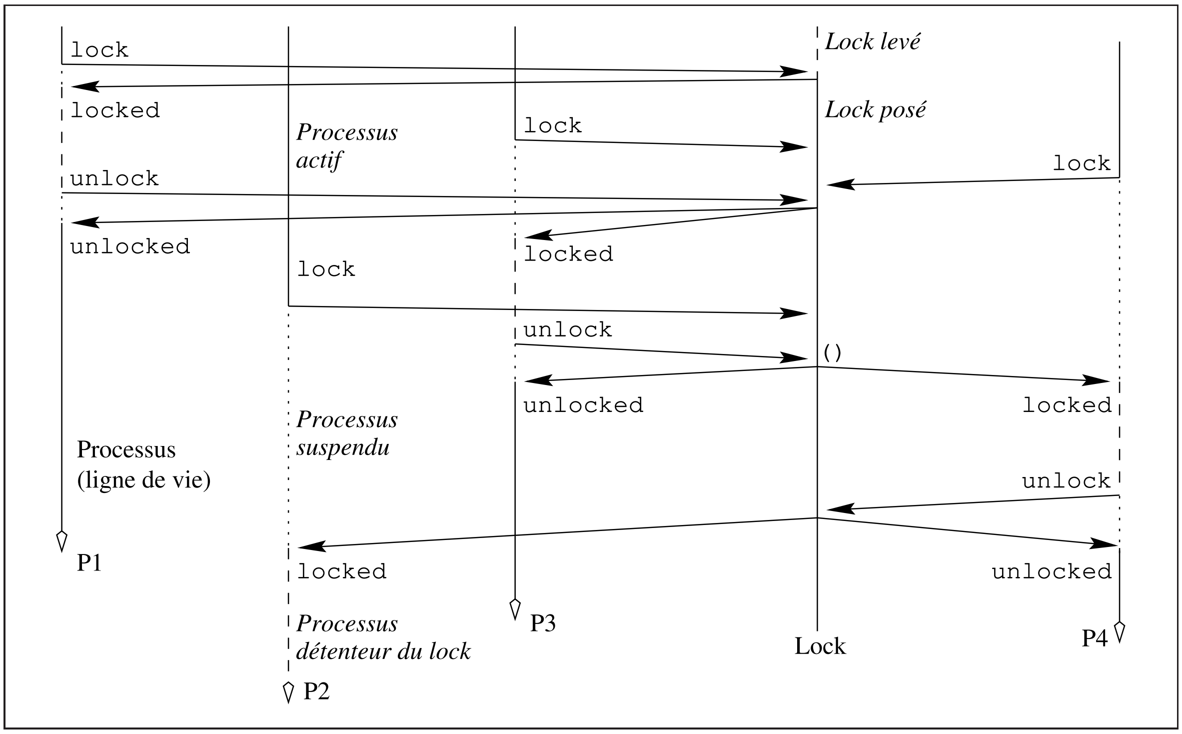

FIG. 8.2 – *Schéma de fonctionnement des locks de Common Lisp*

avant l'appel et retiré après cet appel. Ainsi protégée, une seule lecture et exécution de la fonction réalisant l'activité de réception des messages a lieu à la fois.

La figure (Fig. 8.2) détaille le fonctionnement de plusieurs processus devant obtenir un même lock pour exécuter une section de leur code. Avec les locks classiques, un seul processus à la fois peut être détenteur du lock. Les séquences de code protégées par le même lock se trouvent ainsi séquentialisées les unes par rapport aux autres.

Cette technique très classique ne s'applique pas parfaitement au cas de LActE/R. Elle a le défaut de sérialiser l'exécution des fonctions dont l'appel est ainsi contrôlé. Par exemple, si l'appel de la fonction qui représente l'activité de réception est gardé par un lock, un seul appel pourra avoir lieu à la fois, cet appel bloquant les autres appels jusqu'au retour de la fonction.

Or le contrôle sur l'appel de certaines fonctions n'a, dans LActE/R, de raison d'être que lors de certaines opérations réflexives. En dehors de ces cas assez peu fréquents comparativement aux échanges de messages entre les acteurs, il est souhaitable que la concurrence des réceptions de message soit conservée.

Les pseudo-locks de LActE/R permettent de concilier la concurrence des réceptions de message avec la nécessité de pouvoir bloquer cette activité dans le cas d'une opération réflexive.

Ces pseudo-locks disposent de primitives supplémentaires par rapport aux locks du Common Lisp. Outre les opérations qui permettent d'effectuer un lock classique et son unlock

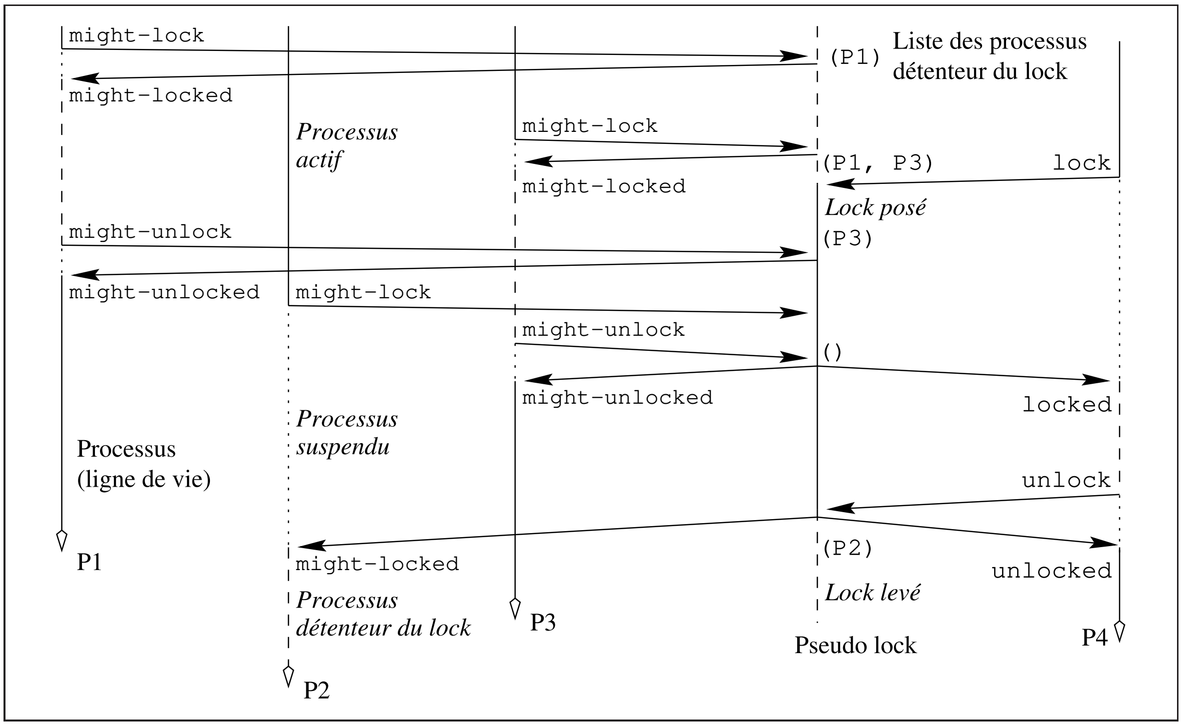

FIG. 8.3 – *Schéma de fonctionnement des pseudo-locks*

symétrique, une opération (*process-lacte-lock-might-lock*) de pseudo-lock, qui est bloquante sur le lock classique sans l'être sur elle même, est introduite. Les appels aux fonctions de réception de message sont enveloppés dans ces pseudo-locks à l'aide de cette dernière opération. L'opération de réification effectue un lock classique sur le pseudo-lock qui bloque la réception des messages.

La figure (Fig. 8.3) schématise le fonctionnement des pseudo-locks de LActE/R. Les processus P1 et P3 signalent leur demande de lock. Ces demandes de lock sont enregistrées mais n'ont pas pour effet de bloquer les autres processus qui effectuent cette même demande. Par conséquent, les deux processus P1 et P3 opèrent en concurrence sans se bloquer sur le pseudo-lock. Par contre, lorsque P4, assimilable au processus d'une méthode :reify−to, pose le lock, il le rend bloquant pour lui-même et les autres processus demandeur du pseudo-lock comme P2 qui est mis en attente de la libération du lock. P4 doit lui même attendre que les processus P1 et P3 libèrent le lock pour vraiment le détenir de façon exclusive.

La concurrence propre aux activités de réception de message peut être ainsi conservée tout en permettant le blocage de ces activités lors des opérations de réification et de réflexion.

Description générale de l'opération de réification

L'opération de réification consiste à substituer un méta-acteur à l'interprète LActE/R. La requête de réification contient les informations nécessaires à la création de ce méta-acteur: son comportement, l'adresse de son proxy, ainsi que les valeurs des accointances qui ne correspondent pas à la description de la dénotation.

Avant même d'entreprendre toute modification de l'acteur, l'ensemble des points qui doivent être modifiés sont protégés par des locks. Ceci nous assure que l'état de l'acteur est constant durant cette opération et surtout qu'il ne reçoit plus de messages. En particulier, aucun autre processus ne lit ni ne détient la fonction qui réalise l'activité de réception des messages.

La première étape de la réification consiste à créer le méta-acteur. L'état de l'acteur étant fixé, les valeurs des accointances du méta-acteur qui représentent les différentes données de l'acteur peuvent être déterminées par lecture des valeurs contenues dans la structure qui implémente l'acteur. La liste complète des valeurs des accointances du méta-acteur est obtenue en ajoutant à la fin de cette liste de valeurs d'accointances celle fournie par l'utilisateur.

La seconde étape consiste à modifier la fonction réalisant son activité de réception des envois de message. Habituellement elle se traduit par une activation de la machine acteur telle qu'elle est implémentée par l'interprète LActE/R. Lorsqu'un acteur est réifié, tout message qui doit être reçu par cet acteur doit se traduire par un envoi d'un :receive−message à son méta-acteur. Dans LActE/R, le dialogue entre les tours de méta-acteurs passe par la base de chaque tour. C'est donc l'acteur à qui on destine initialement le message qui se charge de réifier la réception de tous les messages par un envoi de message :receive−message vers son méta-acteur. Cela se traduit dans l'implémentation par la modification de la fonction réalisant l'activité de réception de message de l'acteur réifié en une fonction qui envoi au méta-acteur un message :receive−message qui enveloppe le message reçu initialement par l'acteur.

C'est ensuite au tour des fonctions réalisant les opérations de réflexion et de réification d'être modifiées. Une fois la réification courante réalisée, l'acteur réifié ne pourra plus l'être jusqu'au moment où il aura été réfléchi. Symétriquement, le méta-acteur doit être doté de la possibilité d'être réfléchi. Cette étape est réalisée en modifiant les champs de la structure *meta-fct-descr* de chaque acteur afin qu'ils pointent vers les fonctions correspondant à la sémantique attendue de ces opérations.

Enfin les champs *meta* et *denotation* de l'acteur et du méta-acteur sont mis à jour. L'acteur devient dénotation du méta-acteur et le méta-acteur méta de l'acteur complétant ainsi la réalisation de la connexion causale.

Ce n'est qu'après ces modifications que les différents locks et pseudo-locks posés seront levés.

Une fois toutes ces modifications accomplies, le méta-acteur a les moyens de remplir pleinement son rôle d'interprète des mécanismes de sa dénotation. L'interprète LActE/R n'agit alors plus sur l'acteur réifié dont l'implémentation est désormais le fait de son méta-acteur. Notons que, outre la réification des valeurs de l'acteur vers les accointances du méta-acteur, c'est par la modification de la fonction réalisant l'activité de réception des messages qu'est obtenue la réelle substitution du méta-acteur à l'interprète LActE/R.

Statut des activités concurrentes de la machine acteur

Les protections qu'utilise LActE/R pour protéger la réification ne permettent pas de garantir l'absence d'activités concurrentes de la machine acteur. Particulièrement dans le cadre des acteurs non sérialisés, plusieurs activations de message peuvent prendre place en concurrence de l'exécution de la méthode :reify−to.

Ce que protègent les pseudo-locks de LActE/R c'est la lecture et l'application de la fonction réalisant l'activité de réception des messages de l'acteur en cours de réification. Ils garantissent aussi que plus aucune modification de l'état de l'acteur n'aura lieu après qu'ils aient été levés.

L'état de l'acteur n'est donc plus sujet à aucune modification par les activités concurrentes de la machine acteur de l'acteur en cours de réification. Et ces activités restent par conséquent cohérentes avec la présence d'un méta-acteur.

Cette concurrence est par ailleurs naturelle dans le modèle acteur. Le modèle de calcul acteur ne garantissant pas l'ordre d'arrivée des messages, les effets de deux événements ayant lieu au même moment peuvent se produire à des instants différents.

Ce modèle de calcul est tel qu'il est pratiquement impossible de stopper toutes les activités relatives à un acteur sans que cela ne soulève un sérieux problème de dead-locks. De plus les activités concurrentes de la machine acteur correspondent à des messages dont le traitement a été enclenché avant le blocage de la réception des messages par la fonction réalisant la réification de l'acteur.

Les activités concurrentes de la machine acteur ne peuvent être dissociées de l'exécution des méthodes qu'elles traitent. L'exécution concurrente de méthodes à la méthode réifiant l'acteur n'est pas un facteur d'incohérence, ces activités concurrentes de la machine acteur ne l'étant pas.

8.4.3 Implémentation de l'opération de réflexion

Des deux opérations relatives à la manipulation des tours réflexives de LActE/R, l'opération de réflexion est la plus délicate à implémenter. Les difficultés de son implémentation proviennent à la fois des possibilités offertes par la réflexivité qui, en ouvrant l'interprète à l'utilisateur, autorise toute une gamme de variations sur le fonctionnement des mécanismes du langage, et par le modèle de calcul sous-jacent, le modèle acteur, qui introduit un certain indéterminisme. Mais commençons par regarder la sémantique de cette opération.

La réflexion consiste en LActE/R à rendre à la dénotation son autonomie. Dans une tour composée d'un acteur et de son méta-acteur, les mécanismes de l'acteur sont interprétés par son méta-acteur. Le fonctionnement de l'acteur dépend alors entièrement de l'implémentation de la mécanique acteur que donne le méta-acteur. Le comportement de ce dernier, défini par l'utilisateur, peut prendre n'importe quelle forme pour autant qu'il respecte un sous-ensemble minimal du protocole. Par exemple, le méta-acteur d'un acteur non sérialisé ne nécessite vraiment pour fonctionner que la méthode de réception des messages, libre au programmeur de spécifier ou non les autres méthodes décrites dans le modèle de réflexivité de LActE/R.

Lors de la restitution de son autonomie à un acteur, l'état courant de la mécanique acteur doit être pris en compte afin d'assurer le maintien de la cohérence entre la description de l'acteur telle qu'elle est donnée par le méta-acteur et celle qui est acceptée par le système. Par état courant il faut comprendre le statut de l'acteur au sein de la tour à laquelle il appartient, ainsi que le statut de la machine qu'il décrit. Ces deux statuts sont, pour des raisons évidentes de simplicité, traités l'un après l'autre.

Réflexion d'une tour de méta-acteurs

Le message du méta qui demande la réflexion d'un méta-acteur peut être envoyé à n'importe quel acteur qui joue le rôle de méta d'un autre acteur. Ce méta-acteur n'est pas nécessairement au sommet d'une tour de méta-acteurs. Lorsqu'il se trouve au milieu de la tour, la véritable opération de réflexion, celle qui consiste à rendre son autonomie à la dénotation, ne peut avoir lieu avant d'avoir rendu son autonomie au méta-acteur. Pour ce faire, il faut commencer par réfléchir l'ensemble des méta-acteurs qui constituent la partie de la tour qui se trouve au dessus de l'acteur réceptionnaire de la requête initiale de réflexion.

Quelle que soit la nature des éléments qui composent une tour de méta-acteurs, l'opération de réflexion débute par un examen de la situation de l'acteur qui reçoit la requête au sein de la tour. Par rapport à l'opération de réflexion, trois situations significatives sont possibles:

- L'acteur constitue la base de la tour. Dans ce cas, la réflexion n'a pas de sens et l'implémentation de la méthode correspondante signifie à l'entité qui en a fait la demande l'impossibilité de réfléchir l'acteur en question.

- L'acteur est au sommet d'une tour de méta-acteurs. La véritable opération de réflexion peut avoir lieu sans condition préalable.

- L'acteur est au milieu d'une tour de méta-acteurs. La réflexion de cet acteur ne pourra se faire qu'après avoir réfléchi l'ensemble des acteurs qui se trouvent être en position de méta par rapport à lui (méta, méta-méta, etc. ...).

La méthode qui implémente l'opération de réflexion commence par traiter le troisième cas. Ce traitement consiste, si le méta-acteur réceptionnaire de la requête initiale de réflexion possède lui même un méta-acteur, à envoyer une requête de réflexion au méta-acteur du méta-acteur courant. Le corps de la méthode de réflexion, pour un acteur sérialisé, prend alors la forme suivante:

```
(defun actor-method-serialized-reflect(pself self sender ...)
    ; Préambule de la méthode
    (setf (result-mailbox-owner pself) (current-process))
    ; Réflexion de la partie supérieure de la tour
```

```
(if (and meta denotation)
  (default-send [:reflect] meta self
                (make-pseudo-cont self sender forward-to) 1 :send)
  ; Sinon réification véritable de l'acteur courant
  ...)
(let ((became? nil)) (default-become)))
```

avec

```
(defun make-pseudo-cont(actor sender forward-to)
  (make-mailbox :put-message
    #'(lambda (message)
        (default-send [:reflect] actor sender forward-to 1 :send))))
```

Il s'agit avec *actor-method-serialized-reflect* de la fonction correspondant à la forme compilée de la méthode :reflect d'un acteur sérialisé ayant le statut de méta-acteur. La réflexion de la partie supérieure de la tour est le résultat de la branche «alors» du **if** (en gris dans le code). Après avoir vérifié que le méta-acteur a bien un méta-acteur, une requête de réflexion est adressée à ce méta-méta-acteur avec comme continuation une pseudo-continuation. Fabriquée à l'aide de la fonction *make-pseudo-cont* qui la construit avec les éléments minimum qui lui permettent de recevoir et de traiter un message, cette continuation enverra à nouveau une requête de réflexion à l'acteur courant lorsqu'elle recevra une réponse du méta-méta-acteur lui signalant qu'il a été réfléchi. La deuxième exécution de la méthode :reflect se fera alors dans un contexte où la variable *meta* aura la valeur *nil*. C'est alors la deuxième branche du **if** qui correspond à la véritable réflexion qui sera exécutée.

La fonction que nous venons de décrire n'utilise pas les instructions du langage mais directement leur implémentation (*default-send*, *default-become*). La raison en est que les méthodes correspondant aux messages du méta doivent être insensibles à la modification éventuelle des définitions de ces instructions. Si elle entre en contradiction avec le protocole de LActE/R, elle assure ainsi une meilleure méta-stabilité de LActE/R.

Ce schéma de fonctionnement est identique quelle que soit la nature de la tour (tour basée sur un acteur, sérialisé ou non, ou encore sur un objet du langage comme les messages). Il nous assure que l'application de la véritable opération de réflexion a toujours lieu dans le même contexte, c'est à dire lorsque le méta-acteur qui la déclenche est au sommet de sa tour de méta-acteurs.

Contexte de l'opération de réflexion

La réflexion doit, pour permettre à l'acteur dénotation de retrouver son autonomie, être capable de définir parfaitement l'état courant de la machine implémentée par le méta-acteur. L'une des étapes les plus importantes de cette opération consiste donc à déterminer l'état exact de la machine qu'il faut réfléchir.

L'état de la machine acteur est défini par les valeurs des accointances du méta-acteur, mais aussi par l'activité de cette machine.

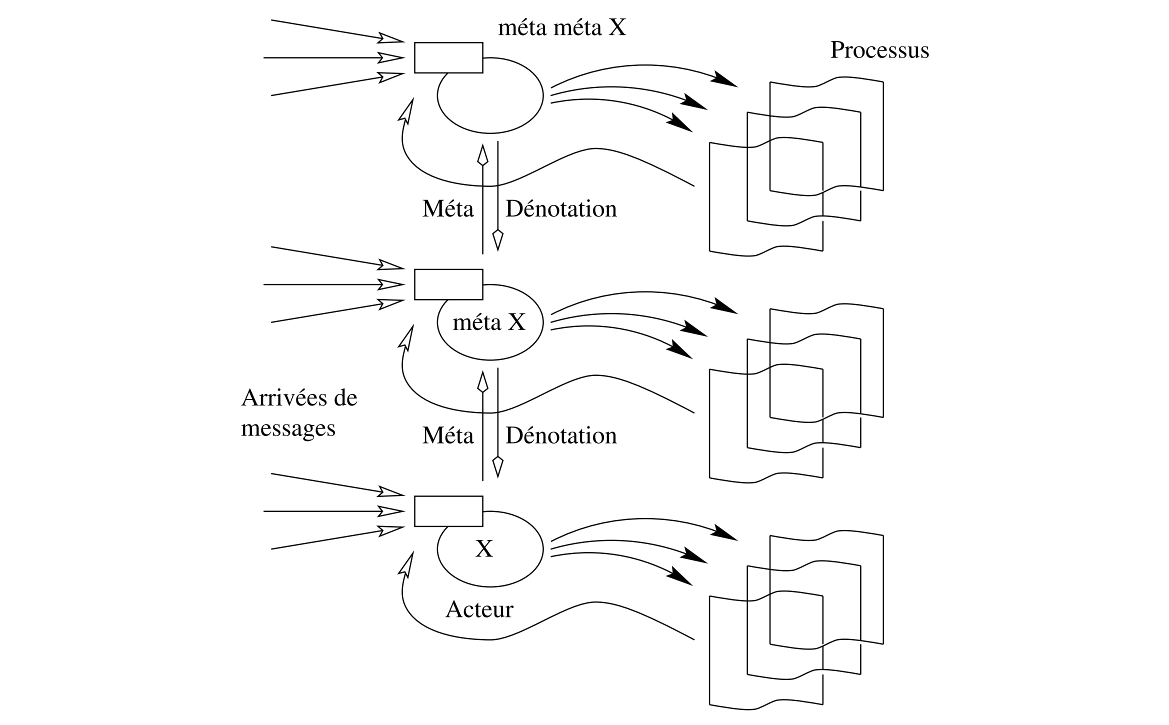

FIG. 8.4 – *Contexte d'une réflexion*

Lorsqu'un méta-acteur reçoit une requête de réflexion, son activité courante représente l'activité de la machine qu'il implémente (acteur, message, comportement ...). De par sa nature d'acteur, la modélisation qu'il en donne est nécessairement traduite par une activité concurrente (Fig. 8.4). Un exemple de cette activité concurrente est la traduction par la concurrence inhérente aux acteurs non sérialisés de la possibilité qu'un acteur non sérialisé a de recevoir et de traiter plusieurs messages en même temps. Or cette concurrence, si elle correspond à la concurrence naturelle du modèle acteur, rend difficile la définition de l'état exact de la machine ainsi implémentée. Pour reprendre le cas du méta-acteur d'un acteur non sérialisé, rien dans le système ne permet de savoir si cet acteur a des messages en cours de traitement ou non. En particulier, il peut avoir délégué cette tâche à un autre acteur qui, à l'instant du déclenchement de l'opération de réflexion, traite une dizaine de messages en parallèle. Le seul critère d'activité du méta-acteur, la présence de processus qu'il a généré, n'est donc pas suffisant pour affirmer que la machine acteur est active ou non.

La seule constante absolument sûre au niveau du comportement du méta-acteur est la nécessaire présence d'une méthode de réception des messages pour tous les types d'acteurs. La présence d'une méthode de changement de comportement dans le cas d'un acteur sérialisé n'est que probable [2]. La liberté que laisse la réflexivité au programmeur, alliée aux particularités du

2. Le protocole de LActE/R est tel qu'il est tout à fait possible de modifier les instructions correspondant à un changement d'état et de passer par d'autres méthodes que celles décrites dans le protocole pour implémenter le changement de comportement d'un acteur sérialisé.

modèle de calcul, interdit une implémentation de la réflexion qui passerait par une suppression immédiate et définitive du méta-acteur en se basant sur la terminaison d'un cycle de la machine acteur.

Cependant l'application de l'opération de réflexion impose d'autres contraintes sur le méta-acteur. En particulier, lors de cette opération, la présence des accointances décrivant l'état de l'acteur est requise. La réflexion va se baser sur la présence et sur la position de ces accointances afin de les réfléchir vers la dénotation. Il est donc impératif pour le bon fonctionnement de cette opération que ces accointances soient présentes parmi les accointances du méta-acteur et que leurs positions dans la liste des accointances de ce méta-acteur correspondent bien à celles définies dans le modèle de réflexivité.

Description générale de la réflexion

L'opération de réflexion à lieu dans un contexte assez complexe. La possibilité même de cette opération n'a de sens que dans la mesure où un ensemble minimum de contraintes est satisfait par le méta-acteur sujet à la réflexion. La réflexion doit cependant avoir lieu sans que l'état de la machine acteur ne soit complètement connu.

La solution adoptée pour LActE/R consiste à modifier le comportement et le statut du méta-acteur sans pour autant le supprimer réellement. L'état précis de la machine acteur n'a ainsi plus a être déterminé. Le méta-acteur continue d'assurer le fonctionnement des actions en cours d'exécution tout en permettant à sa dénotation de retrouver son autonomie. Le fonctionnement de la machine acteur est alors assuré et par le méta-acteur qui adopte un comportement strictement équivalent à celui de l'interprète LActE/R, et par le système pour les actions directement entreprises par la dénotation.

L'opération de réflexion dans LActE/R passe par une modification du comportement du méta-acteur. Elle s'apparente ainsi à un changement de comportement tel qu'il est effectué par un **become**. Le méta-acteur va adopter un comportement qui va lui permettre de répondre à l'ensemble des messages qu'il doit accepter s'il suit le protocole de LActE/R. Ce comportement identique à celui de l'interprète LActE/R le rend capable de travailler de façon cohérente avec cet interprète.

La modification du comportement du méta-acteur se traduit à deux niveaux. D'une part la fonction de réception de message interprète directement les messages que le méta-acteur reçoit en utilisant la puissance de calcul offerte par l'émetteur. Les surcoûts qu'entraînent la génération des processus ainsi que la gestion de la machine acteur du méta-acteur sont ainsi supprimés tout en conservant la possibilité de recevoir des messages pour le méta-acteur. En cela l'implémentation s'apparente au concept des acteurs légers mis en œuvre dans l'implémentation des objets du langage. D'autre part, le méta-acteur, qui conserve sa structure d'acteur, adopte un comportement spécifique qui reprend celui de l'interprète LActE/R. La présence de ce comportement est indispensable pour assurer le bon fonctionnement de l'éventuel méta du méta-acteur auquel l'opération de réflexion est appliquée. Sans sa présence, le caractère récursif de l'opération de réflexion n'est pas assuré et l'application de l'opération de réflexion ne pourrait s'effectuer que sur les méta-acteurs qui n'ont encore jamais eu eux même de méta-acteurs.

Le comportement du méta-acteur consiste désormais à traduire les messages qu'il reçoit vers l'interprète LActE/R. En particulier, tous les messages de type :receive−message voient réexpédier leur argument, le message qui doit être reçu par la dénotation, directement à la dénotation. Le méta-acteur opère par là une réflexion de ces messages vers sa dénotation qui pourra les prendre directement en charge. Les autres messages reçus par le méta-acteur sont traités de façon similaire.

Le méta-acteur ainsi équipé conserve la connaissance de sa dénotation. Mais alors en quoi cette dénotation redevient autonome ? La dénotation subit à son tour un certain nombre de transformations qui ont pour objectif de lui rendre son autonomie. La première de ces transformations consiste à la doter d'une fonction de réception des messages qui lui permette d'assurer son rôle d'acteur autonome. Pour cela le champ *put-message* de sa boîte aux lettres pointe à nouveau vers la fonction de réception par défaut qui ne réifie plus les messages qu'elle reçoit vers le méta-acteur. Selon le type d'entité qu'est cette dénotation, les différentes autres valeurs et fonctions sont réinitialisées avec leurs valeurs par défauts. En particulier, pour les acteurs sérialisés, les messages contenus dans la boîte aux lettres du méta-acteur doivent être réfléchis. Cette opération est réalisée par le réenvoi de l'ensemble des messages contenus dans la boîte aux lettres au méta-acteur. Ils seront alors interprétés par le méta-acteur ayant adopté le comportement équivalent à celui de l'interprète LActE/R. Pour finir, elle perd la connaissance de son méta dont elle se détache afin d'assurer elle-même son implémentation.

La dénotation reprend ainsi son autonomie. Son ex-méta-acteur continue d'interpréter les actions en cours de la machine qui l'implémente en s'appuyant désormais sur l'interprète LActE/R alors que la dénotation prend en charge tous les nouveaux cycles de sa machine acteur. Le lien causal n'est alors plus parfaitement respecté: le méta-acteur continue de disposer de la possibilité de modifier l'acteur qu'il gère directement, mais les modifications de l'acteur ne se traduisent plus systématiquement par des modifications du méta-acteur. Ce n'est que lorsqu'il aura besoin des données de la dénotation qu'il ira les chercher au sein même de la structure qui lui correspond. Ce fonctionnement reprend en fait directement celui de l'interprète LActE/R.

Après un certain temps, les actions de la machine acteur en cours au moment de la réflexion vont se terminer. Elles vont se terminer d'autant plus vite que l'implémentation qu'en donne désormais le méta-acteur ne génère pas d'autres messages comme il le faisait avec son comportement précédent. Une fois ces actions terminées, et dans la mesure où le méta-acteur n'est pas utilisé par d'autres acteurs que ceux de la tour réfléchie, la dénotation reprendra de fait sa complète autonomie assurant ainsi une efficacité en terme d'exécution égale à celle d'un acteur qui n'a subit aucune réification.

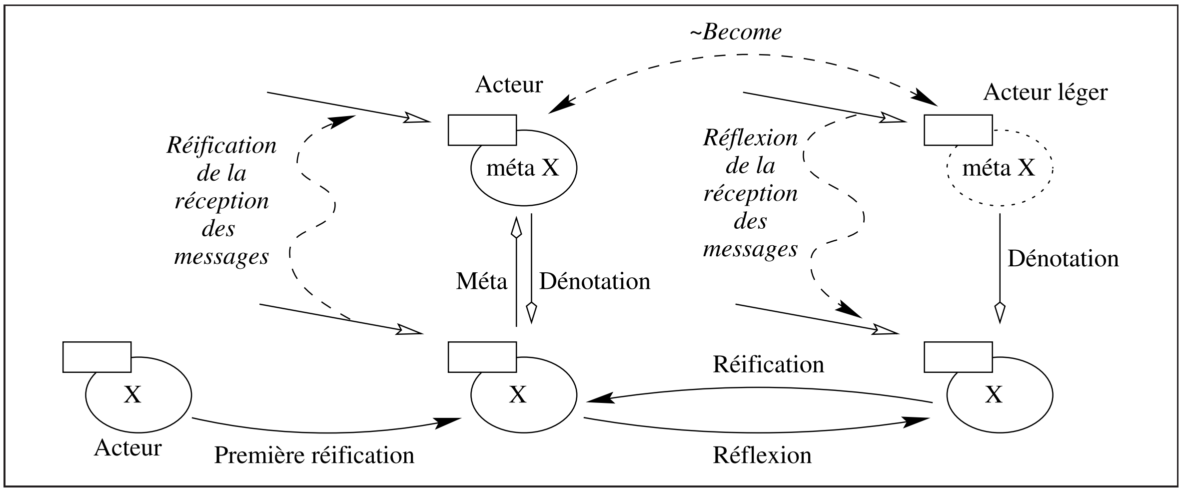

Fig. 8.5 – *Enchaînement des opérations de réflexion et de réification*

8.4.4 Enchaînement des opérations de réflexion et de réification

L'enchaînement immédiat d'une réification à la suite d'une opération de réflexion, c'est à dire la reconstruction d'une tour de méta-acteurs sur la base d'une tour qui vient d'être réfléchie, semble, à la vue de la description de la réflexion qui précède, être problématique. En effet, si la modification du méta-acteur lui permet de fonctionner en cohérence avec l'interprète LActE/R, elle n'assure a priori cette cohérence que dans le cadre d'une dénotation non réifiée. L'ajout d'un nouveau méta-acteur, comme le fait la première opération de réification, risquerait d'entraîner des incohérences provenant du fonctionnement en concurrence de l'«ex»-méta-acteur avec le nouveau méta-acteur. En effet, lors de la réification, si le méta-acteur nouvellement ajouté se substitue bien à l'interprète LActE/R, il ne se substitue pas à l'«ex»-méta-acteur qui continue de subsister.

L'«ex»-méta-acteur ne peut être supprimé ni lors de la réflexion, ni même juste avant la prochaine réification. Sa conservation est nécessaire pour assurer que des messages provenant d'autres acteurs participant au fonctionnement de la machine acteur dont il représentait le point d'entrée puissent être traités conformément au protocole. Le changement de comportement imposé lors de la réflexion est déjà en lui même une opération assez contraignante qui impose une certaine conformité du comportement du méta-acteur avec le protocole.

Or si le méta-acteur est conservé lors de la réflexion, c'est pour assurer le bon fonctionnement de la machine acteur dont l'état est inconnu lors de la réflexion. Une réification enchaînée après une réflexion pose les mêmes problèmes. Des messages provenant de la machine acteur qui a été réfléchie peuvent toujours continuer à arriver après une nouvelle réification. Cette nouvelle réification doit pallier à ce problème et éviter la mise en concurrence de deux méta-acteurs sur la même dénotation.

La solution mise en œuvre dans LActE/R est de donner une sémantique différente aux réifications qui suivent des réflexions. Si la première réification d'un objet de LActE/R construit un méta-acteur, les autres réifications procèdent de façon similaire à la réflexion en imposant un changement de comportement au méta-acteur conservé après une réflexion (Fig. 8.5 page précédente). La conservation du même méta-acteur lors de la construction et la reconstruction de la tour de méta-acteurs garantit que les messages engendrés pour le traitement de la machine acteur seront toujours traités par une même entité dont seul le comportement aura changé. L'écueil de la mise en concurrence de deux méta-acteurs sur la même dénotation est ainsi évité.

La substitution de la modification du comportement à la création du méta-acteur est la seule différence entre la première opération de réification et les suivantes. Il faut dans tous les cas reconstruire les différents éléments qui composent le lien causal, et en particulier modifier la fonction réalisant l'activité de réception des messages vers une fonction qui réifie tous les messages qu'elle reçoit.

Les différentes sémantiques de la réification sont facilement obtenues en gérant correctement le champ *reify-to* de la structure *meta-fct-descr* de chaque acteur. Il suffit de le faire pointer lors de la création d'un acteur vers la fonction qui réalise la première réification, pour ensuite, lors de chaque réflexion, actualiser ce champ pour le faire pointer vers la fonction réalisant les autres réifications. Un comportement différent est alors adopté pour la première des réifications d'un acteur de celui adopté pour les suivantes.

8.4.5 Réification et réflexion des autres objets de LActE/R

Dans les paragraphes qui précèdent, l'accent est mis sur la réification et la réflexion des acteurs laissant sous silence l'implémentation de ces opérations pour les autres objets de la couche LActE/R.

La réification et la réflexion des autres objets de LActE/R diffèrent peu de celles des acteurs. Les problèmes qu'elles soulèvent sont identiques sinon qu'ils portent sur des machines différentes. Le procédé mis en œuvre est par conséquent identique si ce n'est que les opérations d'accès aux structures décrivant ces objets doivent elles aussi être protégées par des pseudo-locks.

Outre cette spécificité des objets de LActE/R, les opérations de réification et de réflexion appliquées à ces objets suivent un schéma identique à celle des acteurs. Notons que ces objets se trouvent toujours à la base d'une tour.

8.4.6 Contraintes sur l'application des opérations de réification et de réflexion

L'implémentation des opérations de réification et de réflexion qui vient d'être décrite en montre les limites. Ces opérations ne fonctionnent que dans un cadre précis. En particulier, la position ainsi que la sémantique des accointances du méta-acteur doivent suivre de près le protocole.

Mais aussi la description de la mécanique acteur donnée par le protocole doit être respectée. Cette contrainte peut cependant être relâchée entre une réification et une réflexion pourvu qu'elle le soit au moment où ces opérations sont demandées.

L'utilisation des opérations de réification et de réflexion ne peut avoir lieu que dans le respect du modèle de réflexivité. Un cadre plus large soulèverait de nombreux problèmes difficilement solubles. La liberté que donne la réflexivité au programmeur n'est pas pour autant restreinte. Toutes les variations imaginables sont envisageables pour autant que les méta-acteurs trop éloignés du protocole ne soient pas sujet à une réflexion. Et, entre les opérations de réification et de réflexion, l'utilisateur dispose de toute la liberté souhaitée quant à la forme prise par ses méta-acteurs.

Lors de la réflexion, le respect de ces contraintes reste à la charge de l'utilisateur. Une vérification complète de la correspondance du positionnement ainsi que des valeurs des accointances du méta-acteur avec celle de l'objet qu'il dénote est impossible. LActE/R se contente de vérifier que le nombre des accointances est correct et que leurs valeurs sont compatibles avec celles attendues par la dénotation.

En acceptant les limitations imposées par ces opérations, le principe de localité temporelle reste entièrement valable. La plupart des modifications temporelles d'une tour de méta-acteurs rentrent dans le cadre du modèle de réflexivité et rendent possible l'application de ce principe.

8.5 Implémentation des objets de LActE/R

Les objets de LActE (messages, comportement, méthode, etc. ...) adoptent dans LActE/R un statut d'acteur tout en conservant une implémentation basée sur des structures spécifiques. Afin d'uniformiser les concepts à manipuler lors de l'utilisation des capacités réflexives, les différents éléments de ces objets sont accessibles à l'utilisateur par des envois de message. Cependant, pour des raisons d'efficacité, l'interprète LActE/R ne les manipule pas en utilisant des envois de message, mais en accédant directement aux champs des structures qui les implémentent. Les objets de la couche LActE/R s'apparentent donc à des Janus dont l'une des faces observe le monde des acteurs alors que la deuxième face dialogue avec le monde fonctionnel auquel appartient l'interprète.

L'implémentation des objets de la couche tient compte des deux mondes qui les manipulent. Tout objet est muni de deux interfaces: une interface avec le monde des acteurs et une interface avec le monde fonctionnel. Dans la suite de cette section, nous allons, à travers l'implémentation des messages, montrer comment ces deux mondes coexistent. L'implémentation est identique pour les autres objets de la couche aux détails propres à chaque objet près.

8.5.1 Interface avec le monde fonctionnel

L'interface des objets de la couche LActE avec le monde fonctionnel offre à l'interprète LActE/R un moyen efficace de gérer les structures représentant ces objets. Elle fournit un accès fonctionnel aux champs de ces structures tout en gardant la possibilité de modifier ces fonctions d'accès vers des envois de message lorsque l'objet est réifié vers un méta.

L'implémentation des messages dans LActE/R, comme celle de l'ensemble des objets de la couche LActE, passe par une structure spécifique héritée de LActE. Son adéquation avec les contraintes imposées par l'introduction de la réflexivité est assurée par l'inclusion de la structure *object-mailbox* qui réunit les éléments communs aux objets. Plus précisément, cette dernière structure regroupe les champs utilisés par la partie réflexive de l'interprète tant au niveau de la gestion des tours de méta-acteurs qu'à celui de l'uniformité linguistique introduite par le modèle de réflexivité.

La structure qui implémente les messages à la forme suivante:

```
(defstruct (message
                 (:print-function print-message)
                 (:include object-mailbox
                        ; Interface avec les acteurs
                        (put-message #'default-message-put-message)))
           (message   '())
           (to        nil)
           (lock      (make-process-lock))
           (sender    nil)
           (forward-to nil)
           (type      :send)
           (take      1))
```

avec

```
(defstruct (object-mailbox
                 ; Inclusion de mailbox = interface avec les acteurs
                                (:include mailbox))
           (name       nil)
           (meta       nil)
           (meta-calls nil)     ; Interface fonctionnelle
           (proxy      nil))
```

En supprimant de la définition de *message* l'inclusion de la structure *object-mailbox*, nous obtenons sa définition originelle telle qu'elle existe dans la version non réflexive de LActE. Les éléments additionnels nécessaires à la gestion de la réflexivité se trouvent réunis dans la définition de *object-mailbox*. A l'exception du champ *name* qui appartient aux deux versions de LActE, les champs de *object-mailbox* décrivent les aspects d'un message qui relèvent soit de l'uniformité linguistique, les champs en provenance de *mailbox* et le champ *proxy*, soit de la tour de méta-acteurs, le champ *meta*, soit de l'interface fonctionnelle, le champ *meta-calls*. C'est ce dernier champ qui va être l'objet de notre attention dans la suite de cette section.

Lors de la définition d'une structure, Common Lisp génère automatiquement les fonctions nécessaires à sa manipulation et, en particulier, les fonctions d'accès aux champs en lecture et en écriture. Il existe une correspondance immédiate entre les fonctions d'accès aux champs et les points d'entrées du protocole. L'envoi d'un message **:get-message** à un message et la fonction *message-message* appliquée à la structure qui représente ce message retournent la même valeur. La différence entre ces deux formes d'accès à une même structure réside dans l'identité de leur manipulateur: l'utilisateur pour la première et l'interprète LActE/R pour la seconde.

Le propos de l'interface fonctionnelle est de factoriser sous la seule fonction d'accès à un champ le retour de la valeur correspondante quelle que soit la nature et le statut du message. L'interprète utilise systématiquement la fonction *message-message* pour obtenir le contenu d'un message que ce message soit un message non réifié, un message réifié ou même un acteur. Si dans le premier cas l'accès à la valeur peut se faire en utilisant la fonction générée automatiquement par Common Lisp d'accès au champ *message*, les deux derniers cas nécessitent un envoi de message vers le méta-acteur ou vers l'acteur qui joue le rôle d'un message. La prise en compte de ces différents cas n'est pas obtenue par une modification de l'interprète LActE, mais par une modification des fonctions d'accès. Toute fonction d'accès à un champ en lecture ou en écriture est modifiée pour tenir compte des différents cas possibles. Par exemple, la fonction *message-message* d'accès au contenu d'un message est modifiée de la façon suivante:

```
; Version modifiée de l'accès au champ message d'un message
(setf (symbol-function 'message-message)
      #'(lambda (message)
          (cond ((message-p message)
                 (process-lacte-lock-might-lock
                   (meta-fct-descr-lock
                     (message-meta-calls message)))
                 (funcall (message-fct-get-message
                            (message-meta-calls message)) message)))
                ((actor-mailbox-p message)
                 (default-ask '(:get-message) message self 1))
                (t (error ...)))))
```

La fonction *message-message* agit différemment selon que son paramètre est un message au sens de LActE ou un acteur. Dans ce dernier cas, elle demande le contenu du message à l'aide d'un envoi synchrone du message [:get−message]. Dans le cas d'un véritable message, elle utilise la fonction correspondante contenue dans le champ *meta-calls* de la structure message qu'elle appelle après avoir posé un lock qui en protège la modification. Ce champ contient une autre structure basée sur la structure *meta-fct-descr* et dont chacun des champs correspond à l'une des fonctions d'accès aux champs de la structure message reconnus par le protocole. Cette structure nommée *message-fcts* adopte la définition suivante:

```
; Structures des pointeurs sur les fonctions d'accès aux champs
(defstruct (message-fcts
             (:include meta-fct-descr
                  (reify   #'message-default-reify)
                  (reflect #'default-reflect)))
            (get-message   #'default-message-message)
            (get-to        #'default-message-to)
```

```
(get-sender     #'default-message-sender)
(get-forward-to #'default-message-forward-to)
(get-type       #'default-message-type)
(get-take       #'default-message-take)
(set-to         #'default-message-set-to))
```

Elle inclut *meta-fct-descr* qui contient les pointeurs sur les fonctions qui implémentent les opérations réflexives pour l'objet qu'elle représente. Les champs additionnels contiennent les pointeurs sur les fonctions d'accès aux champs. Par exemple *get-message* pointe sur *default-message-message* qui est la fonction d'accès au champ *message* d'un message. Cette fonction est une fermeture lexicale qui capture la définition générée automatiquement par Common Lisp qu'elle appellera pour obtenir la valeur du champ *message*. Cette indirection permet le contrôle du lock posé par la fonction *message-message*. Le lock n'est pas levé par cette dernière. La définition de *default-message-message* est la suivante:

```
(setf (symbol-function 'direct-message-message) #'message-message)
```

```
(defun default-message-message(message)
  (let ((result (funcall direct-message-message message)))
     (process-lacte-lock-might-unlock
       (meta-fct-descr-lock (message-meta-calls message)))
     result))
```

Grâce à l'utilisation d'une structure contenant les pointeurs sur les fonctions réalisant l'accès aux différents champs de la structure qui représente un objet de LActE, la modification de ces pointeurs est facilement réalisable. Lors de la réification d'un message, le méta-message implémente alors le message. L'accès aux champs ne se fait alors plus directement mais indirectement par l'intermédiaire du méta. Dans ce cas le champ *get-message* de *message-fcts* pointe vers la fonction suivante:

```
; Version réifiée de l'accès au champ message d'un message
(defun message-reified-get-message(message)
  (declare (type message message))
  (process-lacte-lock-might-unlock
    (meta-fct-descr-lock (message-meta-calls message)))
  (default-ask '(:get-message) (message-meta message) self 1))
```

La fonction *message-reified-get-message* envoie sous forme synchrone une demande d'accès au champ correspondant directement au méta-message. Le lock qui protège la modification des fonctions d'accès doit être levé avant l'envoi de message. Dans le cas contraire, un blocage du système pourrait se produire.

8.5.2 Interface avec le monde des acteurs

Le deuxième monde avec lequel un objet du langage dialogue est celui des acteurs. L'interface avec le monde des acteurs dote les objets du langage de la capacité de recevoir et de traiter des messages.

Les objets du langage sont implémentés par des structures dédiées et suivent des mécanismes de fonctionnement différents de ceux des acteurs. Les modalités du traitement des messages reçus sont donc propres à chaque type d'objet. Une fonction de réception des messages est donc définie pour chaque objet du langage.

Les fonctions de réception des messages des objets du langage doivent assurer l'intégralité du traitement des messages reçus. Contrairement aux acteurs qui utilisent des comportements pour définir les actions qui correspondent à chaque message reçu, les objets du langage utilisent un modèle plus léger dans lequel l'ensemble des fonctionnalités relatives au traitement des messages sont intégrées dans la fonction de réception des messages. Les mécanismes mis en jeu ne génèrent plus de processus comme pour les acteurs mais utilisent la puissance de calcul de l'émetteur du message pour en calculer la réponse. Le coût d'une communication avec les objets du langage est ainsi diminué par un traitement direct plus proche des mécanismes fonctionnels que des mécanismes acteurs.

Une telle approche n'est possible que parce que les protocoles d'accès aux objets du langage sont définitifs et non modifiables par l'utilisateur. Seule la réification vers un méta-acteur spécialisé permet de le modifier. Mais ce n'est plus l'objet qui répond aux messages mais son méta.

Une autre particularité favorise cette approche: les messages envoyés aux objets du langage correspondent à des requêtes dont la réponse peut être directement et rapidement calculée. Dans la quasi totalité des cas ces requêtes demandent la valeur de l'un des champs de la structure qui implémente l'objet. La réponse ne dépend alors que du contenu du champ et non du calcul effectué par tout un système d'acteurs.

Pour obtenir un tel comportement chaque objet dispose de sa propre fonction de réception des messages. Les messages non réifiés utilisent la fonction suivante:

```
(defun default-message-put-message(mesg self message)
  (declare (type message self))
  (cond ((and (listp message)
              (eq (length message) 1))
         (let* ((delegate? nil)
                (error     nil)
                (result
                  (case (first message)
                    (:get-values
                      (list (direct-message-to self)
                            (direct-message-sender self)
                            (direct-message-forward-to self)
                            (direct-message-message self)
                            (direct-message-take self)
                            (direct-message-type self)))
                    (:get-type
                      (list (direct-message-type self)))
                    (:get-to
                      (list (direct-message-to self)))
                    (:get-sender
```

```
                        (list (direct-message-sender self)))
                      (:get-forward-to
                        (list (direct-message-forward-to self)))
                      (:get-take
                        (list (direct-message-take self)))
                      (:get-message
                        (list (direct-message-message self)))
                      (:meta '(nil))
                      (:denotation '(nil))
                      ...)
              (let ((proxy (message-proxy self)))
                (process-lacte-lock-might-unlock
                  (mailbox-put-message self))
                (when error?
                  (default-send error? error self self 1 :send))
                (cond (delegate?
                        (cond (proxy (setf-message-to mesg proxy)
                                     (send-message mesg))
                              (t (default-send (list :error ...)
                                               error self self 1 :send))))
                      (t (default-send result
                                       (message-forward-to mesg)
                                       self self 1 :send))))))
          ...)
```

La plupart des requêtes envoyées à un message correspondent à des opérations sur les champs de la structure qui le représente. Dans le cas d'un :get−sender, la fonction utilise *direct-message-sender* pour accéder à la valeur de son émetteur. L'accès au champ *sender* se fait directement.

Lorsqu'un message est réifié, sa fonction de réception des messages doit être modifiée afin de réifier vers son méta-acteur les requêtes qui lui sont adressées. La fonction de réception des messages d'un message réifié a la forme suivante:

```
(defun reifyied-message-put-message(mesg self message)
  (if (meta-message-pattern? message)
    (case (first message)
      (:meta
        (let ((result (list (message-meta self))))
          (process-lacte-lock-might-unlock
                    (mailbox-put-message-lock self))
          (default-send result
                    (message-forward-to mesg self self 1 :send)))
      (:denotation
        (process-lacte-lock-might-unlock
                    (mailbox-put-message-lock self))
        (default-send '(nil) (message-forward-to mesg)
                    self self 1 :send))
        ...)
    (let ((meta (message-meta self)))
      (process-lacte-lock-might-unlock
                  (mailbox-put-message-lock self))
```

```
        (setf-message-to mesg meta)
        (send-message mesg)))))
```

Seuls les messages du méta continuent d'être directement interprétés par la fonction de réception des messages. Les autres requêtes sont déroutées vers le méta-acteur qui a désormais la charge de leur interprétation.

8.5.3 Les objets du langage vus comme des acteurs légers

Les objets du langage munis de leurs deux interfaces s'assimilent à des acteurs légers. Tout en conservant les mêmes fonctionnalités que les acteurs, leur implémentation est plus légère et plus efficace.

Des tests montrent que l'interface fonctionnelle est 19 fois plus rapide que l'interface acteur et 132 fois plus rapide que les messages implémentés par des acteurs. Ces chiffrent très significatifs démontrent l'intérêt de l'approche utilisée pour implémenter les objets de LActE.

8.6 Implémentation des instructions

Le triplet (instructions, réificateurs, méta-instructions) est l'outil fourni par LActE/R pour manipuler les instructions introduites par la couche acteur. Il permet la définition de nouvelles instructions et la construction de réificateurs qui précisent si et quelles instructions sont réifiées et vers quelles méta instructions. Cette modélisation restreinte aux éléments de la couche LActE de l'interprète est l'une des clefs de l'efficacité de LActE/R: par la restriction des possibilités de réification aux seules instructions de la couche acteurs, le surcoût d'une approche «tout interprété» est évité.

Le constructeur **def-instruction** de LActE/R offre à l'utilisateur la possibilité de définir de nouvelles instructions mais aussi de redéfinir les instructions de LActE. Les six paramètres de **def-instruction** spécifient toutes les caractéristiques d'une instruction:

- Sa syntaxe i.e. l'ensemble des paramètres acceptés par l'instruction.

- Sa forme compilée, une expression Lisp évaluée dans le contexte lexical de l'appel de l'instruction qui permet non seulement de spécifier comment l'instruction appelle son implémentation (à l'aide du **call-implementation**), mais aussi de récupérer des informations lexicales.

- Sa lambda-liste i.e. les paramètres passés à l'implémentation ou à la forme réifiée.

- L'adresse de la fonction Lisp qui implémente l'instruction.

- La forme réifiée de l'instruction i.e. la forme du message passé à la méta-instruction d'une instruction réifiée.

Le constructeur **def-instruction** est implémenté par une macro qui a la forme suivante:

```
; Définition des instructions
(defmacro def-instruction(name &key syntax compilation
                               lambda-list implementation
                               reified-form)
          ; Obtention du numéro de l'instruction
  `(let* ((inst-nb (new-instruction-number))
          ; Construction de la structure conservant l'ensemble des
          ; informations relatives à l'instruction
          (inst (make-instruction :name ',name
                                  :syntax ',syntax
                                  :compilation ',compilation
                                  :lambda-list ',lambda-list
                                  :implementation ,implementation
                                  :reified-form ',reified-form
                                  :number inst-nb
                                  :meta-instruction
                                    (make-meta-instruction
                                       ,lambda-list ,reified-form))))
     (format t "~&Defining instruction no ~a: ~a." inst-nb ',name)
     ; Ajout de l'instruction dans la liste des instructions
     (with-process-lock ((*all-instructions-lock*))
          (setf *all-instructions* (cons inst *all-instructions*)))
      ; La structure contenant les informations relatives
      ; à l'instruction est ajoutée à la propriété instruction
      ; de la P-liste du symbole qui nomme l'instruction
     (push inst (get ',name 'instruction))
      ; Le réificateur par défaut est mis à jour: il doit
      ; contenir un pointeur vers l'implémentation
     (setf (svref *default-reifier* inst-nb) ,implementation)
      ; La macro qui définit l'appel à l'instruction est
      ; construite et retournée comme résultat. Elle est
      ; liée à la F-val du symbole qui nomme l'instruction
     (build-macro ',name ',syntax
              (process-call-implementation inst-nb ',compilation))))
```

Le constructeur **def-instruction** commence par demander au système de lui fournir un numéro unique pour l'instruction à définir. À chaque instruction est affecté un numéro unique qui permet son identification indépendamment de son nom. Ainsi, la redéfinition d'une instruction ne supprime pas l'ancienne définition qui reste utilisée par les méthodes non recompilées, les deux versions d'une même instruction ayant chacune un numéro différent.

La seconde étape de la construction d'une instruction consiste à créer une structure contenant les différentes informations nécessaires à son fonctionnement. Il faut distinguer les informations utilisées pour le fonctionnement de l'instruction de celles conservées à des fins de consultation par l'utilisateur. Les informations utilisées pour le seul fonctionnement d'une instruction sont son numéro d'identification, l'adresse de son implémentation et le constructeur contenu dans le champ *meta-instruction*. Les autres champs ne servent qu'à conserver les informations consultées par l'utilisateur.

Cette structure est alors ajoutée dans la liste des instructions ainsi que dans la P-liste du symbole qui nomme l'instruction. Le système accède aux différentes informations relatives à l'instruction à travers la propriété *instruction* de cette P-liste. Les définitions successives d'une même instruction se trouvent regroupées dans la liste contenue dans cette propriété. L'argument *which* de *get-instruction* offre un accès aux différentes versions d'une même instruction.

Le réificateur par défaut est mis à jour. Un réificateur est un simple vecteur de fonctions. Le premier élément de ce vecteur sert à la conservation d'informations: il contient un vecteur de listes constituées de l'adresse de la méta-instruction et du rang de l'instruction réifiée i.e. un numéro qui distingue les différentes versions d'une même instruction. Les autres éléments du réificateur contiennent des pointeurs vers la fonction qui implémente l'instruction ou, dans le cas d'une instruction réifiée, une fermeture lexicale qui effectue la demande d'interprétation de l'instruction à la méta-instruction à l'aide d'un envoi synchrone de message.

Au moyen d'une simple indirection, le réificateur permet de combiner instructions réifiées et non réifiées et limite le coût de l'introduction de la réflexivité à une indirection. La mise à jour du réificateur consiste donc à donner à l'élément du vecteur correspondant au numéro de l'instruction la valeur du pointeur sur son implémentation.

La dernière étape est la construction de la macro qui représente le point d'appel d'une instruction. Toute instruction est vue par le système depuis la macro qui l'implémente. Elle est construite à partir de la syntaxe qui en fournit les paramètres, et de la compilation qui en constitue le corps. L'utilisation d'une macro réduit le coût de l'appel d'une instruction (elle est expansée lors de la compilation des méthodes) et autorise la récupération d'informations lexicales qui seraient perdues avec une fonction.

Le modèle de réflexivité de LActE/R n'autorise que la construction d'instructions et de réificateurs mais pas leur modification. Le système n'a pas par conséquent à gérer les nombreux locks qu'aurait nécessités la possibilité de modifier les instructions ou les réificateurs. L'implémentation est ainsi grandement simplifiée.

Grâce au triplet (instruction, réificateur, méta-instruction), le coût de la réification des instructions de LActE est négligeable. Dans la plupart des cas, ce coût se limite au niveau de l'exécution à une simple indirection vers la fonction qui implémente l'instruction. Les instructions réifiées font appel à un envoi de message synchrone vers la méta-instruction. Le coût d'une telle opération dépend essentiellement de la méta-instruction.

8.7 Conclusion

L'efficacité de l'exécution dans LActE/R est le résultat d'un modèle ad hoc et d'une implémentation qui minimise la présence des constructions réflexives lorsqu'elles ne sont pas utilisées. En particulier, l'implémentation de LActE/R ne repose pas sur la présence d'un interprète méta-circulaire. L'utilisation intensive de l'approche fonctionnelle et la minimisation du nombre de processus générés ont conduit à un interprète plus efficace que ceux traditionnellement utilisés. L'opposition avec les modèles d'implémentation de la première partie est significative. Alors que notre premier interprète basé sur une tour active a des performances qui le rendent presque inutilisable, et que le second interprète, plus performant au niveau de l'exécution, nécessite de longues phases de compilation, LActE/R concilie favorablement les problèmes de performances avec un monde qui reste très dynamique.

Nous pensons donc que ce type d'approche reste intéressant pour l'avenir des systèmes réflexifs, l'évaluation partielle se traduisant par une perte des aspects dynamiques du système.

Chapitre 9

LActE/R en action

Ce chapitre illustre à travers trois exemples l'utilisation des capacités réflexives de l'interprète LActE/R. Il s'agit avant tout de démontrer les aptitudes du modèle de réflexivité de LActE/R à traiter un certain nombre de problèmes. Mais il s'agit aussi de montrer quelles peuvent être les applications et les apports de la réflexivité.

9.1 Introduction d'une capacité à mémoriser

La capacité d'une fonction à mémoriser ses résultats est un principe utilisé dans les langages fonctionnels afin d'éviter la répétition de calculs lourds. Nous nous proposons d'adapter, à l'aide de la réflexivité, ce mécanisme aux acteurs.

Dans les langages fonctionnels basés sur le passage des paramètres par valeur, le mécanisme de mémorisation peut être implémenté en enveloppant la fonction concernée dans une fonction chargée de mémoriser les couples (*valeurs d'appel*, *résultat*) et de court-circuiter les calculs dont le résultat a déjà été mémorisé.

Le dynamisme inhérent au monde des acteurs disqualifie l'approche fonctionnelle. Selon le modèle acteur, le calcul résulte de la collaboration de plusieurs acteurs. L'acteur qui reçoit la requête doit être considéré comme le point d'entrée d'un système d'acteurs dont la topologie, a priori inconnue, peut évoluer dynamiquement (Fig. 9.1 page suivante). En particulier, l'entité qui retournera la réponse n'est pas connue. Par contre, le message qui constitue la requête contient une valeur qui représente l'adresse de l'entité en attente du résultat. Le processus de mémorisation ne doit s'appuyer que sur les trois constantes connues: l'acteur récepteur de la requête, la requête, ainsi que l'adresse où retourner la réponse, i.e. la continuation du calcul.

Notre solution s'appuie sur les trois constantes précédemment identifiées. Elle consiste à placer une indirection sur la continuation initiale avant même que le calcul demandé ne soit lancé. L'indirection a alors la charge de capturer le résultat qu'elle envoie à la continuation initiale et au méta-acteur de l'acteur récepteur de la requête afin qu'il le mémorise (Fig. 9.2 page suivante). Ce méta-acteur a une deuxième fonction, celle de filtrer les messages reçus par l'acteur et de retourner immédiatement le résultat des requêtes qui correspondent à des

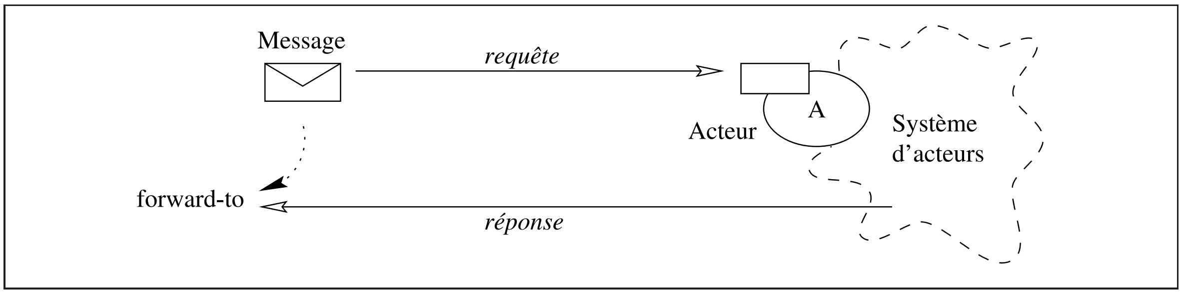

FIG. 9.1 – *Cheminement d'une requête et de son résultat*

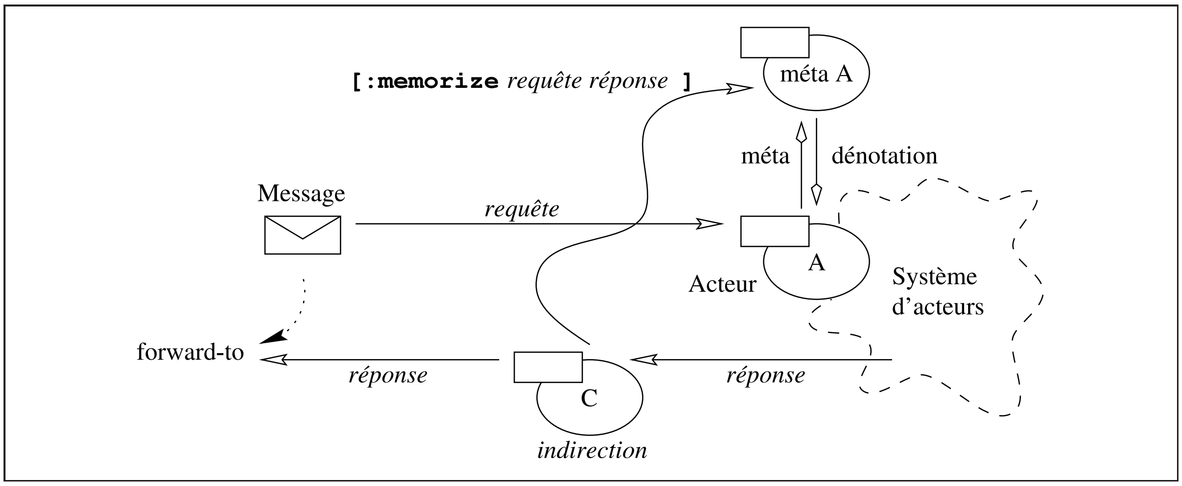

FIG. 9.2 – *Méta-acteur et mémorisation d'un résultat*

calculs déjà effectués. Le comportement de ce méta-acteur est:

```
(def-behavior a-memorize-result (results) t
  (declare (is-a a-meta-actor-unserialized))
  ; Mémorisation des résultats
  ((:memorize message-content the-result)
   (setf results (acons message-content the-result results))
   (update))
   ; Filtrage des messages
  ((:receive-message a-mesg)
   (become)
   (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg
                                      mesg-take mesg-type)
                        (ask [:get-values] :to a-mesg)
     ; Les messages du méta ne sont pas mémorisés (:meta, :denotation, ...)
     (if (meta-message? mesg)
        (send [:activate-message a-mesg] :to self)
        (let ((known-result (assoc mesg results :test #'equal)))
          ; Si le résultat est connu, il est directement retourné
          (if known-result
             (send [(cdr known-result)] :to mesg-forward-to)
             ; Sinon le calcul est lancé avec un forward-to modifié
             (send [:activate-message
```

```
(new-anonymous an-actor-message ()
               mesg-to mesg-sender
               ; Construction du forward-to
               (continuation ((mesg mesg)
                              (the-meta self)
                              (reply-to mesg-forward-to))
                 ((value)
                  (send [:memorize mesg value] :to the-meta)
                  (send [value] :to reply-to)))
mesg mesg-take mesg-type)] :to self)))))))
```

Le fonctionnement du méta-acteur *a-memorize-result* est restreint aux acteurs non sérialisés. Le mécanisme de mémorisation des résultats ne s'applique qu'aux calculs dont le résultat ne varie pas dans le temps. Seuls les acteurs non sérialisés offrent une certaine garantie du caractère constant de leurs résultats. Ils peuvent cependant faire appel à des acteurs dont le comportement varie dans le temps. Comme dans les langages fonctionnels[1], il n'y a pas dans les langages d'acteurs de moyen de garantir la constance des résultats. C'est à l'utilisateur de déterminer si une fonction ou un acteur donnera toujours le même résultat pour un même jeu de paramètres.

Outre l'accointance *results* qui contient l'ensemble des couples (*requête*, *résultat*) sous la forme d'une A-liste, *a-memorize-result* se distingue d'un méta-acteur de base d'un acteur non sérialisé par deux méthodes. La méthode :memorize sert à la mémorisation des couples (*requête*, *résultat*) qu'elle stocke dans l'accointance *results*. Notons qu'elle utilise l'instruction **update** afin de prendre en compte la nouvelle valeur de *results* et de donner au méta-acteur la possibilité de traiter un autre message. L'inclusion de comportement mise en œuvre pour simplifier l'écriture de notre comportement ne fait pas apparaître l'ensemble des accointances dont il faudrait spécifier la valeur lors de l'utilisation d'un **become** avec paramètres.

La seconde méthode est une modification du mécanisme de réception des messages qui se charge désormais de mettre en place les éléments nécessaires à la mémorisation ou de retourner directement le résultat d'un calcul déjà effectué. À la réception d'un message, trois cas sont possibles:

- Le message est un message du méta. Un tel message ne doit pas être mémorisé dans la mesure où les données relatives à la relation méta/dénotation peuvent être modifiées. De plus, le calcul qu'il nécessite ne justifie pas l'utilisation d'un mécanisme de mémorisation.

- La requête reçue correspond à un calcul déjà effectué. Le résultat conservé dans l'accointance *results* est extrait et directement envoyé à la continuation.

- La requête constitue une demande d'un calcul qui n'a encore jamais été effectué. Le message est reconstruit avec une continuation modifiée. La nouvelle continuation prend

1. Les langages purement fonctionnels i.e. totalement dépourvus d'effet de bord sont plus à même d'offrir une telle garantie.

en charge la réception du résultat, son émission vers la continuation initiale et sa mémorisation par le méta-acteur.

La réification d'un acteur non sérialisé vers un méta-acteur adoptant notre comportement lui permet de mémoriser l'ensemble des résultats des calculs qui lui sont demandés. En prenant une factorielle comme exemple, le calcul initial est notablement plus lent que celui effectué par la même factorielle mais sans méta-acteur (environ 5 fois plus lent). Mais, une fois ce calcul initial effectué, le calcul de la même factorielle est immédiat. L'utilisation du mécanisme de mémorisation ne se justifie donc que dans la mesure où le même calcul est demandé plusieurs fois (plus de 5 fois pour commencer à gagner du temps).

L'algorithme utilisé par *a-memorize-result* n'est cependant pas totalement satisfaisant: de substantielles améliorations sont possibles dans le cas de certains calculs. C'est le cas pour le calcul parallèle des nombres de Fibonacci. La suite de Fibonacci est définie par:

$$U_n = U_{n-1} + U_{n-2}$$

avec

$$U_0 = U_1 = 1$$

Cette suite peut être implémentée de plusieurs façons allant d'un déroulement séquentiel des opérations à différents algorithmes parallèles. L'expression la plus directe de la définition de la suite de Fibonacci en LActE/R est :

```
(def-behavior a-fibonacci () nil
  ((0) (result 1))
  ((1) (result 1))
  ((n)
   (let ((join (jointure ((reply-to forward-to))
                         ((value-1) (value-2))
                  (send [(+ value-1 value-2)] :to reply-to))))
      (send [(− n 1)] :to self :forward−to join)
      (send [(− n 2)] :to self :forward−to join))))
```

```
(new fibonacci a-fibonacci ())
```

Le calcul s'effectue parallèlement en demandant en même temps à l'acteur *fibonacci* de calculer les valeurs de U_{n-1} et de U_{n-2} et de retourner leurs résultats respectifs à une même entité qui prend ici la forme d'une *jointure*. Elle fonctionne à la manière d'une continuation qui attendrait l'arrivée de plusieurs messages, (*value-1*) et (*value-2*), avant d'évaluer son corps. Le corps de la jointure effectue alors la somme des valeurs de U_{n-1} et de U_{n-2} dont il envoie le résultat à la continuation initiale.

Le calcul parallèle d'un nombre de Fibonacci suit la forme d'un arbre binaire dont les nœuds sont constitués par les calculs des U_i et les branches par les demandes de calcul des

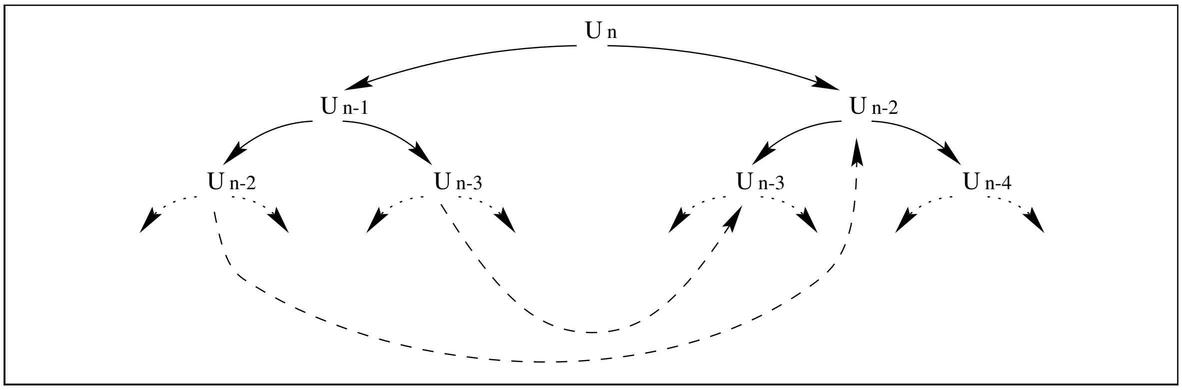

FIG. 9.3 – *Calcul de fibonacci avec la mémorisation*

U_{i-1} et U_{i-2}. La figure (Fig. 9.3) qui représente ce calcul fait aussi apparaître des branches identiques qui soulignent la répétition de certains calculs.

L'utilisation du mécanisme de mémorisation ne supprime pas ces redondances. Selon le déroulement du parallélisme, certaines valeurs seront déjà mémorisées et certains calculs inutiles évités. Mais il est plus probable que le méta-acteur passe son temps à stocker plusieurs fois les résultats de calculs identiques lancés en même temps.

Le fonctionnement du méta-acteur peut être modifié pour éviter les redondances de calculs. Il nous faut pour cela garder trace des calculs demandés et ne pas activer un calcul lorsqu'un calcul identique est déjà en cours. Dans ce dernier cas, l'indirection sur la continuation de la requête initiale est notifiée de cette requête supplémentaire. Lorsqu'elle recevra le résultat escompté, elle le transmettra à l'ensemble des continuations en attente du résultat.

Le comportement de l'indirection devient plus complexe:

```
(def-behavior a-result-memorize (mesg memorizer reply-to the-result is-result?) t
  ((:add−reply reply-address)
   (cond (is-result?
            (become)
            (send [the-result] :to reply-address))
         (t
            (become a-result-memorize () mesg memorizer
                    (cons reply-address reply-to) the-result is-result?))))
  ((value)
   (become a-result-memorize () mesg memorizer reply-to value t)
   (send [:memorize mesg value] :to memorizer)
   (multiple-send [value] :to reply-to)))
```

Le comportement *a-result-memorize* est le nouveau comportement adopté par les indirections sur les continuations initiales. Il est étendu pour traiter le cas ou plusieurs continuations sont en attente du résultat d'un même calcul. En particulier, une méthode supplémentaire est introduite afin de pouvoir augmenter la liste de ces continuations tout en tenant compte de la disponibilité d'un résultat. Lors de la réception du résultat, ce dernier, accompagné du contenu de sa requête, est comme avant envoyé au méta-acteur. En plus de cela, le résultat est envoyé à la liste des continuations en attente du résultat.

Le comportement du méta-acteur est désormais le suivant:

```
(def-behavior a-memorize-result (results computing-results) t
  (declare (is-a a-meta-actor-unserialized))
  ; Mémorisation des résultats
  ((:memorize message-content the-result)
   (setf results (acons message-content the-result results)
         computing-results (remove message-content computing-results
                                   :test #'equal :key #'car))
   (update))
  ; Filtrage des messages
  ((:receive-message a-mesg)
   (multiple-value-bind (mesg-to mesg-sender mesg-forward-to mesg
                         mesg-take mesg-type)
                        (ask [:get-values] :to a-mesg)
     ; Les messages du méta ne sont pas mémorisés (:meta, :denotation, ...)
     (if (meta-message? mesg)
       (progn
         (become)
         (send [:activate-message a-mesg] :to self))
       (let ((known-result (assoc mesg results :test #'equal)))
         ; Si le résultat est connu, il est directement retourné
         (if known-result
           (progn
             (become)
             (send [(cdr known-result)] :to mesg-forward-to))
           ; Si le même calcul est en cours, ajout du forward-to dans la
           ; liste des entités en attente du résultat
           (let ((computing? (assoc mesg computing-results :test #'equal)))
             (if computing?
               (progn
                 (become)
                 (send [:add-reply mesg-forward-to] :to (cdr computing?)))
               ; Construction de l'indirection et activation du message modifié
               (let ((memoizer (new-anonymous a-result-memorize ()
                                 mesg self (list mesg-forward-to) nil nil)))
                 (setf computing-results (acons mesg memorizer computing-results))
                 (update)
                 (send [:activate-message
                        (new-anonymous an-actor-message ()
                                       mesg-to mesg-sender
                                       ;Modification du forward-to
                                       memorizer
                                       mesg mesg-take mesg-type)] :to self)))))))))))
```

Une accointance supplémentaire, *computing-results*, fait son apparition dans le comportement *a-memorize-result* du méta-acteur. Elle contient la liste des requêtes en cours de traitement. Lors de la réception d'un message, le méta-acteur tient compte de cette liste afin de ne pas activer de calculs identiques à ceux déjà en cours. Dans le cas d'un calcul déjà en cours, il demande à l'indirection d'ajouter la nouvelle continuation à l'ensemble des continuations en attente du résultat qu'elle gère.

L'utilisation de ce méta-acteur avec l'acteur Fibonacci se traduit par une amélioration de l'efficacité: le temps de calcul initial d'un nombre de Fibonacci par un acteur muni d'un méta-acteur de mémorisation se révèle être beaucoup plus rapide que le même calcul par l'acteur non réifié (de l'ordre de 35 fois plus rapide) ! Ce gain très significatif est dû à l'élimination des calculs redondants qui sont remplacés par des attentes. Le nombre de processus nécessaires au calcul d'un nombre de Fibonacci s'en trouve d'autant diminué.

9.1.1 Conclusion

Cet exemple illustre l'intérêt de la réflexivité. À l'aide d'un méta-acteur, l'algorithme de calcul utilisé par un acteur a été modifié sans que son expression ne le soit. Il montre aussi que la réflexivité n'est pas toujours synonyme de perte d'efficacité. Utilisée judicieusement, elle peut permettre des gains notables sur les temps d'exécution. Notons cependant que les gains obtenus sont liés à l'architecture matérielle. Avec une machine parallèle, le gain ne serait sans doute pas le même.

9.2 De LActE/R à ABCL/1

ABCL/1 ([Yonezawa 86b], [Yonezawa 86a]) est le premier membre de la famille ABCL (ABCL/R [Watanabe 88], ABCL/R2 [Masuhara 92], ABCL/R3 [Masuhara 95]). Conçu par l'équipe de A. Yonezawa, c'est un langage à objets concurrents dont le modèle de calcul s'inspire largement du modèle acteur tel qu'il a été défini par C. Hewit [Hewit 69]. Contrairement à ses successeurs, ce langage n'est pas réflexif.

ABCL/1 et LActE/R, outre les aspects réflexifs propres à ce dernier, se différencient par leur modèle de calcul et l'approche minimaliste qui a présidé à la conception de LActE [Regourd 94]. Les différences les plus notables sont:

- Les objets de ABCL/1 sont créés directement à partir d'un modèle contrairement à la relation d'instanciation qui lie les comportements aux acteurs de LActE/R.

- Les méthodes de ABCL/1 s'exécutent séquentiellement les unes après les autres: il n'y a pas de concurrence entre les méthodes d'un même objet.

- ABCL/1 préserve l'ordre d'arrivée des messages: si m_i est envoyé avant m_j alors m_i sera reçu avant m_j.

D'autres différences opposent ABCL/1 et LActE/R. Nous allons cependant nous focaliser sur celles qui viennent d'être citées en les insérant à une couche ABCL/1 introduite dans LActE/R à l'aide de ses capacités réflexives.

9.2.1 Arrivée ordonnée des messages

L'une des différences les plus importantes entre LActE/R et ABCL/1 réside dans la conservation de l'ordre d'arrivée des messages garanti par ABCL/1. Notre interprète ne préserve par cet ordre des messages. Introduire les objets de ABCL/1 dans LActE/R nécessite donc l'introduction de la préservation de l'ordre d'arrivée des messages.

Les messages de LActE/R peuvent arriver dans n'importe quel ordre. Cette caractéristique permet, lors de la distribution de LActE, de ne pas se préoccuper des chemins employés par différents messages. L'implémentation des objets d'ABCL/1 demande une modification des mécanismes de LActE/R afin de préserver l'ordre d'arrivée des messages.

Pour garantir l'ordre d'arrivée des messages nous allons utiliser un protocole simple basé sur la numérotation des messages. Chaque message transmis d'un objet à un autre reçoit une estampille contenant un numéro. Ces numéros sont spécifiques à chaque couple (*émetteur*, *récepteur*) et traduisent l'ordre d'émission des messages sachant que le premier message envoyé vers un destinataire donné est toujours numéroté 1.

Un message numéroté est dérivé des messages de LActE/R auxquels sont ajoutées une accointance qui contient le numéro du message et une méthode qui en autorise la lecture:

```
; Les messages numérotés
(def-behavior a-numbered-message(number) t
   ; number est le numéro du message
   (declare (is-a an-actor-message))
   ; Méthode de lecture du numéro du message
   ((:get-number) (result number)))
```

La manipulation de ces messages numérotés nécessite l'emploi d'un méta-acteur. Il doit permettre l'émission de messages numérotés ainsi que leur réception dans un ordre correct.

Les messages recevant un numéro qui dépend du récepteur du message, la gestion de l'ordre d'arrivée des messages dépend de l'émetteur du message. Les numéros du dernier message reçu d'un émetteur donné ainsi que du dernier message envoyé à un récepteur donné doivent être conservés. Les accointances *senders* et *receivers* contiennent la liste de chacun de ces numéros respectivement pour les messages émis et les messages reçus. Enfin une troisième accointance contiendra la liste des messages qui, reçus dans le désordre, seront en attente de réception par la dénotation. Le comportement de notre méta-acteur prend la forme suivante:

```
(def-behavior a-meta-abcl-actor
              (receivers senders unordered-messages) t
    (declare (is-a a-meta-actor-serialized))
    ...)
```

Notons que le méta-acteur est sérialisé et méta-acteur d'un acteur sérialisé. Tous les objets d'ABCL/1 seront implémentés par des acteurs sérialisés, qu'ils aient ou non des accointances. L'utilisation d'acteurs sérialisés est imposée par le modèle de calcul d'ABCL/1 basé sur une exécution des méthodes en exclusion mutuelle.

L'ensemble des messages envoyés par un objet reçoivent un numéro. Ce numéro doit correspondre à l'ordre dans lequel les messages sont émis. La numérotation des messages est à la charge d'une méthode dédiée, :send−message. Les instructions d'envois de message sont réifiées vers le méta-acteur qui leur donne une implémentation basée sur l'utilisation de :send−message. Afin que la numérotation des messages corresponde bien à l'ordre de leur émission, le blocage des instructions réifiées est prolongé jusqu'à ce que le numéro du message soit déterminé. En particulier, dans le cas des envois asynchrones de message, ce n'est pas la méthode qui implémente l'instruction qui retourne son résultat mais la méthode :send−message. L'ensemble des méthodes qui implémentent les instructions d'envois de message adoptent le code suivant:

```
 ; Envoi synchrone
((:ask mesg-message a-to a-sender a-take)
 (send [:send−message mesg-message a-to a-sender forward-to
                      a-take :ask] :to self))
 ; Retour asynchrone d'un résultat
((:result mesg-content mesg-to)
 (send [:send−message mesg-content mesg-to denotation
                      denotation 1 :send]
       :to self :forward−to forward-to))
 ; Envoi asynchrone
((:send content destination mesg-sender mesg-forward-to
        mesg-take mesg-type)
 (send [:send−message content destination mesg-sender
                      mesg-forward-to mesg-take mesg-type] :to self))
 ; Numérotation et envoi des messages
((:send−message content destination mesg-sender mesg-forward-to
                mesg-take mesg-type)
  ; Recherche du numéro du dernier message envoyé au
  ; destinataire
 (let* ((descr (assoc destination receivers))
          ; Calcul du numéro du message courant:
          ; Si c'est le premier message envoyé au destinataire
          ; alors le numéro du message est 1
          ; sinon c'est le numéro du dernier message envoyé plus 1
         (number (cond (descr (cadr (rplacd descr (list (1+ (cadr descr))))))
                       (t (setf receivers
                                (cons (list destination 1) receivers)) 1))))
    (update)
      ; Dans le cas d'un result ou d'un send, l'instruction
      ; peut être maintenant débloquée.
    (when (eq mesg-type :send) (result t))
     ; Envoi d'un message numéroté
    (send-message
            (new-anonymous a-numbered-message ()
                           destination mesg-sender mesg-forward-to
                           content mesg-take mesg-type number)))))
```

Le *multiple-send* ainsi que les envois avec promesse de résultat ne sont pas implémentés.

Le numéro du message à envoyer est déterminé en fonction de l'adresse du destinataire du message. L'accointance *receivers* contient une A-liste qui, à chaque adresse de destinataire, associe le numéro du dernier message envoyé. L'acteur commence par examiner cette accointance afin de déterminer le numéro du dernier message envoyé. Si n, ce numéro, existe, le numéro du message courant est $n + 1$. Dans le cas contraire, le message est le premier message envoyé à ce destinataire. Son numéro est, par convention, 1.

La réification des instructions d'envois de message vers le méta-acteur est obtenue à l'aide du réificateur suivant:

```
; Réificateur d'un objet d'ABCL/1
(defvar *abcl-reifier*
        (make-reifier :send   :meta
                      :ask    :meta
                      :result :meta)
```

La réception des messages est un peu plus complexe. C'est là que les messages sont remis dans le bon ordre avant d'être traités. Mais aussi, et afin d'assurer une certaine symbiose entre les acteurs et les objets d'ABCL/1, c'est là que sont reçus les messages non numérotés (en provenance d'acteurs). Ces derniers sont directement déposés dans la boîte aux lettres de la dénotation.

Les messages numérotés ne sont déposés dans la boîte aux lettres de la dénotation que lorsqu'ils sont reçus dans l'ordre. Le schéma de transmission des messages de LActE/R peut provoquer des désordres dans la réception des messages. Lorsqu'un message numéroté est reçu avant que son prédécesseur ne le soit, il est mis en attente dans une liste spécifique. Ce n'est que lorsque son prédécesseur aura été effectivement reçu que ce message sera déposé dans la boîte aux lettres.

La méthode de réception des messages pour les objets d'ABCL/1 est:

```
; Réception des messages
((:receive-message mesg)
 (cond
   ; Réception de messages numérotés
   ((eq (behavior-of mesg) a-numbered-message)
    ; Récupération de l'adresse de l'émetteur
    (let* ((mesg-sender (ask [:get-sender] :to mesg))
           ; Récupération du numéro du message
           (mesg-number (ask [:get-number] :to mesg))
           ; Récupération du dernier numéro reçu
           (descr (assoc mesg-sender senders))
            ; Le message est il reçu dans un ordre correct
            ; (faux si l'ordre est incorrect ou si c'est
            ; le premier message en provenance de l'émetteur)
           (right-order? (and descr (eq (1+ (cadr descr)) mesg-number))))
           ; Si le message est reçu dans l'ordre
      (cond ((or right-order? (eq 1 mesg-number))
                 ; Il est ajouté à la liste des messages reçus
                (setf actor-messages
```

```
                    (append actor-messages (list mesg)))
                  ; Le numéro du dernier message reçu de l'émetteur
                  ; est mis à jour ou créé
                (if right-order?
                  (rplacd descr (list mesg-number))
                  (setf senders (cons (list mesg-sender 1) senders)))
                  ; Si des messages reçus dans le désordre sont
                  ; en attente ils sont à nouveau traités
                (when unordered-messages
                  (send [:send−unordered (cdr unordered-messages)] :to self)
                  (setf unordered-messages ())))
                ; Le message est reçu dans le désordre:
                ; il est mis en attente dans unordered-messages
               (t (setf unordered-messages
                        (append unordered-messages (list mesg)))))))
  ; Réception de messages provenant d'acteurs
 (t (setf actor-messages (append actor-messages (list mesg)))))
  ; Activation des messages
 (when (and (not actor-active ) actor-messages)
   (send [:activate−message (first actor-messages)] :to self)
   (setf actor-active t
         actor-messages (rest actor-messages)))
 ; Mise à jour des accointances modifiées
(update))
```

Lorsqu'un message numéroté est reçu, le numéro du dernier message reçu en provenance de l'émetteur est recherché et comparé à celui du message reçu. Si ces numéros se suivent, le message est accepté et mis dans la boîte aux lettres de la dénotation. Ce comportement est reproduit dans le cas d'un message portant le numéro 1 et dont l'émetteur n'est pas encore répertorié dans *senders*. Une fois le message déposé, les messages en attente sont réexpédiés afin de les examiner à nouveau. Un schéma plus élaboré de gestion des messages en attente de réception pourrait être implémenté en utilisant une A-liste les associant avec l'adresse de leurs émetteurs. Mais le mécanisme utilisé ici est suffisant bien que pas optimal.

Les messages non ordonnés sont eux mis en attente dans *unordered-messages*.

Après la mise à jour de la boîte aux lettres, le méta-acteur détermine s'il y a lieu d'activer un message ou non. Finalement, l'ensemble des accointances modifiées est mis à jour.

La méthode :send−unordered assure le réexamen des messages arrivés dans le désordre. Elle parcourt la liste des messages qu'elle envoie un à un à la méthode :receive−message. Le corps de cette méthode est:

```
 ; Renvoi des messages reçus dans le désordre pour réexamen
((:send−unordered messages)
 (become)
 (when messages
   (send [:receive−message (first messages)] :to self)
   (send [:send−unordered (rest messages)] :to self)))
```

La méthode effectue une simple récursion sur la liste des messages arrivés dans le désordre.

Chaque objet d'ABCL/1 réifié vers un méta-acteur qui adopte ce comportement recevra les messages échangés avec les autres objets dans l'ordre où ils auront été émis. Plus généralement, tout acteur muni d'un tel méta-acteur se comportera, du point de vue de l'envoi et de la réception des messages, selon le modèle adopté par ABCL/1.

9.2.2 Construction des objets ABCL/1

À l'exception des routines internes aux objets, la manipulation des objets d'ABCL/1 suit le modèle défini dans [Shibayama 86]. Un objet d'ABCL/1 est décrit par la syntaxe suivante:

> (*abcl-object* <*nom*>?
>   (*state* (<*variable*> <*valeur*>)+)?
>   (*script* (<*filtre*> <*corps*>)+))

avec

> <*filtre*> ::= <*filtre du message*> *where* <*contrainte*>

Cette déclaration se traduit par la création d'un objet. Il faut donc adapter le modèle basé sur l'instanciation de comportements propre à LActE/R en couplant la création d'un comportement avec celle de son instance. Un objet est donc créé par instanciation d'un comportement défini à partir des informations extraites de sa déclaration. Les deux opérations, création du comportement et instanciation, se confondent dans une seule et même opération, la création de l'objet.

Ce couplage des deux opérations n'est possible que parce qu'une certaine équivalence entre le modèle LActE/R et le modèle ABCL/1 peut être établie. La déclaration précédente se traduit en LActE/R par les déclarations suivantes:

> (**def-behavior** <*nom du comportement de l'objet*> (<*variable*>+) *t*
>   (<*filtre*> <*corps*> (**update**))+)

et

> (**new** <*nom*> <*nom du comportement de l'objet*> () <*valeur*>+)

ou

> (**new-anonymous** <*nom du comportement de l'objet*> () <*valeur*>+)

Cette équivalence n'est cependant pas totale: les filtres de méthodes de LActE/R n'autorisent pas la présence d'une contrainte. L'ajout des contraintes implique la modification des filtres des méthodes. Une telle modification comprend deux aspects: statique et dynamique. Les aspects statiques concernent la compilation du filtre de la méthode associé avec sa contrainte. Les aspects dynamiques concernent la modification du mécanisme de recherche d'une méthode dans le but de transmettre au prédicat les éléments complétant l'environnement d'évaluation des contraintes. Les contraintes pourront ainsi porter sur les informations contenues dans le message mais aussi sur les valeurs des variables d'état (ou accointances).

Aspects dynamiques

La transmission des valeurs des variables d'état aux prédicats qui filtrent les méthodes implique la modification de l'ensemble des éléments inclus dans la chaîne qui réalise la recherche de la méthode qui correspond à un message donné. Par conséquent, les acteurs qui représentent les objets d'ABCL/1 devront avoir un comportement spécifique et des méthodes adaptées.

Les méthodes des objets se caractérisent par une méthode :matching? qui accepte un paramètre supplémentaire, la liste des valeurs des accointances. L'accès à cette liste étend les éléments sur lesquels la contrainte peut porter. Le comportement d'une telle méthode, dérivé de celui d'une méthode de LActE/R, est:

```
(def-behavior an-abcl-method() ()
  (declare (is-a an-actor-method))
  ((:matching? a-message acqs-values)
   (result (funcall compiled-pattern a-message acqs-values))))
```

Le paramètre *a-message* passé au prédicat de filtrage des messages est différent de celui habituellement utilisé. Au lieu de ne passer que le contenu du message, l'ensemble du message est passé. Les contraintes accèdent ainsi à la totalité des informations transmises par le message.

Les comportements des objets dérivent des comportements de LActE/R avec une adaptation de leur méthode de recherche de la méthode qui filtre le message passé en paramètre. Il s'agit toujours de parcourir la liste des méthodes du comportement pour trouver la première qui filtre le message mais en transmettant les valeurs demandées lorsque la méthode est une méthode d'ABCL/1. Les méta-méthodes, qui restent des méthodes de LActE/R, continuent de recevoir le contenu du message en paramètre. Le comportement des comportements d'objets d'ABCL/1 est:

```
(def-behavior an-abcl-object-behavior() ()
  (declare (is-a an-actor-behavior))
   ; Parcours de la liste des méthodes
  ((:getting-method a-message mesg-content acqs-values methods)
   (if (null methods)
     (result nil)
     (let* ((a-method (first methods))
            (methods (rest methods))
            (match?
              (if (eq (behavior-of a-method) an-abcl-method)
                ; Cas d'une méthode d'ABCL/1
                (ask [:matching? a-message acqs-values] :to a-method)
                ; Cas d'une méthode de LActE/R
                (ask [:matching? mesg-content] :to a-method))))
       (if match?
         (result a-method)
         (send [:getting-method a-message acqs-values methods]
               :to self :forward-to forward-to)))))
```

```
    ; Complète les paramètres nécessaires à la recherche de la méthode
  ((:get-actor-method a-message acqs-values)
   (send [:getting-method a-message
                          (ask [:get-message] :to a-message)
                          acqs-values behavior-actor-methods]
         :to self :forward-to forward-to)))
```

La chaîne des acteurs qui participent à la recherche de la méthode qui filtre un message reçu commence avec le méta-acteur (Fig. 9.4 page suivante). Le méta-acteur des acteurs qui implémentent les objets d'ABCL/1 a déjà été décrit dans la section précédente. En plus des mécanismes de réception des messages dans l'ordre d'émission, il faut modifier la méthode d'activation des messages qui déclenche la recherche d'une méthode. Cette modification se traduit par l'ajout d'un paramètre, la liste des valeurs des accointances. Le corps de cette méthode est:

```
((:activate-message a-message)
 (become)
   ; Transmet les valeurs des accointances au comportement
   ; pour que le test de contrainte y ait accès
 (let ((a-method (ask [:get-actor-method a-message actor-acquaintances]
                      :to actor-behavior)))
   (if a-method
     (send [:evaluate a-method a-message] :to self)
     (send [:no-matching-actor-method a-message] :to self))))
```

Cette modification du méta-acteur des objets complète la chaîne des acteurs qui participent à la recherche de la méthode qui filtre un message reçu.

Aspects statiques

La construction des objets d'ABCL/1 (Fig. 9.4 page ci-contre) est obtenue par:

- Une macro qui détermine la syntaxe de la déclaration des objets et s'occupe de la création de l'acteur qui représente l'objet ainsi que de sa réification.
- Un constructeur de comportement, *abcl-object-constructor*, qui, à partir des déclarations, calcule le comportement de l'acteur qui représente l'objet.
- Un constructeur de prédicat de filtrage des messages, *process-pattern*, qui construit la forme de la lambda qui détermine si un message est accepté, en fonction du filtre de la méthode et de sa contrainte.

Ces trois éléments gèrent l'ensemble des aspects statiques relatifs à la manipulation des objets de ABCL/1.

Les filtres des méthodes des objets d'ABCL/1 incluent, en plus d'un modèle de filtre de messages, une contrainte optionnelle. L'utilisation de méthodes, d'un comportement et d'un méta-acteur propres aux objets a permis le transfert des informations nécessaires au contexte

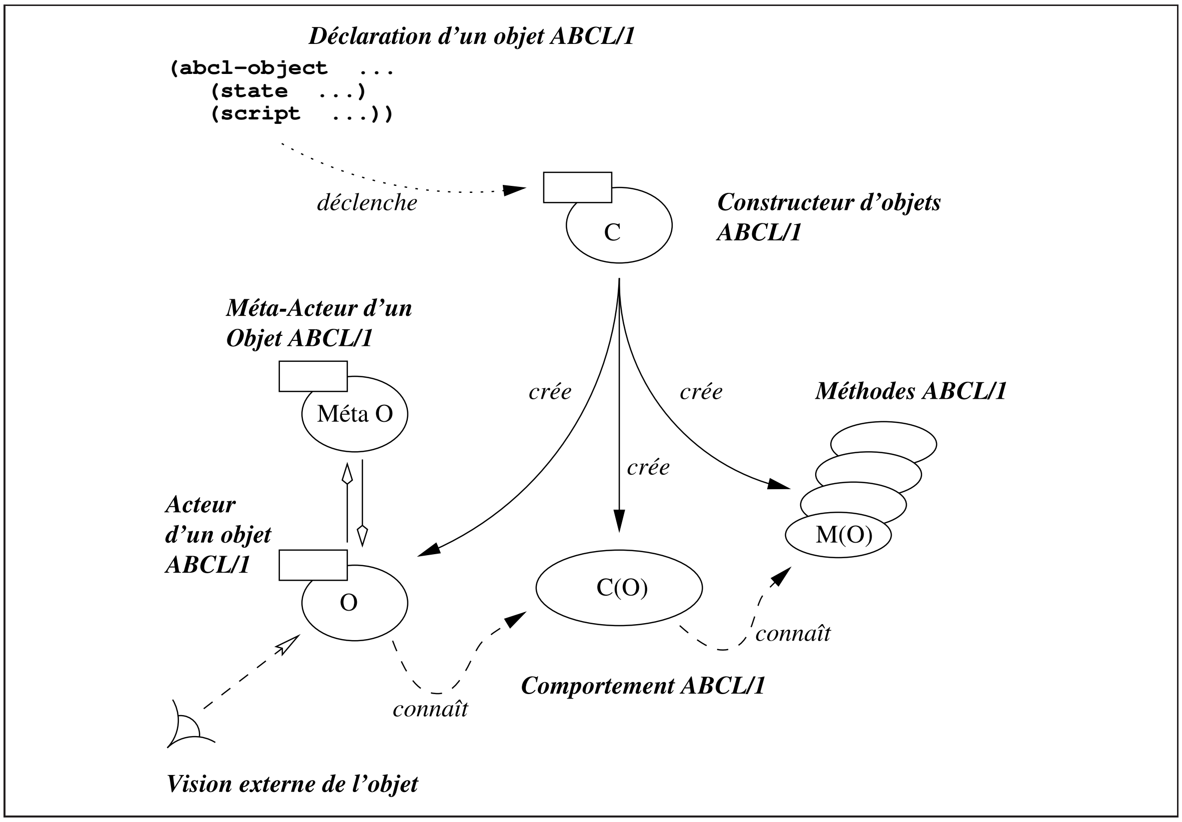

FIG. 9.4 – *Les objets d'ABCL/1 en LActE/R.*

d'évaluation des contraintes du méta-acteur vers le prédicat de filtrage d'une méthode. Ce prédicat accepte désormais deux paramètres et sa valeur de retour dépend du modèle du filtre et de la contrainte. L'adaptation du prédicat s'opère lors de la construction du comportement de l'objet.

La compilation du filtre de la méthode est obtenue grâce au comportement et à l'acteur suivants:

```
(def-behavior a-process-pattern() ()
  ; Remplace les constantes par des symboles
  ; Utilisée pour transformer un modèle de filtre
  ; en liste de paramètres. La liste retournée en résultat
  ; peut servir de vecteur de paramètres à une lambda.
 ((:replace-constants pattern)
  ; Effectue un parcours récursif du pattern
  (if (null pattern)
    (result ())
    (send [:replace-constants (rest pattern)] :to self :forward-to
         (continuation ((cont forward-to)
                        (elt (first pattern)))
           ((rest-pattern)
             ; Si l'élément du modèle est une constante,
             ; elle est remplacée par un symbole.
```

```
                   (send [(cons (if (constantp elt)
                                    (gensym "constant-")
                                    elt)
                                rest-pattern)] :to cont))))))
    ; Construit la lambda qui représente un
    ; prédicat de filtrage
   ((:process-pattern pattern constraint acqs-names)
     ; La compilation du filtre est déléguée à une fonction
     ; de LActE/R.
    (let* ((compiled-matching (compile-matching pattern))
           ; Le pattern est transformé en vecteur de paramètres
           (message-args (ask [:replace-constants pattern] :to self)))
      (result
        (if constraint
           ; Construction d'un prédicat avec contrainte
           `(lambda (message acqs-values)
               ; Construction de l'environnement d'évaluation de
               ; la contrainte: elle accède aux informations
               ; contenues dans le message
              (multiple-value-bind
                          (self sender forward-to mesg-content
                                mesg-take mesg-type)
                          (ask [:get-values] :to message)
                  ; Le filtrage du message est d'abord testé
                 (and (funcall ,compiled-matching mesg-content)
                        ; Construction de l'environnement d'évaluation
                        ; de la contrainte: elle accède aux valeurs
                        ; des accointances
                      (multiple-value-bind ,acqs-names
                                       (values-list acqs-values)
                          ; Construction de l'environnement d'évaluation
                          ; de la contrainte: elle accède aux éléments
                          ; du message.
                        (multiple-value-bind ,message-args
                                             (values-list mesg-content)
                           ; Évaluation de la contrainte
                           ,constraint)))))
           ; Construction d'un prédicat sans contrainte
           `(lambda (message acqs-values)
               ; Seul le premier paramètre est utilisé
              (declare (ignore acqs-values))
               ; message contient l'acteur message et non
               ; simplement son contenu.
              (funcall ,compiled-matching
                       (ask [:get-message] :to message))))))))
```

```
  ; Création de l'acteur qui génère les prédicats
  (new process-pattern a-process-pattern ())
```

Le comportement de l'acteur *process-pattern* est constitué de deux méthodes. La première de ces méthodes, :replace−constants, élimine toutes les constantes du filtre. Le résultat est une liste de symboles utilisée pour lier les valeurs contenues dans le message aux symboles initiaux contenus dans le filtre. Le prédicat utilise cette liste pour accéder aux valeurs transmises dans le contenu du message.

La seconde méthode construit la forme de la lambda utilisée comme prédicat. L'évaluation de cette forme permet d'obtenir le prédicat. Deux formes différentes sont construites selon que le filtre du message est associé ou non à une contrainte. En l'absence de contrainte, le rôle de la lambda est d'adapter les paramètres passés au prédicat à ceux attendus par le résultat de la compilation du filtre. Lorsque le filtre est associé à une contrainte, la lambda doit en plus construire l'environnement d'évaluation de la contrainte. Le résultat du prédicat est alors obtenu par un *et* logique entre l'appel de la lambda compilée qui résulte du filtre de la méthode et l'évaluation de la contrainte.

L'acteur *abcl-object-constructor* produit à partir de la définition d'un objet, son nom, la liste des expressions qui correspondent aux valeurs des variables de l'objet ainsi que le comportement de l'acteur qui le représente. L'ensemble de ces données est directement utilisé par la macro chargée de la création de l'objet.

Le comportement de *abcl-object-constructor*, *an-abcl-object-constructor*, comporte cinq méthodes:

- :compile−method crée une méthode compilée. Elle se distingue de la procédure normale par la prise en compte de la contrainte dans le prédicat de filtrage de la méthode.
- :compile−script construit la liste des méthodes compilées.
- :build−behavior crée le comportement de l'acteur qui représente l'objet à partir de la liste des méthodes compilées.
- :process−state construit la liste des noms des accointances et la liste des expressions qui permettent d'en calculer la valeur à partir des déclarations contenues dans le champ *state* de l'objet.
- :build−object−behavior est le point d'entrée de l'acteur *abcl-object-constructor*. Elle extrait les informations du texte de la déclaration avant de les passer aux autres méthodes qu'elle appelle. Elle retourne l'ensemble des résultats attendus par la macro.

Le comportement *an-abcl-object-constructor* et l'acteur *abcl-object-constructor* sont créés à partir des déclarations suivantes:

```
(def-behavior an-abcl-object-constructor() ()
   ; Création d'une méthode compilée
   ((:compile-method acquaintances method)
    (let* ((pattern (first method))
             ; Extrait la contrainte si elle existe
           (constraint (when (eq (second method) 'where)
                              (third method)))
             ; Construction du corps d'une méthode
```

```
          (body (append
                     ; Élimination de la contrainte du corps
                     ; de la méthode.
                 (if constraint
                     (cons (first method) (cdddr method))
                     method)
                    ; Ajout d'un update à la fin du corps
                 '((update))))
            ; Compilation du filtre de la méthode.
          (compiled-pattern
            (compile nil
                 (ask [:process-pattern pattern constraint acquaintances]
                      :to process-pattern)))
            ; Compilation du corps de la méthode
            ; (déléguée à une fonction de LActE/R)
          (compiled-body
            (compile-actor-method-body t acquaintances body)))
         ; Création de la méthode compilée
     (result (new-anonymous an-abcl-method ()
                                  (cons pattern constraint)
                                  (rest body) t compiled-pattern
                                  compiled-body))))
 ; Création de la liste des méthodes compilées à partir
 ; du texte décrivant les méthodes
((:compile-script acquaintances script)
 (if (null script)
   (result ()))
   (send [:compile-script acquaintances (rest script)]
         :to self :forward-to
         (continuation ((cont forward-to)
                        (myself self))
           ((methods)
            (let ((method (ask [:compile-method acquaintances
                                              (first script)] :to myself)))
              (send [(cons method methods)] :to cont)))))))
 ; Construction du comportement de l'acteur qui représente l'objet
((:build-behavior name acquaintances script)
 (let* ((declared-methods
                (ask [:compile-script acquaintances script] :to self)))
     (result (new-anonymous an-abcl-object-behavior ()
                                  name acquaintances t
                                  (append (make-meta-actor-methods t)
                                  declared-methods)))))
 ; Extraction de la liste des accointances et de la
 ; liste de leur valeurs à partir des déclarations du
 ; champ state de l'objet
((:process-state state)
 (if (null state)
   (result () ())
   (send [:process-state (rest state)] :to self :forward-to
         (continuation ((cont forward-to)
                        (a-decl (first state)))
           ((state-variables state-values)
```

```
                (send [(cons (first a-decl) state-variables)
                       (cons (second a-decl) state-values)]
                      :to cont))))))
 ; À partir des déclarations fourni le nom de l'objet
 ; (si il existe), la liste des valeurs des accointances,
 ; et le comportement de l'acteur qui représente l'objet.
((:build-object-behavior body)
 (let* ((name (if (symbolp (first body)) (first body) nil))
        (behavior-name (if name
                           (gensym (string name))
                           (gensym "anony-")))
        (body (if name (rest body) body))
        (state (rest (assoc 'state body)))
        (script (rest (assoc 'script body))))
    (multiple-value-bind (state-variables state-values)
                         (ask [:process-state state] :to self)
      (result name state-values
              (ask [:build-behavior behavior-name state-variables
                                    script] :to self))))))
```

```
(new abcl-object-constructor an-abcl-object-constructor ())
```

Enfin la macro *abcl-object* fournit une interface ad hoc au constructeur d'objets. L'utilisateur dispose ainsi d'un outil syntaxique qui lui donne le moyen de respecter la syntaxe définie en début de section. Cette macro est:

```
(defmacro abcl-object(&rest body)
   (multiple-value-bind (name state-values behavior)
                        (ask [:build-object-behavior body]
                             :to abcl-object-constructor)
       ; Création de l'acteur qui représente l'objet
       ; (les valeurs des variables sont évaluées dans
       ; l'environnement lexical courant)
     `(let ((actor ,(if name
                        `(new ,name ,behavior nil ,@state-values)
                        `(new-anonymous ,behavior nil ,@state-values))))
        ; Mise en place du réificateur
       (ask [:set-reifier *abcl-reifier*] :to actor)
        ; Réification de l'acteur vers le méta-acteur qui
        ; implémente les mécanismes des objets ABCL/1
       (ask [:reify-to a-meta-abcl-actor () () () ()] :to actor)
       actor)))
```

La macro **abcl-object** transforme la définition d'un objet ABCL/1 en terme d'acteurs de LActE/R. Elle force l'évaluation des valeurs initiales des variables d'état de l'objet dans l'environnement lexical de la définition. La construction du comportement et des méthodes de l'objet est assurée par l'acteur *abcl-object-constructor*. À l'aide des informations retournées par cet acteur, la macro crée l'acteur qui implémente l'objet ABCL/1 qu'elle réifie immédiatement vers un méta-acteur qui lui donne la capacité d'échanger des messages ordonnés. Par ailleurs, elle dote l'acteur d'un réificateur qui réifie les envois de messages vers le méta-acteur.

À l'aide de l'ensemble de ces déclarations, des objets d'ABCL/1 peuvent être créés et manipulés au sein de la couche LActE/R.

9.2.3 Introduction de l'instruction *select*

La dernière étape de l'intégration d'ABCL/1 au sein de LActE/R consiste en l'ajout de l'une des instructions spécifiques d'ABCL/1, le *select*. Cette instruction permet de se mettre en attente d'un message l'intérieur du corps d'une méthode. La syntaxe du *select* est (adaptée à LActE/R):

```
(abcl-select (<filtre> [where <contrainte>]? <corps>)+)
```

Lorsque l'interprète rencontre une telle instruction, il examine les messages contenus dans la boîte aux lettres jusqu'à ce qu'il en trouve un accepté par l'un des filtres du *select* (la sémantique précise de cette instruction est décrite dans [Shibayama 86]). Le corps associé au filtre sélectionné est alors évalué dans le contexte courant enrichi des informations contenues dans le message. Le message est retiré de la boîte aux lettres.

L'implémentation d'une telle instruction dans LActE/R n'est possible qu'en associant approches statiques et dynamiques. La syntaxe de l'instruction laisse apparaître des expressions dont l'évaluation doit avoir lieu dans le contexte lexical de la méthode qui fait appel au *select*. Seule l'approche statique autorise, dans LActE/R, l'évaluation d'expressions dans l'environnement lexical.

Tout commence par la définition de l'instruction *abcl-select*. Cette définition va déterminer la part de dynamique et de statique dans notre implémentation. La recherche du premier des messages accepté par l'un des filtres de l'instruction *select* sera dynamique et représentera ce qu'il sera possible de réifier de l'instruction. L'évaluation des expressions de l'instruction et la construction de la forme qui permettra cette évaluation seront statiques.

Une instruction *select* est transformé en:

```
(multiple-value-bind (mesg-nb message)
                     (call-implementation (list <pfiltre_1> ... <pfiltre_n>))
  (multiple-value-bind (to sender forward-to mesg-message mesg-take mesg-type)
                       (ask [:get-values] :to message)
    (case mesg-nb
      (0 (multiple-value-bind (<s1_1> ... <s1_i>)
                              (value-list mesg-message)
           <corps_du_filtre_1>))
      ...
      (n-1 (multiple-value-bind (<sn_1> ... <sn_k>)
                                (values-list mesg-message)
             <corps_du_filtre_n>)))))
```

où

– Les <*pfiltre_j*> sont les prédicats de filtrages obtenus à partir des filtres de rang i.

– Les <*si_j*> sont les symboles contenus dans le filtre de rang i obtenus soit directement, par extraction du filtre, soit indirectement, en remplacement d'une constante ou d'un mot-clé contenu dans le filtre.

Le **call-implementation** passe à l'acteur qui réifie l'instruction *abcl-select* la liste des prédicats composés à partir des filtres de l'instruction. Le travail de la méta-instruction consistera à rechercher le premier message accepté par l'un de ces prédicats et à retourner le rang du prédicat ainsi que le message correspondant. Ce message est alors décomposé dans l'environnement lexical courant afin de construire l'environnement dans lequel l'expression qui correspond au filtre doit être évaluée. Le rang du prédicat sert à déterminer, à l'aide d'un **case** de Common Lisp, laquelle des expressions doit être évaluée.

L'instruction *abcl-select* est définie de la manière suivante:

```
(def-instruction abcl-select
     ; L'ensemble du corps de l'instruction est récupéré dans body
   :syntax (&rest body)
      ; Construit la forme finale de l'instruction pour évaluation
      ; dans l'environnement lexical courant. La transformation du
      ; corps de l'instruction est déléguée à l'acteur process-select
   :compilation
          ; Transformation du corps de l'instruction
        (multiple-value-bind (patterns body)
                                 (ask [:process—select—body 0 body *acquaintance-names*]
                                      :to process-select)
            ; Mise en forme de l'instruction
          '(multiple-value-bind (mesg-nb message)
                                 (call-implementation (list ,@patterns))
             (multiple-value-bind (to sender forward-to mesg-message
                                      mesg-take mesg-type)
                                   (ask [:get-values] :to message)
                 (case mesg-nb
                       ,@body))))
   :lambda—list (filters)
    ; L'instruction n'est utilisée que sous forme réifiée
   :implementation #'(lambda (&rest args) "Not Implemented")
      ; L'ensemble des prédicats de filtrage est transmis à l'acteur
      ; chargé de rechercher le premier message accepté.
   :reified—form '(:select-one-of ,filters))
```

La totalité du corps du *select* est récupérée par le paramètre *body* de l'instruction. Le champ :compilation assure la construction de l'environnement lexical dans lequel le **case** qui sélectionne une des branches du *select* est évalué. Cet environnement est construit pour étendre l'environnement courant et non se substituer à ce dernier. Le corps du *select* est préalablement transformé par l'acteur *process-select*. Il adapte sa syntaxe au **case** de Common Lisp et transforme les filtres du *select* en des prédicats de Common Lisp. L'instruction ne compile pas les prédicats de filtrage. C'est la compilation de la méthode qui s'en charge.

Lors de la compilation d'une méthode qui contient une instruction *select*, la forme compilée de l'instruction est substituée à l'appel de l'instruction. Le champ :compilation définit donc les aspects statiques de l'instruction. Il spécifie aussi quels sont les aspects dynamiques en laissant à l'implémentation de l'instruction *select* le soin de traiter la sélection d'un message accepté par l'un des filtres de l'instruction. Le **call-implementation** traduit l'appel de l'implémentation de l'instruction depuis la forme compilée.

Les aspects dynamiques de l'instruction, la sélection d'un message, seront traités par la méta-instruction du *select*. La requête *select-one-of* avec comme paramètres les prédicats des filtres de l'instruction est envoyée à cette méta-instruction. La recherche sera effectuée à chaque exécution de l'instruction.

L'instruction *select* n'est définie que pour fonctionner en mode réifié. La lambda contenue dans le champ :implementation de l'instruction n'effectue aucun calcul. La nécessité pour l'instruction d'accéder à la boîte aux lettres de l'objet rend obligatoire sa réification vers le méta-acteur de l'objet d'ABCL/1.

L'acteur *process-select* transforme le corps d'un *select* en une forme acceptable pour Lisp. Il calcule la liste des prédicats de filtrage ainsi que le contenu du **case** généré par l'instruction. Sa définition est:

```
(def-behavior a-process-select() ()
  ((:process-one-select a-select rank acqs)
   (let* ((pattern (first a-select))
          (constraint (when (eq (second a-select) 'where)
                            (third a-select)))
          (body (if constraint (cdddr a-select) (rest a-select))))
     ; La construction du prédicat est déléguée à process-pattern
     (result (ask [:process-pattern pattern constraint acqs]
                  :to process-pattern)
             ; Construction d'un élément du corps du case
             `(,rank (multiple-value-bind ,(ask [:replace-constants pattern]
                                                :to process-pattern)
                                          (value-list mesg-message)
                       ,@body)))))
  ((:process-select-body rank select-body acqs)
   (if (null select-body)
     (result () ())
     (send [:process-select-body (1+ rank) (rest select-body) acqs]
           :to self :forward-to
           (continuation ((cont forward-to)
                          (myself self)
                          (method (first select-body)))
             ((patterns rest-body)
              (multiple-value-bind (pattern body)
                                   (ask [:process-one-select method
                                         rank acqs] :to myself)
                (send [(cons pattern patterns)
                       (cons body rest-body)] :to cont))))))))
```

```
(new process-select a-process-select ())
```

L'acteur *process-select* met en forme le code d'un *select* à partir de sa définition. Le calcul des prédicats de filtrage est délégué à l'acteur *process-pattern* de la section précédente. L'acteur retourne la liste des prédicats de filtrage ainsi que le corps du **case**. Dans ce dernier, il remplace les prédicats par leur rang dans le *select*. La sélection d'une des branches du *select* s'opère donc sur le rang du premier filtre qui accepte l'un des messages reçus par l'objet.

Le méta-acteur de l'acteur qui représente l'objet sert de méta-instruction à *abcl-select*. Il faut donc ajouter quelques méthodes à son comportement ainsi qu'une accointance supplémentaire, *select-filters*. Les méthodes qui doivent être ajoutées sont:

```
; Initie la recherche d'un message accepté par le select
((:select-one-of message-patterns)
 (become)
 (send [:searching-messages message-patterns 0]
       :to self :forward-to forward-to))
((:searching-messages message-patterns mesg-nb)
 ; Parcourt la liste des messages contenus dans la boîte aux lettres
 (let ((mesg (nth mesg-nb actor-messages)))
   (cond (mesg
          (become)
            ; La recherche du filtre correspondant à un message
            ; est déléguée à select-pattern
          (send [:select-pattern mesg message-patterns 0
                                 actor-acquaintances forward-to]
                :to select-pattern :forward-to
                (continuation ((myself self)
                               (cont forward-to)
                               (mesg mesg))
                     ((:cont)
                      (send [:searching-messages message-patterns (1+ mesg-nb)]
                            :to myself :forward-to cont))
                     ((:remove-message)
                      (send [:remove-message mesg] :to myself)))))
          ; Lorsque tous les messages ont été examinés, les
          ; informations nécessaires à l'examen du prochain message
          ; sont stockées dans select-filters. La méthode
          ; :receive-message déclenchera la recherche dès réception
          ; d'un message.
         (t (setf select-filters (list message-patterns mesg-nb forward-to))
          (update)))))
```

La première méthode, :select–one–of, réalise l'interface avec l'instruction *select*. Elle reçoit en paramètre la liste des prédicats de filtrage associés à l'instruction. Elle initie le processus de recherche d'un message à l'aide de la seconde méthode.

La méthode suivante, :searching–messages, effectue la recherche du premier message contenu dans la boîte aux lettres qui est accepté par l'un des prédicats contenus dans la liste passée en paramètre. Étant donné que les méthodes des objets d'ABCL/1 s'exécutent en exclusion mutuelle, une seule instruction *select* peut être exécutée à la fois. Par conséquent une seule recherche peut avoir lieu à la fois. Cependant des messages peuvent continuer à arriver au cours de la recherche. C'est pourquoi le parcours de la boîte aux lettres se fait non sur son contenu mais sur la base du rang du message examiné. Cette base reste stable car les messages reçus sont mis en fin de liste et qu'aucun message ne sera extrait de la liste avant la fin de la méthode qui exécute le *select*.

Si aucun des messages contenus dans la boîte aux lettres n'est accepté par l'un des filtres, la recherche sera poursuivie par le mécanisme de réception des messages. Chaque message reçu sera regardé comme un candidat potentiel à l'un des filtres de l'instruction.

L'acteur *select-pattern* effectue la recherche du filtre qui accepte le message qui lui est proposé.

```
(def-behavior a-select-pattern() ()
      ; Parcours de la liste des prédicats de filtrage afin
      ; de trouver celui qui filtre le message
  ((:select-pattern mesg message-patterns pattern-nb
                    actor-acquaintances answer)
   (let ((a-pattern (nth pattern-nb message-patterns)))
     (if a-pattern
         ; Le prédicat accepte-t-il le message
         (cond ((funcall a-pattern mesg actor-acquaintances)
                ; Si oui, il est retourné au select et
                ; retiré de la boîte aux lettres
                (send [pattern-nb mesg] :to answer)
                (result :remove-message))
                ; Si non, passage au prédicat suivant
               (t (send [:select-pattern mesg message-patterns
                               (1+ pattern-nb) actor-acquaintances answer]
                        :to self :forward-to forward-to))))
         ; Si plus aucun prédicat n'est disponible,
         ; passage au message suivant
         (result :cont)))))
```

```
(new select-pattern a-select-pattern ())
```

La méthode :select−pattern parcourt la liste des filtres pour en trouver un qui accepte le message passé en paramètre. S'il ne trouve pas de filtre acceptant le message il envoie à la continuation le message :cont qui continuera la recherche. Sinon il envoie le rang du filtre du message et demande au méta-acteur de retirer le message de la boîte aux lettres.

La méthode de réception des messages doit être adaptée pour prendre en compte la présence d'un *select* en attente d'un message.

```
((:receive-message mesg)
 ...
 (cond ((and (not actor-active ) actor-messages)
        (send [:activate-message (first actor-messages)] :to self)
        (setf actor-active t
              actor-messages (rest actor-messages)))
```

```
              ; Cas d'un select en attente d'un message
              ; (l'accointance actor-active est nécessairement vrai)
            (select-filters
               ; Provoque la recherche du filtre correspondant sur
               ; le dernier message reçu.
              (send [:searching−messages (first select-filters)
                                         (second select-filters)]
                    :to self :forward−to (third select-filters))
              (setf select-filters '())))
  (update))
```

La méthode :receive−message du méta-acteur est adaptée pour poursuivre la recherche d'un message accepté par le *select* lorsque cette recherche sur la boîte aux lettres a échoué. La recherche est déclenchée lorsque le contenu de l'accointance *select-filters* n'est pas vide.

Enfin, le message sélectionné doit pouvoir être retiré de la boîte aux lettres. La méthode *remove-message* qui assure cette fonction est:

```
((:remove−message mesg)
 (setf actor-messages (remove mesg actor-messages))
 (update))
```

9.2.4 Exemple d'utilisation

En suivant la syntaxe décrite plus haut, il est désormais possible de décrire des objets à la manière de ABCL/1. Prenons l'exemple d'une cellule qui est un objet qui implémente une forme de partage de variable. La version implémentée ici utilise un lock pour garantir l'exclusivité de l'accès à la cellule. Seul l'objet qui a préalablement posé son lock est habilité à accéder en lecture et, ou en écriture à la valeur contenue dans la cellule. Les contraintes sur les filtres de messages préservent l'accès à la cellule jusqu'à ce que le lock soit levé par l'objet qui l'a posé. L'adresse de l'objet qui pose le lock est conservée dans une variable de l'objet si bien que les contraintes portent à la fois sur le contenu du message et sur l'une des variables. Le code de la cellule est:

```
(abcl-object cell
  (state (value 0)
         (locker nil))
  (script
    ((:lock) where (not locker)
     (setf locker sender))
    ((:unlock) where (eq sender locker)
     (setf locker nil))
    ((:get) where (eq sender locker)
     (result value))
    ((:set new-value) where (eq sender locker)
     (setf value new-value))))
```

Le *select* de ABCL/1 n'aurait été ici d'aucun secours: il bloque l'accès aux autres méthodes jusqu'à ce que la méthode dans laquelle il apparaît soit terminée. Or le lock utilisé dans l'objet cellule ne doit pas interdire l'accès aux méthodes de lecture et d'écriture pour l'objet qui l'a posé.

Dans cet exemple, extrait de ([Shibayama 86] page 17), le *select* contrôle l'accès à une fenêtre d'affichage en interdisant tout affichage lorsque le lock est posé. L'adaptation du code à notre syntaxe donne:

```
(abcl-object window-controller
    (state (window (ask [:new] :to create-window)))
    (script
      ((:write character)
       (send [:write character] :to window))
      ((:lock-with key)
       (result :locked)
       (abcl-select
              ((:release-lock-with k) where (eq k key)
               (result :unlocked))))))
```

Le *select* utilise une contrainte sur une clé pour garantir la source de la demande de levé du lock. Tant que le lock est posé, aucun accès à la méthode d'écriture ne peut avoir lieu.

9.2.5 Conclusion

L'exemple décrit ici utilise la presque totalité des capacités réflexives de LActE/R. Il montre comment utiliser un méta-acteur, des acteurs jouant le rôle de messages, comportements et méthodes mais aussi comment ajouter une instruction. L'ensemble de ces outils a permis d'implémenter certaines des caractéristiques des objets d'ABCL/1 en LActE/R. Cet exemple montre donc les aptitudes de la réflexivité, en général, et de LActE/R, en particulier, à intégrer de nouveaux mécanismes linguistiques dans un interprète.

9.3 Localité, quand tu nous tiens

L'exemple présenté maintenant se sert de la réflexivité pour en explorer un nouveau modèle: un modèle de décomposition des méta-acteurs à des fins de flexibilité et de performance est proposé et implémenté à l'aide des capacités réflexives de LActE/R. Cette démarche qui contribue à la démonstration des capacités de LActE/R, vise à montrer l'intérêt de l'utilisation de la réflexivité dans l'exploration de modèles réflexifs, ainsi qu'à expérimenter une autre architecture réflexive et à la valider.

Après une présentation du modèle en cause, nous montrerons comment LActE/R a permis son implémentation.

9.3.1 Une réflexivité toujours plus locale

Une des idées défendues par LActE/R est que la localisation de la réflexivité est déjà en soi un facteur de performance. En donnant à l'utilisateur la possibilité de ne réifier que les éléments nécessaires, l'architecture réflexive de LActE/R minimise le coût de la réflexivité que ne supportent plus que les entités réifiées. Un système de trace des messages reçus par un acteur pèsera essentiellement sur l'exécution de sa machine acteur. Les autres acteurs interagissant avec ce dernier n'observeront qu'une diminution minime, et indirecte, de leurs performances. Cette architecture, combinée avec une implémentation appropriée, permet d'obtenir un interprète réflexif utilisable.

La localisation de la réflexivité dans LActE/R est le résultat d'une mise en bijection d'un acteur avec un autre acteur, qui l'implémente, respectivement appelés dénotation et méta. Après réification du premier vers le second, le méta-acteur devient la machine acteur de sa dénotation. L'ensemble des mécanismes acteurs de la dénotation sont alors représentés, implémentés et exécutés directement par le méta-acteur.

Avec un tel modèle, un système de trace nécessite la réification de l'ensemble des mécanismes acteurs, qu'ils soient ou non utiles au dessein du programmeur alors que la trace des messages reçus par un acteur met uniquement en jeu la méthode :receive−message du méta-acteur. La réécriture, directe ou indirecte (par le biais de l'inclusion de comportement), des autres méthodes du méta-acteur s'oppose au principe de localité. Cette opposition est d'autant plus forte qu'un tel système de trace ne nécessite pas le remplacement du comportement par défaut de la méthode :receive−message mais plutôt son extension comme le suggère le code suivant:

```
; Traceur des messages reçus
(def-behavior a-received-messages-tracer() t
  ; Inclusion du comportement par défaut d'un méta-acteur
  ; sérialisé ⟶ réécriture implicite de l'ensemble des
  ; mécanismes de l'acteur dénotation
 (declare (is-a a-meta-actor-serialized))
 ; Modification de la réception des messages
 ((:receive−message mesg)
  (become)
   ; Appel de la méthode par défaut de réception des messages
  (send [:receive−message mesg] :to self :take (next-method))
   ; Ajout des instructions de trace
  (send [:get−message] :to mesg :forward−to
         (continuation ()
              ((mesg-message)
               (send [(format nil "~&Receive ~a" mesg-message)] :to screen))))))
```

Si le méta-acteur qui trace les messages reçus par sa dénotation s'écrit en 9 lignes, il équivaut cependant à la réécriture de l'ensemble des méthodes du méta-acteur. Il implique surtout la réification de l'intégralité des mécanismes acteurs.

Le principe de localité poussé à l'extrême voudrait que la mise en place du mécanisme de trace qui précède ne réifie que les aspects de la machine acteur dont il a vraiment besoin. Idéalement, la réification devrait ici se restreindre à la seule activité de réception des messages, sans qu'aucunes des autres activités, ni qu'aucunes des accointances du méta-acteur, ne soient, elles, réifiées. En allant encore plus loin, le mécanisme de trace, qui ne fait qu'ajouter des instructions qui ne se substituent pas au mécanisme par défaut, devrait pouvoir être mis en place en s'ajoutant à ce mécanisme et non en le remplaçant. Cette localisation de la réflexivité serait alors le reflet exact des mécanismes à réifier pour introduire un système de trace des messages reçus.

Le nouveau type de réflexivité que nous voudrions introduire dans LActE/R correspond à cette traduction plus exacte du principe de localité. À l'augmentation de la localité de la réification devrait correspondre une diminution corrélative du coût de la réflexivité due à la conservation de l'implémentation par défaut des mécanismes non réifiés.

Décomposition de la machine acteur

La localisation de réflexivité impose une identification précise des éléments de base qui composent la machine acteur. C'est en réifiant individuellement ces éléments qu'une localisation poussée de la réflexivité peut être obtenue.

LActE/R représente la machine acteur d'un acteur par un seul et unique acteur, le méta-acteur. À travers ses accointances et ses méthodes, le méta-acteur traduit les différents aspects de la machine. Les accointances représentent l'état de la machine acteur i.e. des variables comme la liste des messages en attente de traitement ou l'adresse du comportement actuel de la dénotation. Les méthodes correspondent aux activités de la machine acteur comme la réception d'un message ou son activation.

L'activité de la machine acteur se traduit par le déclenchement et l'enchaînement des méthodes qui composent le méta-acteur. Le déclenchement initial de la machine acteur est dû à la demande de réception d'un message provenant de l'extérieur i.e. la réception d'un message :receive−message. Les méthodes s'enchaînent ensuite les unes après les autres réalisant ainsi le travail de la machine acteur.

Lorsque le méta-acteur reçoit un message :receive−message, il active la méthode correspondante dans un environnement où l'ensemble de ses accointances sont disponibles. Or, dans le cas où la dénotation est un acteur sérialisé, les seules accointances qu'il manipule dans la méthode :receive−message sont la liste des messages et le booléen qui détermine si un message est déjà en cours de traitement ou non. Et dans le cas d'un acteur non sérialisé, aucune accointance n'est utilisée par la méthode de réception des messages. La représentation de la machine acteur peut être, elle aussi, affinée.

De façon simplifiée, et bien que cela ne soit pas la seule décomposition possible, les accointances et les méthodes d'un méta-acteur classique au sens de LActE/R sont assimilables aux éléments de base de la machine acteur. Chaque accointance et chaque méthode est alors un élément insécable de la machine acteur.

Une réflexivité très localisée doit alors permettre de manipuler ces éléments un à un i.e. de ne réifier que la liste des messages reçus ou l'adresse du comportement de la dénotation. Mais aussi, et parallèlement, une telle réflexivité doit permettre la réification d'une ou plusieurs des activités de la machine acteur, par exemple la réception des messages , aussi bien en

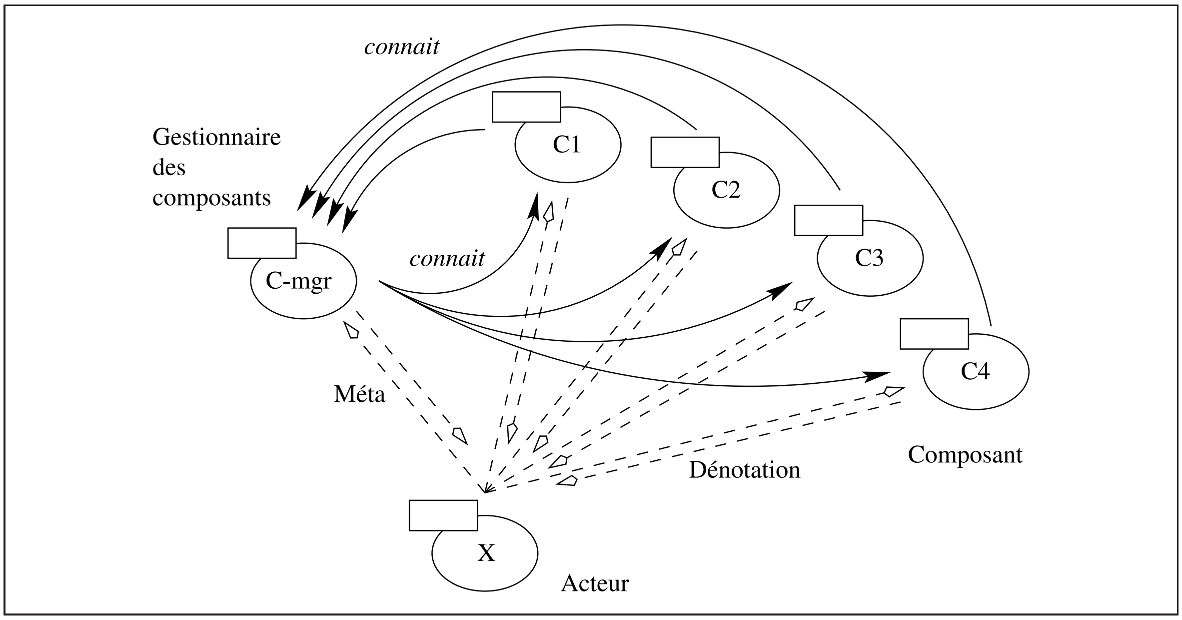

FIG. 9.5 – *Le modèle de réflexivité par composants*

remplacement de cette activité qu'en ajout à l'activité par défaut, comme cela est souhaitable dans le cadre d'un système de trace.

Réification des aspects élémentaires de la machine acteur

Les éléments à réifier de la machine acteur identifiés, l'interprète LActE/R doit être pourvu des outils linguistiques qui y donnent accès. Afin de conserver une uniformité linguistique qui en facilite la prise en main, ces outils doivent être exprimés à l'aide des seules constructions linguistiques disponibles dans LActE/R.

Les acteurs sont les seuls objets du langage qui offrent, en restant à l'intérieur de la couche LActE/R, un pouvoir expressif suffisant. Il ne s'agit pas pour autant de reprendre le schéma mis en place avec les méta-acteurs classiques de LActE/R. La différence entre ces derniers et le schéma de réification que nous voulons introduire ici réside dans la nature du lien entre méta et dénotation. Ce n'est plus l'ensemble de la machine acteur qui va être réifié mais certains de ses éléments, et cette réification ne va pas s'opérer vers un seul acteur mais plusieurs. Chacun de ces acteurs ne réifie pas l'ensemble de la machine acteur, mais les seuls éléments dont il a besoin pour opérer (Fig. 9.5).

La décomposition d'une dénotation vers plusieurs méta-entités présente de nombreux avantages:

- Le lien entre une méta-entité et sa dénotation se réduit à l'essentiel des éléments de la machine acteur. Une accointance qui reflète une partie de l'état de la machine acteur n'est alors réifiée que parce qu'elle est vraiment nécessaire au fonctionnement de la méta-entité. La même remarque vaut pour les activités de cette machine.

- La multiplicité des méta-entités permet de spécialiser chacune d'entre elles dans une tâche précise. La composition d'un mécanisme particulier de réception des messages avec une activation donnée des messages devient naturelle et ne nécessite plus, comme dans le schéma actuel, la réécriture d'un troisième comportement qui combine les deux premiers.

- La décomposition du méta autorise la conservation, pour les aspects non réifiés de la machine acteur, de comportements par défaut. En particulier, au niveau de l'implémentation, ces comportements par défaut peuvent s'exprimer directement en Lisp. Une telle combinaison de fonctions et d'acteurs favorise l'introduction de mécanismes d'optimisation aptes à améliorer notablement les performances de LActE/R.

L'essentiel de ces avantages proviennent de la combinaison de deux facteurs: la décomposition du méta et la réification élément par élément de la machine acteur.

Cette approche n'a cependant pas que des avantages: la réification d'un même élément de la machine acteur vers différentes méta-entités suppose un partage de cet élément entre elles. Un tel partage entre en contradiction avec le principe d'encapsulation qui sous-tend à la notion d'acteur. La réification d'une même variable d'état de la machine acteur par deux méta-entités différentes s'apparente à un partage d'accointances entre deux acteurs: sous des noms qui ne sont pas forcément identiques la même variable est lue et, ou écrite par deux acteurs différents. Les variables d'état de la machine acteur ne sont plus, comme dans le modèle classique, capturées au sein d'un seul et même acteur.

Le problème du partage des activités soulève aussi des questions critiques bien qu'elles ne remettent pas en cause les principes fondamentaux du modèle acteur. Deux types de réification d'une même activité sont distinguables. Le premier type correspond à une réification classique dans laquelle le composant devient le véritable détenteur de cette activité qu'il implémente. Il se substitue alors pleinement à la mécanique par défaut de la dénotation. Un seul composant de ce type peut exister pour une activité donnée (sous peine d'introduire des incohérences). L'autre type correspond à un composant qui vient s'ajouter à l'implémentation de l'activité. Il n'implémente pas cette activité mais accomplit des actions qui doivent être déclenchées à chaque fois que l'activité correspondante est mise en œuvre. Plusieurs composants qui utilisent ce type de réification peuvent coexister pour une même activité.

Le modèle de réflexivité discuté s'inscrit donc dans une rupture avec le modèle acteur. La source d'une telle rupture est à trouver soit dans le modèle acteur soit dans le modèle de réflexivité proposé. Sans entrer dans la polémique, il convient de se demander si la trop grande simplicité du langage n'en est pas la raison et dans quelle mesure cette rupture ne serait pas une extension du modèle de calcul qu'il serait souhaitable d'introduire. Les avantages de l'approche de la réflexivité introduite semblent justifier une telle rupture.

Puisque ces méta-entités n'ont de sens que combinées entre elles, appelons les composants. La vision ainsi donnée à ce type de réflexivité se rapproche d'un système mécanique, chaque composant étant une pièce, au sens fonctionnel du terme, de la machine acteur. Et ce n'est qu'une fois l'ensemble des composants de la machine acteur assemblés que cette dernière peut être mise en route.

En les nommant ainsi, nous les rapprochons du concept de composants défini par J.C. Pazzaglia [Pazzaglia 97]. Nos composants ont en effet des propriétés qui les rapprochent de ceux de J.C. Pazzaglia : par la possibilité qu'ils ont de se combiner entre eux, ils supportent un mécanisme de composition minimal. Cependant nos méta-entités ont un objectif différent — permettre une localisation optimale de la réflexivité en décomposant finement la machine acteur — et des moyens spécifiques. Chaque méta-entité est un acteur indépendant qui réifie certains des éléments de la machine acteur et non une méthode particulière qui s'exécute comme une des composantes du méta-acteur. Ces différences d'objectif et de moyens se traduisent par un mécanisme moins adapté au problème de la composition que la solution proposée par J.C. Pazzaglia.

Support linguistique

Les éléments de la machine acteur réifiés par un composant peuvent être spécifiés soit lors de la définition de son comportement soit au moment de sa création. La très forte dépendance de la nature du lien causal avec le code conduit à choisir une méthode de spécification statique i.e. lors de la spécification d'un comportement. La spécification des éléments réifiés lors de la création de l'acteur, tout en apportant de la souplesse, introduirait une certaine complexité liée à la nécessité pour le programmeur de connaître, au moment de cette création, le comportement dans tous ses détails.

Cette spécification doit identifier les accointances du composant qui correspondent à tel ou tel état de la machine acteur permettant ainsi d'établir un lien causal entre l'état de la machine acteur et la valeur d'une ou des accointances du composant. Une telle déclaration s'insère naturellement dans le **declare** des comportements. En admettant que notre composant veuille réifier la liste des messages reçus et le booléen d'activité de la machine acteur vers ses deux premières accointances, il peut le faire avec la déclaration suivante:

```
(declare
    ...
    (reify
        ...
        '(acquaintances :actor-messages :actor-active)
        ...)
    ...)
```

Le champ *reify* sert alors d'introduction à la description du lien causal entre le composant et sa dénotation. La sous-liste *acquaintances* décrit lesquels des états de la machine acteur sont réifiés vers les deux premières accointances du composant en reprenant, sous forme de mots-clés, les noms des accointances des méta-acteurs classiques de LActE/R.

Une telle description est liée au code par deux aspects: d'une part, la réification des états de la machine acteur intervient dans l'ordre spécifié par le programmeur, ces états étant réifiés vers les premières accointances du composant et, d'autre part, la nature de l'accès à ces variables est modifiée en jouant sur le booléen de sérialisation. Un composant non sérialisé n'a, en effet, pas besoin de détenir les variables d'état de la machine acteur mais seulement de les lire. La non garantie de l'ordre de délivrance des messages permet à un tel acteur de les lire à tout moment.

La spécification des activités réifiées suit une syntaxe similaire qui doit cependant être enrichie afin de différencier les activités réifiées en remplacement de l'activité par défaut de celle qui le sont en ajout à cette activité. La réification de l'activité de réception des messages prend la forme suivante:

```
(declare
        ...
        (reify
            ...
            '(activity :receive-message ...)
            ...)
        ...)
```

Le symbole *activity* introduit la sous-liste qui décrit les activités réifiées. Toujours en se basant sur la description des méta-acteurs classiques de LActE/R, ces activités sont dénommées par le premier mot-clé du message correspondant. Ce choix nécessite l'utilisation d'un mot-clé spécifique pour le :become avec arguments des acteurs sérialisés. Le mot-clé :become−with−args est utilisé à cette fin.

À ce système de désignation par mots-clés des activités réifiées, un système de qualification de la nature de la réification doit être ajouté. Un composant réifiant une activité peut opérer selon deux modes: soit l'activité réifiée vient en remplacement de l'activité par défaut et le composant tient lieu d'interprète pour cette activité, soit le composant ajoute son activité à l'activité par défaut. En qualifiant de :main la substitution à l'activité par défaut, parce qu'il ne peut y avoir qu'une seule activité de cette sorte, et de :other l'addition d'une activité supplémentaire, les deux types de réification des activités de la machine acteur peuvent être spécifiés. Par exemple, dans le cas d'une trace, la réification prendra la forme suivante:

```
(declare
        ...
        (reify
            ...
            '(activity :receive-message :other ...)
            ...)
        ...)
```

Chaque mot-clé spécifiant une activité est donc suivi d'un qualifiant qui précise la nature de la réification. Dans le cas d'un système de trace, le composant qui effectue la trace a juste besoin d'être notifié de chaque réception de message. La réception du message sera alors accomplie par le composant par défaut qui continue de servir d'interprète à cette activité.

Enfin les composants doivent pouvoir dialoguer entre eux. Plus précisément, le fonctionnement de la machine acteur résulte de l'enchaînement de différentes activités. Par exemple, la réception des messages enclenche la recherche de la méthode correspondante. Si un composant remplit le rôle de receveur des messages, et un autre celui de chercheur de méthode, leur coopération nécessite un moyen de communication.

Ce séquencement des activités peut être assuré de deux manières:

- Chaque composant qui représente l'activité principale connaît l'ensemble des composants, principaux ou autres, qui représentent l'activité suivante. Cette architecture est assez complexe à réaliser et la construction de la machine acteur délicate. De plus, le problème du partage des accointances que suppose la réification séparée de chaque variable d'état de la machine acteur ne peut être traité par ce biais.

- Les composants connaissent un acteur chargé d'assurer le bon routage des appels aux activités suivantes. À la réception d'une demande de recherche de méthode, il déroute cette demande vers les composants concernés. Un tel acteur, nommé gestionnaire des composants (*composant-manager*), peut aussi assurer le partage des accointances entre les composants qui en font la demande.

C'est cette deuxième solution que nous avons adoptée. Elle offre, en permettant une gestion plus dynamique des relations entre les composants, une flexibilité plus souple que la première. Sa mise en œuvre est aussi plus aisée.

Avec toutes ces données, un composant réalisant une activité de réception de messages pour les acteurs sérialisés s'écrit:

```
(def-behavior a-message-receiver(actor-messages actor-active) t
   (declare (reify '(acquaintances :actor-messages :actor-active)
                   '(activity :receive-message :main)))
   ((:receive-message mesg)
    (cond (actor-active
             (become a-message-receiver ()
                     (append actor-messages (list mesg))
                     actor-active))
          (t
             (become a-message-receiver ()
                     actor-messages t)
             (send [:activate-message mesg] :to component-manager)))))
```

Dans le **declare** du **def-behavior**, la sous-liste *reify* décrit le lien de causalité qui relie le composant à sa dénotation. L'activité de réception des messages manipule deux des variables d'état de la machine acteur: la liste des messages en attente de traitement et le booléen d'activité. Ce sont ces deux variables qui sont ici réifiées vers les deux premières accointances de l'acteur (respectivement, *actor-messages* et *actor-active*).

La seule activité réifiée est celle qui correspond à la réception d'un message. Un composant muni du comportement précédent ne recevra pas d'autre notification. Ce comportement reprend, en plus simple, la méthode de réception des messages des méta-acteurs classiques de LActE/R. La différence la plus notable dans le corps de la méthode réside dans le séquencement des activités: alors que les méta-acteurs classiques s'envoient à eux mêmes le message qui enclenche l'activité suivante, le composant envoie ce message au gestionnaire des composants (*component-manager*).

La notion de composant montre déjà ses avantages sur l'approche classique de LActE/R. Un des apports les plus remarquables est la lisibilité du code. Contrairement à l'approche classique, les éléments réifiés apparaissent clairement dans le code, et leur nombre réduit permet une prise en main plus aisée par le programmeur. Cette simplicité permet un retour à l'utilisation du **become** au lieu du **update**, dont l'existence est en partie due à la multiplicité des accointances à manipuler dans le code d'un méta-acteur classique. L'inclusion de comportement ne se justifie plus vraiment, chaque composant pouvant se réduire à une seule méthode.

Toujours en suivant les mêmes conventions, un composant qui effectue la trace des messages reçus s'écrit:

```
(def-behavior a-message-receiver() ()
    (declare (reify '(activity :receive-message :other)))
    ((:receive−message mesg)
     (send [:get−message] :to mesg :forward−to
           (continuation ()
             ((mesg-message)
              (send [(format nil "~&Receive ~a" mesg-message)] :to screen))))))
```

Ici, le lien causal se réduit à la notification de réception d'un message. Aucune accointance n'est réifiée par ce composant. L'usage d'une réification de type :other de l'activité de réception des messages réduit le code aux seules lignes requises pour effectuer la trace des messages reçus sans nécessiter de réécriture directe ou indirecte de la machine acteur tel que dans l'exemple de trace pour les méta-acteurs classiques. Le lien entre ce composant et la machine acteur ne relève plus que de la description du lien causal et non de subtilités de programmation. Une telle approche contribue significativement à faciliter l'utilisation de la réflexivité.

Récursivité du modèle à base de composants

L'une des caractéristiques des méta-acteurs est le caractère récursif du modèle de réflexivité dont ils sont à la base: un méta-acteur est, en tant qu'acteur, réifiable vers un méta-acteur. Les composants, par la décomposition du méta qu'ils induisent, peuvent-ils reproduire un même schéma récursif?

Dans notre modèle, un composant est présenté comme étant un acteur. La spécification du comportement d'un composant s'apparente pour beaucoup à celle d'un acteur classique. Il ne faut cependant pas occulter la spécificité des composants: la manipulation de certaines de leurs accointances, celles dont les valeurs dépendent du lien causal, passe par un dialogue avec le gestionnaire des composants. Ce dialogue, transparent au niveau du composant, doit, au niveau méta, être explicite.

L'explicitation des mécanismes de partage des accointances se traduit par deux modifications: une machine pour les composants qui représente correctement les dépendances entre composants et gestionnaire de composants, et l'introduction d'un axe réflexif supplémentaire, une sorte de réification exécutoire pour reprendre le terme utilisé par P. Carle [Carle 92], que représente la réification du gestionnaire de composants.

La machine des composants nécessite une mécanique légèrement différente de celle d'une machine acteur. Les différences essentielles résident dans la construction du contexte d'évaluation des méthodes et dans l'implémentation des instructions de changement de comportement. Ces deux opérations doivent prendre en compte la gestion du partage des accointances avec des opérations différentes selon que le composant est sérialisé ou non sérialisé. En effet, un composant non sérialisé ne fait que lire les valeurs de ses accointances alors qu'un composant sérialisé a besoin de détenir les variables d'état depuis le début de l'évaluation de l'une de ses méthodes jusqu'à ce qu'il spécifie son prochain comportement. Les méta-composants, i.e. les composants des composants, doivent donc offrir une représentation de la machine composant différente de la machine acteur. Ce n'est pas en soi un obstacle à la conservation du modèle de réflexivité présenté ici. Une telle différenciation est à rapprocher de celle d'un méta-message avec un méta-acteur.

Les gestionnaires de composants soulèvent les mêmes problèmes: leur machine est différente de celle d'une machine acteur. Le propos d'un gestionnaire de composant étant de gérer la distribution des informations sur l'ensemble des composants d'une même dénotation, le méta de ces gestionnaires doit représenter cette gestion. Si les méta-composants ne se différencient des composants que par le comportement de certains d'entre eux (ceux qui ont à gérer les accointances de la dénotation), les méta des gestionnaires de composants présentent de plus grandes particularités. Non seulement les comportements de ces méta-gestionnaires sont spécifiques, mais aussi leurs accointances. Là encore, ces distinctions ne remettent pas en cause le modèle de réflexivité décrit.

Le modèle de réflexivité basé sur la notion de composants peut s'appliquer récursivement de méta-niveau en méta-niveau. Des adaptations des composants par défauts ainsi que des accointances qui modélisent la dénotation sont parfois nécessaires. Ces adaptations ne remettent cependant pas en cause l'architecture du modèle de réflexivité.

Comparaison entre modèle classique et composants

La spécification détaillée du lien causal permet d'englober dans le modèle de réflexivité à base de composants le modèle à base de méta-acteurs classiques.

Ces deux modèles se différencient essentiellement par la nature du lien causal qu'entretient le méta avec sa dénotation (Fig. 9.6 page suivante). Dans le modèle classique, un seul acteur, le méta-acteur, réifie l'ensemble des éléments de la machine acteur. Le lien entre méta-acteur et dénotation est alors unique. Les composants offrent plus de souplesse en permettant de

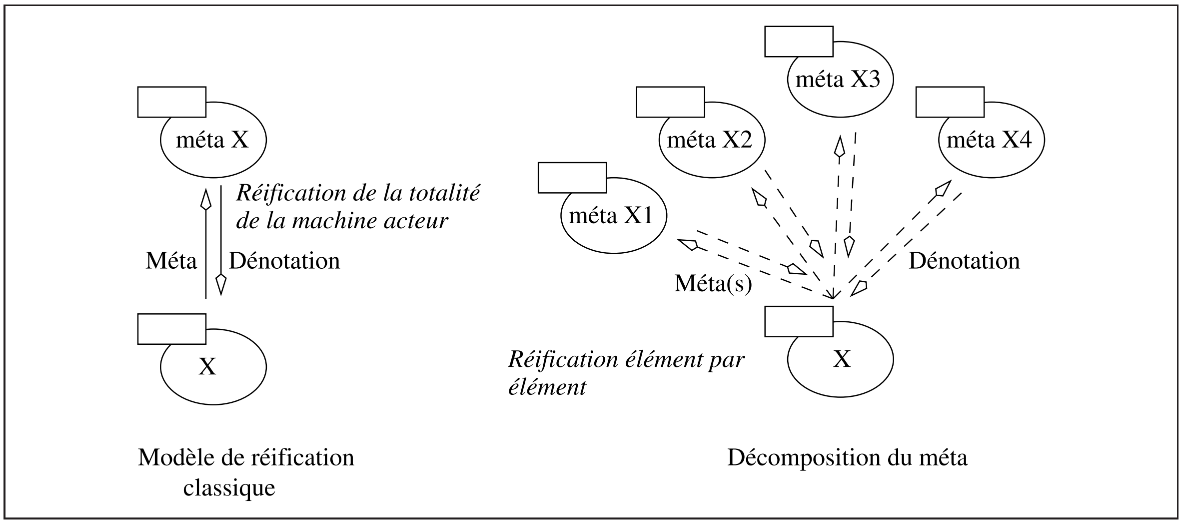

FIG. 9.6 – *Approche classique versus composants*

ne réifier que certains des éléments de la machine acteur. Mais, un même composant peut réifier plusieurs éléments de cette machine. En réifiant la totalité des éléments d'une machine acteur vers un seul composant, on établit une équivalence entre l'approche par composants et l'approche par méta-acteurs. Cette dernière peut alors être vue comme un cas particulier de la première.

L'équivalence qui vient d'être établie entre les deux modèles démontre la supériorité des composants sur les méta-acteurs. En plus de la souplesse et de la simplicité qu'ils introduisent dans la réflexivité, l'approche par composants est assez expressive pour intégrer directement le modèle à base de méta-acteurs.

Par contre les composants, par le nécessaire partage d'accointances qu'ils induisent, s'inscrivent en rupture avec le modèle de calcul acteur. Ce modèle de réflexivité s'oppose donc à la stricte uniformité linguistique habituellement présente dans les systèmes réflexifs. En cela il se montre moins adéquat que les méta-acteurs.

L'existence d'une méta-entité assurant la coordination des composants introduit une dimension réflexive supplémentaire qui n'existe pas dans le modèle à base de méta-acteurs. Cette dimension supplémentaire complexifie légèrement le modèle de réflexivité.

9.3.2 La réflexivité comme support à l'exploration de modèles réflexifs

Cette section décrit une implémentation du modèle de réflexivité proposé ci-dessus. Elle reprend l'essentiel du modèle sans toutefois en implémenter l'intégralité. En effet, nous ne décrirons que l'implémentation des composants sur un niveau. Cette implémentation permet donc la réification d'un acteur vers de multiples composants sans qu'il soit possible de réifier ces mêmes composants vers d'autres composants ou de réifier le gestionnaire des composants.

Schéma général de l'implémentation

L'utilisation de LActE/R comme support à l'implémentation du modèle de réflexivité à base de composants impose certaines contraintes et limitations. Les capacités réflexives de LActE/R s'articulent essentiellement autour des mécanismes acteurs. Elles n'incluent pas la possibilité de manipuler la relation méta/dénotation. Il faudrait pour cela que la réflexivité soit elle même réflexive, ce qui, à notre connaissance, n'est le cas à ce jour d'aucun système réflexif. Or l'implémentation des composants nécessite la manipulation de la relation méta/dénotation.

Dans LActE/R, la relation méta/dénotation prend la forme d'un lien bijectif entre le méta-acteur et l'acteur dénotation. C'est cette mise en bijection de deux acteurs que le modèle à base de composants veut rompre. De l'observation de la description précédente de ce modèle surgissent deux types d'acteurs: le gestionnaire des composants, et les composants à proprement parler. Du premier, nous savons qu'il a la charge de la gestion du partage des accointances ainsi que la coordination de l'enchaînement des actions de la machine acteur. Il occupe donc une position unique et un rôle central dans le fonctionnement d'un méta-système à base de composants. L'unicité de ce gestionnaire de composants ainsi que la nécessité qu'il a d'accéder aux accointances qui représentent les variables de la machine acteur conduisent naturellement à l'implémenter à l'aide d'un méta-acteur. Dans notre implémentation, le méta-acteur, au sens de LActE/R, d'un acteur A réifié vers un composant C servira de gestionnaire pour l'ensemble des composants réifiant A.

Si le méta-acteur de A est utilisé pour implémenter le gestionnaire des composants, les composants ne peuvent être eux mêmes des méta-acteurs de LActE/R. Or ces composants fonctionnent, du moins en grande partie, comme des acteurs. La différence la plus notable entre un composant et un méta-acteur réside dans la gestion du lien méta/dénotation: ils ne se substituent pas en totalité à la machine acteur par défaut, mais seulement en partie, ne réifiant que les aspects de cette machine qui sont relatifs à la fonction qu'ils occupent. Et c'est du gestionnaire des composants qu'ils obtiendront ces informations.

Un composant doit donc avoir la connaissance de son gestionnaire des composants. L'inclusion d'une telle information ne peut se faire directement au niveau du composant. Cela aurait pour conséquence de mélanger au même niveau d'interprétation des informations de différents niveaux i.e. les accointances du composant et une accointance supplémentaire qui contiendrait l'adresse du gestionnaire des composants. La gestion de cette connaissance est donc déléguée au méta-acteur (au sens de LActE/R) du composant.

En outre, le méta-acteur d'un composant aura en charge la gestion de la connaissance de la dénotation. La variable *denotation* habituellement utilisée à cet effet ne peut être manipulée grâce au protocole. Une autre variable doit donc être utilisée pour enrichir le contexte d'exécution des méthodes d'un composant de la connaissance de sa dénotation.

Le schéma de la figure (Fig. 9.7 page suivante) récapitule les différents points qui viennent d'être énoncés. Le méta-acteur qui remplit la tâche de gestionnaire des composants a, en plus

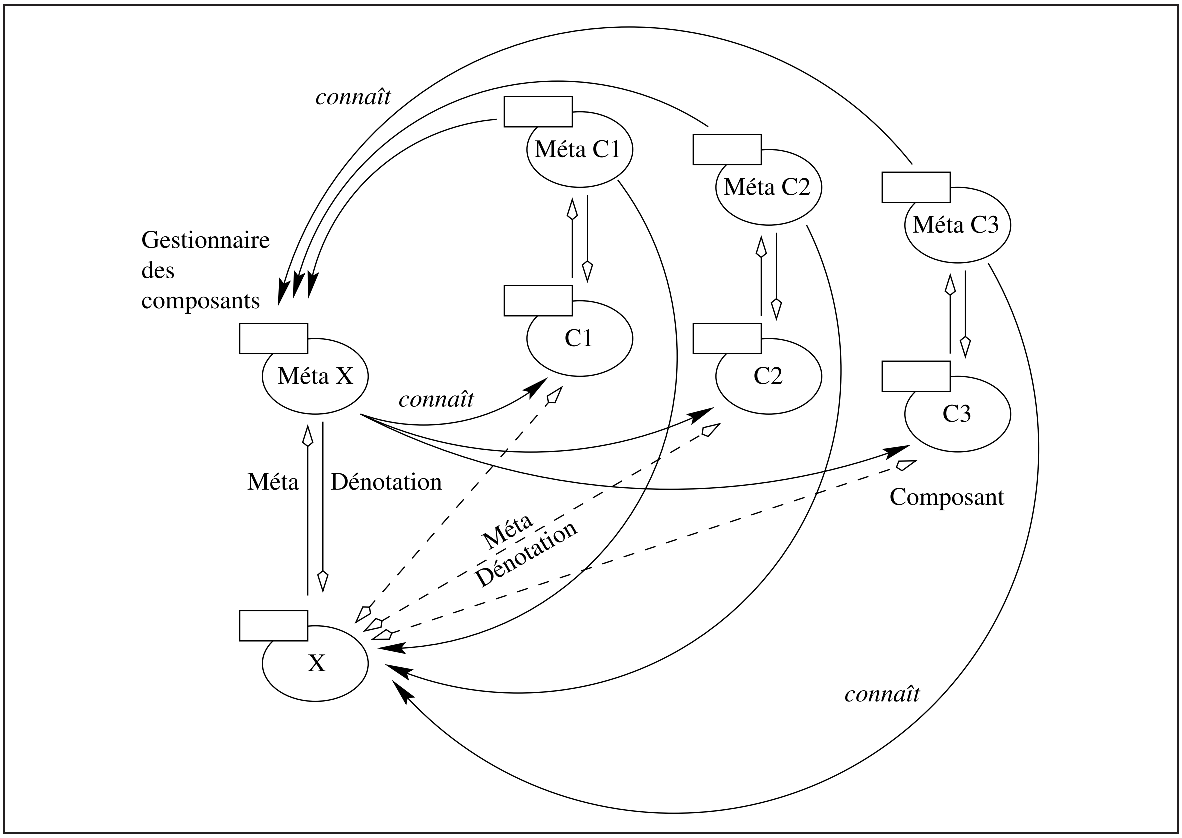

FIG. 9.7 – *Schéma d'implémentation des composants*

des connaissances propres à tout méta-acteur, la connaissance de l'ensemble des composants qui réifient sa dénotation. Chaque composant est réifié vers un méta-acteur qui détient la connaissance de son gestionnaire de composants ainsi que de l'adresse de la dénotation. À l'exception du lien méta/dénotation qui lie Méta-X à X, les liens qui apparaissent sur le schéma sont gérés dynamiquement à l'aide d'accointances. Ils sont donc modifiables à tout moment et offrent ainsi la possibilité d'ajouter ou de supprimer un composant en cours d'exécution.

Le gestionnaire des composants

Du gestionnaire des composants dépend le bon fonctionnement d'un système à base de composants. Pièce centrale de la machine acteur, il coordonne la circulation des informations entre les différents éléments qui la composent. Les deux tâches principales qui lui incombent sont le partage des variables d'état de la machine acteur ainsi que l'enchaînement des différentes activités.

L'accès aux variables d'état prend des formes différentes selon que le composant est un acteur sérialisé ou non. Dans le premier cas, le gestionnaire des composants doit garantir l'exclusion mutuelle sur une variable d'état détenue par un composant depuis sa lecture, effectuée au moment où le composant active l'une de ses méthodes, jusqu'à son écriture, effectuée au moment où le composant spécifie son prochain comportement. Entre ces deux événements, un seul composant à la fois est détenteur d'une variable d'état donnée. Rien n'interdit cependant la détention par deux composants différents de deux ensembles distincts de variables d'état.

La gestion en exclusion mutuelle des variables d'état est assurée au moyen de deux méthodes: *get-denotation-acqs* et *set-den-acqs*. À l'aide de la première, un composant demande la prise de contrôle d'un ensemble de variables d'état. Plusieurs composants pouvant demander la détention d'une même variable, le premier arrivé est le premier servi. Les autres sont mis en attente de la libération de la variable. Les demandes sont conservées dans une liste contenue dans l'accointance *waiting-den-acqs*. Lorsque la demande est acceptable, les valeurs des accointances demandées sont retournées et la liste des variables d'état détenues (*seized-den-acqs*) par des composants est enrichie des noms de ces mêmes variables d'état. Le code de la méthode *get-denotation-acqs* est le suivant:

```
  ; Demande de lecture avec prise de contrôle.
  ; den-acq-names contient la liste des noms des accointances
((:get-denotation-acqs den-acq-names)
   ; Si des demandes sont déjà en attente
   ; ou
   ; S'il existe dans la demande des accointances déjà
   ; détenues par d'autres composants
 (cond ((or waiting-den-acqs
             (intersection seized-den-acqs den-acq-names))
          ; La demande courant est mise en attente
          (setf waiting-den-acqs
                (append waiting-den-acqs
                        (list (cons forward-to den-acq-names)))))
          ; Sinon la demande est traitée
        (t
          ; La liste des accointances déjà détenues
          ; est mise à jour
          (setf seized-den-acqs (append den-acq-names seized-den-acqs))
          ; La liste des valeurs des variables d'état demandées
          ; est calculée à partir de la liste des noms
          ; et des valeurs des accointances
          ; et retournée au composant
          (result (process-acqs-names den-acq-names
                                      (list actor-name actor-behavior
                                            actor-acquaintances actor-proxy
                                            actor-reifier actor-messages
                                            actor-active)))))
 (update))
```

La mise en attente de la demande est conditionnée par un *ou* entre la présence de demandes en attente et la disponibilité des variables d'état demandées (lorsque l'intersection des noms des variables d'état déjà détenues par des composants et les noms des variables d'état demandées est l'ensemble vide). La première de ces conditions sert à éviter les cas de famine. Si une demande était satisfaite dès que l'ensemble des variables d'état demandées sont disponibles, un cas de famine pourrait se produire. Par exemple la demande de la totalité des variables d'état par un seul composant ne serait satisfaite que lors de l'absence d'aucune autre demande plus restreinte. Ce cas est évité en vérifiant auparavant la présence de demande mise en attente, la première de ces demandes devant être satisfaite avant qu'aucune autre ne le soit. Il en résulte une certaine diminution du parallélisme, une demande pouvant être mise en attente alors qu'elle est en état d'être directement satisfaite.

La récupération des valeurs des accointances est déléguée à la fonction *process-acqs-names*. L'utilisation de fonctions allège le code des méthodes et met en arrière plan des mécanismes de second ordre. Le code de cette fonction est:

```
(defun process-acqs-names(acqs-names acqs-values)
   (if acqs-names
     (cons (case (car acqs-names)
                 (:actor-name           (first   acqs-values))
                 (:actor-behavior       (second  acqs-values))
                 (:actor-acquaintances  (third   acqs-values))
                 (:actor-proxy          (fourth  acqs-values))
                 (:actor-reifier        (fifth   acqs-values))
                 (:actor-messages       (sixth   acqs-values))
                 (:actor-active         (seventh acqs-values))
                 (else nil))
           (process-acqs-names (cdr acqs-names) acqs-values))
     '()))
```

Elle établit une correspondance directe entre les noms des variables d'état et leur valeur. La liste des valeurs est construite récursivement sur la liste des noms des variables demandées.

La méthode symétrique de *get-denotation-acqs* est *set-denotation-acqs*. Elle assure l'accès en écriture des variables d'état et leur mise à disposition pour d'autres composants. Les deux paramètres qu'elle accepte sont la liste des noms des variables d'état et la liste de leur valeur. Notons que cette méthode n'est appelée que lorsqu'un composant effectue son **become**. Le code de cette méthode est:

```
((:set-denotation-acqs den-acq-names den-acq-values)
 ; Mise à jour des variables d'état
 (multiple-value-bind (actor-name actor-behavior
                       actor-acquaintances actor-proxy
                       actor-reifier actor-messages actor-active)
        ; Retourne la liste des valeurs des variables d'état
        ; en modifiant les valeurs de celle données par le
        ; composant
       (process-acqs-values den-acq-names den-acq-values
                            (list actor-name actor-behavior
                                  actor-acquaintances actor-proxy
                                  actor-reifier actor-messages
                                  actor-active))
        ; La liste des variables d'état détenues par des
        ; composants est mise à jour.
```

```
(setf seized-den-acqs
      (set-difference seized-den-acqs den-acq-names))
 ; Lorsque des demandes sont en attente de traitement
(when waiting-den-acqs
  (let* ((den-acq-names (cdar waiting-den-acqs)))
      ; Et si la première de ces demandes peut être
      ; satisfaite.
     (unless (intersection den-acq-names seized-den-acqs)
       ; Alors cette demande est traitée
       (send (process-acqs-names
                       den-acq-names
                       (list actor-name actor-behavior
                             actor-acquaintances actor-proxy
                             actor-reifier actor-messages
                             actor-active))
             :to (caar waiting-den-acqs))
       (setf seized-den-acqs (append den-acq-names seized-den-acqs)
             waiting-den-acqs (cdr waiting-den-acqs)))))
 ; Lorsque la demande ne peut être immédiatement traitée
 ; rien n'est fait.
(update)))
```

Après avoir mis à jour les valeurs des accointances qui représentent les variables d'état, le gestionnaire des accointances ajuste la valeur de *seized-den-acqs*. Sa dernière tâche consiste à satisfaire les éventuelles demandes en attente. Pour éviter les cas de famine, seule la première des demandes est examinée. Si cette demande ne peut être satisfaite, le gestionnaire des composants la laisse dans la file d'attentes. Ce n'est que lorsque l'ensemble des variables d'état demandées seront disponibles qu'elle sera satisfaite.

L'accès en simple lecture aux variables d'état est disponible à tout instant. Les valeurs demandées sont retournées dès réception de la demande que les variables d'état soient détenues par des composants ou non. Cette réponse immédiate est rendue possible par la non garantie de l'ordre d'arrivée des messages qui régit le modèle acteur [Agha 86]. Le code de cette méthode est:

```
((:read-denotation-acqs den-acq-names)
 (become)
 (result (process-acqs-names
                den-acq-names
                (list actor-name actor-behavior actor-acquaintances
                      actor-proxy actor-reifier actor-messages
                      actor-active))))
```

Les autres messages que reçoit le gestionnaire des composants correspondent aux activités de la machine acteur. Ces messages doivent être redistribués vers les composants correspondants. L'accointance *components* contient la liste des activités associées à leurs composants. Cette liste adopte la forme suivante:

```
([<mot-clé-de-l-activité> (<composant-principal> <composant-annexe>*)]+)
```

Chaque mot-clé désignant une activité donnée y est suivi de la liste des composants correspondants. Le premier élément de cette liste est le composant principal i.e. celui qui implémente l'activité en question. Il peut être suivi d'un ou de plusieurs composants annexes.

Le gestionnaire des composants qui réceptionne une activité, par exemple une activité de réception des messages, recherche la liste des composants correspondants à cette activité. Le message reçu est alors envoyé à l'ensemble des composants à l'aide d'un *multiple-send*. Le code de cette méthode est:

```
(message-pattern
   (become)
   (let* ((mesg-key (let ((key (first message-pattern)))
                          (if (and (eq key :become)
                                   (> (length message-pattern) 1))
                              :become-with-args
                              key)))
          (activity-components (cadr (member mesg-key components))))
      (multiple-send message-pattern
                     :to activity-components
                     :sender sender :forward-to forward-to
                     :type mtype))))
```

La recherche des composants concernés est précédée de l'examen du message qui décrit l'activité. Il s'agit d'identifier un éventuel :become avec paramètres pour le différencier d'un :become sans paramètre. Dans ce cas, le mot-clé est modifié en :become−with−args afin de permettre l'identification de cette activité à partir du seul mot-clé.

Une fois la liste des composants attachés à l'activité extraite de *components*, le message leur est envoyé. Cette méthode ne se préoccupe pas du statut des composants. Ce n'est que lors de l'ajout d'un composant que le traitement approprié est effectué.

La méthode :add−components assure l'ajout d'un composant à la machine acteur. Elle met à jour l'accointance *components* en fonction des activités que le composant veut réifier. Son code est:

```
((:add-component component activity)
 (when (member :default activity :test #'eq)
   (setf default-components
         (process-default-components component activity components)))
 (setf components
       (process-components component activity components))
 (update)
 (result t))
```

Pour des raisons de simplicité, le code de la fonction *process-default-components* n'est pas donné ici. Il est d'ailleurs facile à imaginer.

La fonction *process-components* calcule en fonction de l'adresse d'un composant et de la liste des activités qu'il souhaite réifier la liste associant les activités aux composants correspondants. Son code est:

```
(defun process-components(component activity components)
   (if components
     (let* ((components-activity-type (first  components))
            (activity-components      (second components))
            (component-activity-descr (member components-activity-type
                                              activity :test #'eq)))
       (cons components-activity-type
             (cons (if component-activity-descr
                       (if (eq (second component-activity-descr) :other)
                         (cons (first activity-components)
                               (cons component (rest activity-components)))
                         (cons component (rest activity-components)))
                       activity-components)
                   (process-components component activity
                                       (cddr components)))))
     '()))
```

L'ensemble du comportement d'un gestionnaire de composants pour un acteur sérialisé est obtenu en rassemblant les méthodes précédentes au sein d'un **def-behavior**. Ce comportement a, en plus des accointances habituelles des méta-acteurs, les accointances qui servent à la gestion des composants. Son code en version abrégée est:

```
(def-behavior a-meta-components-coordinator
              (default-components components waiting-den-acqs seized-den-acqs) t
   (declare (is-a a-meta-actor-serialized))
   ((:read-denotation-acqs den-acq-names)  ...)
   ((:get-denotation-acqs den-acq-names)   ...)
   ((:set-denotation-acqs den-acq-names den-acq-values) ...)
   ((:add-component component activity) ...)
   (message-pattern ...))
```

L'unique contrainte sur l'ordre des méthodes est que la méthode d'aiguillage des activités se trouve en dernière position afin qu'elle ne filtre pas les autres méthodes du gestionnaire des composants.

Le code décrit correspond à celui du gestionnaire des composants d'un acteur sérialisé. Le cas d'un acteur non sérialisé est facilement déduit de ce code en restreignant la liste des variables d'état aux accointances d'un acteur non sérialisé et en incluant non plus le comportement du méta-acteur d'un acteur sérialisé mais celui du méta-acteur d'un acteur non sérialisé. Notons que, dans les deux cas, le gestionnaire des composants est un acteur sérialisé.

Implémentation des composants

L'implémentation des composants passe par des méta-acteurs qui modifient sensiblement le fonctionnement de leur machine acteur. La présence de ces méta-acteurs permet d'adapter les mécanismes aux spécificités des composants:

– Leurs méthodes sont évaluées avec deux paramètres supplémentaires, l'adresse de la dénotation et celle du gestionnaire des composants.

- Les valeurs de leurs accointances dépendent en partie des valeurs des variables d'état réifiées.

- Le **become** des composants sérialisés doit mettre à jour les valeurs des variables d'état détenues.

L'ensemble de ces particularités se traduit par la modification de deux des méthodes de la machine acteur d'un composant: la méthode :evaluate et la méthode :become.

La méthode :evaluate doit, en plus de son comportement habituel, gérer la lecture des valeurs des variables d'état réifiées et l'ajout des deux paramètres supplémentaires au contexte d'évaluation des méthodes. Le code de la méthode reprend celui de la méthode :evaluate d'un méta-acteur classique auquel sont ajoutées les lignes qui gèrent la lecture des variables d'état et l'ajout des paramètres supplémentaires. Ce code est le suivant:

```
((:evaluate a-method a-message)
 (multiple-value-bind (mesg-to mesg-sender mesg-forward-to
                                mesg-message mesg-take mesg-type)
                      (ask [:get-values] :to a-message)
    ; Lecture des noms des variables d'état réifiées
    (let* ((reified-acqs (ask [:get-reify-acqs] :to actor-behavior))
       ; Lecture des valeurs des variables d'état réifiées
       (reified-acqs-values
          (if component-manager
             (ask [:get-denotation-acqs reified-acqs] :to component-manager)
             reified-acqs)))
       ; Calcul des valeurs des accointances en positionnant les
       ; valeurs des variables d'état réifiés en tête de liste
       (setf actor-acquaintances
             (append reified-acqs-values
                     (nthcdr (length reified-acqs-values) actor-acquaintances)))
       (let ((actor-ctxt (process-actor-ctxt mesg-to mesg-sender mesg-forward-to
                                             self actor-proxy mesg-type mesg-take
                                             a-method a-message t
                                             actor-acquaintances actor-reifier)))
          (update)
          (evaluate-compiled-actor-method a-method
             (process-compiled-actor-method-arguments
                        actor-ctxt mesg-message
                        ; Ajout de l'adresse du gestionnaire
                        ; des composants et de celle de la
                        ; dénotation au contexte d'évaluation
                        ; de la méthode.
                        :other-values
                           (if (meta-actor-method? a-actor-method)
                              '()
                              (list component-manager component-denotation))))))))
```

Le texte qui apparaît en gris correspond aux lignes de code différentes de la méthode classique. Remarquons que, pour des raisons de simplicité, nous nous restreignons au seul cas des méthodes compilées.

La méthode :become est surchargée de manière à mettre à jour les valeurs des variables d'état détenues par le composant. La liste de ces valeurs est extraite de la liste des valeurs des accointances avant d'être envoyée au gestionnaire des composants. Le code de cette méthode est:

```
((:become)
 (when component-manager
   (let ((reify-acqs (ask [:get-reify-acqs] :to actor-behavior)))
     (send [:set-denotation-acqs reify-acqs
                                 (subseq actor-acquaintances 0 (length reify-acqs))]
           :to component-manager)))
 (send [:become] :to self :take (next-actor-method) :forward-to forward-to)))
```

La réification d'une dénotation par un composant est obtenue à l'aide de la méthode :reify. La demande de réification doit être directement expédiée au méta-acteur du composant. Le code de la méthode est:

```
((:reify component-denotation)
 (let ((component-manager
             ; Recherche de l'adresse du gestionnaire des composants
             ; Ou le méta-acteur de la dénotation existe
             ; Ou il faut le créer
             (or (ask [:meta] :to component-denotation)
                 (ask [:reify-to
                        (if (ask [:get-serialized]
                                  :to (behavior-of component-denotation))
                            a-meta-components-coordinator-initializer
                            an-us-meta-components-coordinator-initializer)
                        ()] :to component-denotation))
      ; Demande de rattachement du composant au gestionnaire
      ; des composants de la dénotation
     (ask [:add-component denotation
                (ask [:get-reify-activity] :to actor-behavior)]
          :to component-manager)
     (update)
     (result t)))
```

La méthode :connect-with sert uniquement à la gestion des composants par défaut lors de la phase d'initialisation. Elle permet une connexion plus rapide des composants avec leur gestionnaire et évite une boucle infinie sur la réification des composants. Son code est:

```
((:connect-with component-manager component-denotation)
 (update)
 (result t))
```

Le comportement global du méta-acteur d'un composant sérialisé est obtenu en rassemblant les méthodes décrites précédemment au sein d'un même comportement. La version abrégée de ce comportement est:

```
(def-behavior a-meta-meta-component-serialized
                        (component-manager component-denotation) t
  (declare (is-a a-meta-actor-serialized))
  ((:connect−with component-manager component-denotation) ...)
  ((:reify component-denotation) ...)
  ((:evaluate a-method a-message) ...)
  ((:become) ...)
```

Les composants non sérialisés adoptent un code sensiblement identique au code précédent. Ils ne disposent cependant pas de la méthode :become et la lecture des valeurs des variables d'état ne se fait plus à travers :get−den−acqs mais :read−den−acqs.

Construction d'un composant

Les composants, bien que présentant de fortes similitudes avec les acteurs, se distinguent des acteurs de base par un contexte d'évaluation des méthodes enrichi, la présence d'une déclaration qui décrit le lien causal, et des comportements particuliers. Le fonctionnement de LActE/R, basé sur la compilation des méthodes, exige la prise en compte de ces spécificités dès la création des composants. Seul un constructeur de comportements de composants dédié autorise l'introduction de ces caractéristiques.

Un constructeur de comportement est un acteur pourvu d'une méthode :build−behavior. Cette méthode qui prend en paramètres le nom du comportement, la liste des noms de ses accointances, le booléen de sérialisation, la liste des déclarations et la liste des méthodes sous forme textuelle doit retourner un comportement [2]. L'acteur *meta-component-builder* de comportement *a-meta-component-builder* sera notre constructeur de comportement. La création de ce constructeur s'effectue de la façon suivante:

```
(def-behavior a-meta-component-builder() ()
  ((:build−behavior behavior-name behavior-acqs behavior-serialized
                    behavior-declarations behavior-script)
                 ; Création du comportement d'un composant comme étant un acteur spécialisé
    (result (new-anonymous  a-behavior-of-meta-component ()
                            behavior-name behavior-acqs behavior-serialized
                            (append (make-meta-actor-methods behavior-serialized)
                                    (ask [:make−methods behavior-serialized
                                                 behavior-acqs behavior-script ()]
                                         :to self))
                            (get-reify-activity behavior-declarations)
                            (get-reify-acqs behavior-declarations))))
  ((:make−methods serialized acqs script methods)
   (if script
     (let* ((method-def (car script))
            (pattern    (car method-def))
            (body       (cdr method-def))
            (method (new-anonymous a-actor-method ()
```

2. Le protocole détaillé de la création des comportements est disponible dans la présentation de LActE/R.

```
                                          pattern body serialized
                                          (compile-matching pattern)
                                          (compile-actor-method-body
                                                    serialized acqs method-def
                                                     ; Le contexte d'évaluation de la méthode
                                                     ; est étendu
                                                    :other-parameters
                                                       '(meta-of-denotation
                                                         component-denotation)))))
         (send [:make-methods serialized acqs (cdr script)
                                          (append methods (list method))]
               :to self :forward-to forward-to)))
       (result methods))))
```

```
(new meta-component-builder a-meta-component-builder ())
```

Comparativement au constructeur par défaut, *a-meta-component-builder* se distingue par l'adoption d'un comportement de comportement propre et par l'ajout de deux paramètres au contexte d'évaluation des méthodes du composant (les différences sont en gris). Le comportement *a-behavior-of-meta-component* est utilisé afin de conserver une description du lien causal. Cette description du lien causal ne sera utilisée que lors de la réification d'une dénotation vers le composant. Les deux paramètres additionnels du contexte d'évaluation des méthodes donnent accès à l'adresse du gestionnaire des composants et à celle de la dénotation depuis le corps d'une méthode. Le composant dispose ainsi des informations relatives à l'entité réifiée et à l'ensemble des composants auquel il appartient.

Les comportements des composants se distinguent par leur gestion de deux accointances supplémentaires, *reify-activity* et *reify-acqs*, et par une méthode de création qui les réifie automatiquement. Le comportement des comportements des composants est:

```
(def-behavior a-behavior-of-meta-component
              (reify-activity reify-acqs) ()
  (declare (is-a a-behavior))
  ; Création d'un composant
  ((:create a-name a-proxy some-acquaintances)
   ; Création de l'acteur composant
   (let ((actor (build-actor self a-name a-proxy behavior-serialized
                             some-acquaintances)))
      ; Création du méta-acteur du composant
      (if behavior-serialized
        (ask [:reify-to a-meta-meta-component-serialized () nil nil]
             :to actor)
        (ask [:reify-to a-meta-meta-component-unserialized () nil nil]
             :to actor))
      ; Retour de l'adresse du composant créé
      (result actor)))
   ; Lecture des noms des variables d'état réifiés
  ((:get-reify-acqs)      (result reify-acqs))
   ; Lecture des activités réifiés
  ((:get-reify-activity) (result reify-activity)))
```

Les méthodes :get−reify−acqs et :get−reify−activity offrent un accès en lecture à la description du lien causal. La méthode :create enrichit la création par défaut de la création automatique du méta-acteur du composant. Ainsi, un composant est toujours créé en même temps que son méta-acteur. Lors du fonctionnement d'un composant, ces informations sont exploitées par son méta-acteur.

Initialisation et composants par défaut

L'ensemble d'un système basé sur les composants ne fonctionne que s'il a été correctement initialisé. Au début de son existence, il ne se traduit que par la présence de deux entités: la dénotation et un composant. Le gestionnaire des composants n'est alors pas encore créé. Pour que le système à base de composants puisse correctement opérer, le gestionnaire des composants doit être créé et initialisé.

La création du gestionnaire des composants est déléguée au premier des composants désireux de réifier une dénotation. La méthode :reify d'un composant commence par rechercher son gestionnaire des composants. Pour cela elle demande à sa dénotation de lui communiquer l'adresse de son méta-acteur. Si l'adresse est *nil*, alors la dénotation ne dispose pas encore de méta-acteur et doit être réifiée vers un méta-acteur adoptant le comportement d'un gestionnaire de composants. Cette réification n'a pas directement lieu vers le comportement d'un gestionnaire des composants tel qu'il a été présenté dans les paragraphes précédents. Elle s'effectue vers un comportement dont l'unique raison d'être est l'initialisation des différentes accointances du gestionnaire des composants. Cette initialisation a lieu dès la réception d'une demande d'ajout de composant à la description de la machine acteur. Le code d'un tel comportement est:

```
(def-behavior a-meta-components-coordinator-initializer() t
  (declare (is-a a-meta-actor-serialized))
  ((:add−component component activity)
   (multiple-value-bind (components default-components)
                        (make-components actor-name t self
                                         denotation component activity)
     (become a-meta-components-coordinator ()
             actor-name actor-behavior actor-acquaintances actor-proxy
             actor-reifier actor-messages actor-active
             default-component components '() nil)
     (result t))))
```

L'essentiel de l'initialisation n'apparaît pas dans le code précédent. Elle est déléguée à la fonction *make-components* qui a pour tâche de créer une description de la machine acteur à base de composants par défaut. Le code de cette fonction est:

```
(defun make-components(name serialized self denotation
                            component component-activity)
  (let* ((components-behavior
            (if serialized
               (list :receive−message   a-sa-receive-message
```

```
                     :become-with-args    a-sa-become-with-args
                     :become              a-sa-become
                     :update              a-sa-update
                     :evaluate            a-sa-evaluate
                     :activate-message    a-sa-activate-message
                     :no-matching-method  a-sa-no-matching-actor-method
                     :delegate            a-sa-delegate)
               (list :receive-message     a-ua-receive-message
                     :evaluate            a-ua-evaluate
                     :activate-message    a-sa-activate-message
                     :no-matching-method  a-ua-no-matching-actor-method
                     :delegate            a-ua-delegate)))
      (activity nil)
      (default-components
        (mapcar #'(lambda (elt)
                    (if (keywordp elt)
                      (setf activity elt))
                      (ask [:create
                             (gensym (concatenate 'string "MC-"
                                                  (string name) "."
                                                  (string activity)))
                             nil
                             (make-list
                              (length (ask [:get-acquaintances] :to elt)))]
                      :to elt))))
                  components-behavior)))
      (values
        (mapcar #'(lambda (elt)
                    (cond ((keywordp elt) (setf activity elt))
                          (t (ask [:connect-with self denotation]
                                  :to (ask [:meta] :to elt))
                             (let ((component-activity
                                     (member activity component-activity)))
                               (if component-activity
                                (if (eq (second component-activity) :other)
                                  (list elt component)
                                  (list component))
                                (list elt))))))
                  default-components)
  default-components)))
```

Un exemple de composant par défaut est le composant chargé de la réception des messages pour un acteur sérialisé. Un tel composant suit une description similaire aux autres composants à l'exception du qualifiant de réification de son activité principale. Ce qualifiant devient :default au lieu de :main. Le gestionnaire des composants adopte un traitement spécifique pour ce type de réification. Ce traitement lui permet de conserver une liste des composants par défaut. Cette liste qui n'est pas utilisée ici à pour objet de permettre la suppression des composants.

Outre cette particularité, le corps de l'unique méthode du composant reprend celui de la méthode de réception des messages d'un méta-acteur classique de LActE/R. La description du lien causal comprend, en plus de l'activité de réception des messages, la réification des variables d'état utilisées par la méthode. Le code de ce composant est:

```
(def-behavior a-sa-receive-message(actor-messages actor-active) t
  (declare (constructor meta-component-builder)
           (reify '((activity      :receive-message :default)
                    (acquaintances :actor-messages  :actor-active))))
  ((:receive-message a-message)
   (cond (actor-active
          (setf actor-messages (append actor-messages (list a-message)))
          (update))
         (t
          (setf actor-active t)
          (update)
          (send [:activate-message a-message]
                :to meta-of-denotation)))))
```

Chacune des activités de la machine acteur est décrite selon le même procédé. Il y a donc autant de composants par défaut qu'il y a d'activité. Ce nombre de composant est augmenté des composants qui gèrent différemment ces activités selon que la dénotation est sérialisée ou non.

Exemple

Pour illustrer le fonctionnement d'un système à base de composants, nous reprenons l'exemple de la trace des messages en l'adaptant au monde des composants.

Une trace des messages reçus est, sous forme de composant, plus concise et plus directe. L'ensemble du comportement ne décrit que les aspects relatifs au mécanisme de trace sans qu'il soit fait appel, directement ou indirectement, aux autres mécanismes de la machine acteur. Tout l'effort du programmeur reste concentré sur la mise en œuvre du mécanisme de trace sans qu'il n'ait à prendre en considération les mécanismes par défaut de la machine acteur. Le code d'un composant qui trace les messages reçus est:

```
(def-behavior a-trace-received-message() ()
  (declare (constructor meta-component-builder)
           (reify '((activity :receive-message :other))))
  ((:receive-message a-message)
   (send [:get-message] :to a-message :forward-to
         (continuation ()
           ((mesg-message)
            (send [(format nil "~&Receive ~a" mesg-message)]
                  :to screen))))))
```

Le mécanisme de trace est implémenté par un composant non sérialisé. Un tel composant s'utilise aussi bien pour tracer les messages reçus par un acteur sérialisé que ceux reçus par un acteur non sérialisé. C'est un avantage notable de cette approche par composant qui réduit significativement la quantité de code à écrire pour implémenter des mécanismes réflexifs en minimisant le nombre de cas à considérer.

La présence d'une déclaration de constructeur résulte des contraintes imposées par le modèle de réflexivité de LActE/R. Une macro habillerait avantageusement cette déclaration en la cachant et offrirait une forme syntaxique plus apte à différencier les comportements des composants de ceux des acteurs. L'utilisation directe d'une déclaration de constructeur est cependant suffisante pour notre propos.

La seconde déclaration décrit le lien causal entre le composant et sa dénotation. Le mécanisme de trace ne réifie que l'activité de réception des messages. La réification d'aucune autre information n'est nécessaire. Le mécanisme de trace ne représentant pas pour la machine acteur l'activité principale de réception des messages, la réification de cette activité est de type :other. Le composant ne sera que notifié de la réception d'un message.

L'unique méthode de notre composant consiste exclusivement en l'affichage du contenu des messages reçus. Contrairement au premier exemple de trace, la présence de lignes de code pour connecter le mécanisme de trace à la machine acteur n'est plus nécessaire dans le corps de la méthode.

La mise en place du mécanisme de trace sur une factorielle s'opère de la manière suivante:

```
; Création de la factorielle
(new fact a-fact ())
; Création du composant traceur de messages
(new trm a-trace-received-message ())
 ; Établissement de la connexion entre le composant
 ; et sa dénotation
(send [:reify fact] :to (ask [:meta] :to trm))
```

Une fois la connexion établie, les messages reçus par la factorielle seront affichés à l'écran.

L'un des attraits de l'approche par composants tient en sa capacité à combiner différents composants au sein d'une même machine acteur. C'est par cette capacité que notre modèle offre une solution partielle au problème de composition soulevé par J.C. Pazzaglia [Pazzaglia 97]. Si, en plus d'être tracés, les messages reçus doivent être comptés, il suffit de créer un composant compteur de messages que l'on ajoutera à la machine acteur.

Un composant compteur de messages est un composant sérialisé dont l'accointance mémorise le nombre de messages reçus. La description du lien causal est identique à celle du traceur de messages. Le comportement d'un tel composant est:

```
(def-behavior a-received-message-counter(number-of-messages) t
  (declare (constructor meta-component-builder)
           (reify '((activity :receive-message :other))))
    ; Comptabilisation des messages reçus
  ((:receive−message a-message)
   (become a-received-message-counter () (+ number-of-messages 1)))
    ; Lecture du nombre de messages reçus
  ((:get−number−of−messages) (become) (result number-of-messages)))
```

L'ajout d'une capacité de comptage des messages reçus s'opère de manière identique à l'ajout de n'importe quel composant:

```
; Création du compteur de messages
(new rmc a-received-message-counter () 0)
 ; Établissement de la connexion entre le composant
 ; et sa dénotation
(send [:reify fact] :to (ask [:meta] :to rmc))
```

Après établissement de la connexion, l'ensemble des messages reçus par la factorielle seront non seulement affichés à l'écran, grâce à la présence du traceur de messages, mais aussi comptabilisés, grâce à la présence du compteur de messages. Ces deux composants coexistent en même temps au sein de la même machine acteur. Et la combinaison de ces deux fonctionnalités n'est pas le résultat de l'écriture, explicite ou implicite, d'un comportement les associant.

Limitations

L'implémentation qui vient d'être décrite impose de sérieuses limitations sur le modèle. Elle n'est qu'une ébauche de ce que devrait être un système à base de composants. Les principales limitations sont dues à l'absence de mécanisme de gestion des modifications dynamiques du système.

La première de ces limitations est l'impossibilité pour un acteur sérialisé de modifier la description de son lien causal lors d'un changement de comportement. Or, si les descriptions de liens causaux sont attachées aux comportements, le lien causal entre un composant et sa dénotation devrait pouvoir être modifié à chaque changement de comportement.

La seconde limitation concerne la transformation d'un acteur sérialisé en acteur non sérialisé. Une telle transformation impose la modification de l'ensemble de la machine acteur. En particulier, le traitement des messages est différent selon qu'un acteur est sérialisé ou non. Le passage d'un acteur sérialisé à un acteur non sérialisé suppose donc une modification complète de la description de la machine acteur. Les liens causaux entre les différents composants et la dénotation ne peuvent alors plus être correctement respectés.

Enfin, il n'est pas possible de supprimer un composant. Une telle opération correspondrait à une réflexion partielle de la machine acteur. L'existence de cette opération poserait des problèmes identiques à celle d'un changement de lien causal lors de la spécification d'un nouveau comportement.

9.3.3 Comparaison avec d'autres modèles

Les usines de ReActalk [Giroux 93] présentent des similitudes avec notre modèle en particulier en tant que système de décomposition du méta d'un acteur. Chaque étape du fonctionnement d'un acteur y est représentée par un acteur distinct appartenant à une organisation encore nommée usine. Le dynamisme de l'organisation qui supporte ces acteurs de méta-niveau confère au modèle des capacités de composition similaires à notre modèle.

Il y a cependant un écart important sur la finalité et la réalisation du modèle. Pour S.

Giroux, la décomposition du méta relève d'une approche organisationnelle du concept de réflexivité dans laquelle l'usine regroupe des entités interchangeables. Si notre modèle rejoint le sien sur le caractère interchangeable des composants, l'objectif de la décomposition du méta n'est pas, pour nous, l'introduction de la notion d'organisation au méta-niveau, mais de rendre possible, par une spécification détaillée du lien causal, la réification élément par élément de la machine acteur. En cela, notre modèle constitue une exploration poussée du concept de localité avec pour finalité, l'augmentation des performances, une implémentation optimisée du modèle à base de composants devant permettre la localisation des mécanismes réflexifs introduits aux seules parties de la machine acteur qui y font appel.

Il résulte de ces différences d'objectifs des différences de rôles des éléments appartenant à l'usine. Si ces éléments sont, pour S. Giroux, les constituants d'une unité plus importante, l'usine, ils apparaissent dans la notion de composants comme des entités individuelles réifiant des sous-ensembles de la machine acteur. Ils ne servent donc pas tant la décomposition du méta, mais la décomposition de la dénotation.

La notion de composant se combinant par ajout aux mécanismes par défaut ne fait pas partie intégrante de ReActalk. Cette notion, due à J.C. Pazzaglia [Pazzaglia 97], confère au modèle à base de composants une flexibilité que n'atteint pas ReActalk.

Dans sa thèse, J.C. Pazzaglia traite du problème de la composition. Les solutions qu'il met en place se rapprochent de notre modèle. Cependant, si les composants apportent des solutions au problème de la composition, le point de vue présenté ici diffère sensiblement de celui de J.C. Pazzaglia. Avec nos composants, il ne s'agit pas tant de traiter le problème de la composition que celui de l'efficacité du système. Les composants se présentent comme un outil qui permet la réification des éléments de la machine acteur.

Le modèle à base de composants est très proche de CodA [Mcaffer 95]. Tout deux utilisent une décomposition du méta avec cependant une démarche et des objectifs différents. CodA résulte d'une approche de bas en haut du méta: le système est conçu à partir de l'assemblage de briques de base plutôt que de la réification d'un système préexistant propre à notre modèle. La notion de méta-composant est, dans CodA, directement liée à cet assemblage de briques de base alors que nos composants s'articulent autour de la description de la machine acteur donnée par le comportement d'un méta-acteur classique. Cette approche donne à CodA une souplesse sur la description des interactions entre les méta-composants inexistante dans notre modèle.

Enfin, le modèle de réflexivité par composant peut faire penser à la réflexivité de groupe telle qu'elle est proposée dans Act/R [Watanabe 90]. Cependant, si l'ensemble constitué du gestionnaire de composant et des composants constitue bien un groupe, il ne réifie pas un groupe d'acteurs mais un seul acteur. De plus, ce groupe d'acteurs ne décrit pas les mécanismes généraux d'un système d'acteurs mais le mécanisme d'un unique acteur. Dans notre modèle, la relation méta/dénotation n'est donc (n, n) mais $(n, 1)$.

9.3.4 Conclusion

En tant qu'exemple d'application des capacités réflexives de LActE/R, l'approche par composants démontre l'intérêt de la réflexivité pour explorer d'autres modèles de réflexivité. L'architecture d'un système à base de composant ne s'adapte pas directement au modèle de réflexivité à la base de LActE/R. La souplesse de notre interprète permet néanmoins de l'étudier.

En lui même, le modèle à base de composants se révèle être une évolution souhaitable de LActE/R. Il introduirait dans notre interprète un degré supplémentaire de souplesse et de performance. Il doit cependant être auparavant mieux étudié tant au niveau de son architecture que de ses limitations.

9.4 Conclusion

L'ensemble des exemples présentés dans ce chapitre montre l'aptitude de LActE/R à modéliser des problèmes assez différents. Avec les exemples disponibles dans la thèse de J.C. Pazzaglia [Pazzaglia 97], ils participent à la validation du modèle de réflexivité de LActE/R.

De manière plus générale, ces exemples démontrent l'intérêt même de la réflexivité. Nous avons pu introduire différents mécanismes en nous appuyant sur la réflexivité. Le premier exemple démontre que la réflexivité n'est pas toujours un facteur de perte de performances. Elle peut très bien améliorer les performances du système. Le second exemple démontre l'aptitude de la réflexivité à introduire des mécanismes linguistiques supplémentaires dans un interprète. Le dernier exemple, en permettant d'explorer un modèle de réflexivité à base de composants, montre que la réflexivité peut aussi servir de support à l'implémentation de modèles de réflexivité.

Chapitre 10

Conclusion et Perspectives

Dans cette thèse, un modèle de réflexivité pour une couche acteur a été défini et implémenté après une étude approfondie de la réflexivité. Ce modèle est issu d'une approche pragmatique de la réflexivité. Un caractère essentiel de notre approche était le maintien d'un lien constant avec la pratique.

Les nombreux travaux sur la réflexivité ont porté essentiellement sur les modèles de réflexivité. Mais chacun d'entre eux ne s'intéresse qu'à un seul modèle mis en place dans un contexte unique. En nous intéressant à plusieurs modèles de réflexivité et plusieurs paradigmes linguistiques nous avons pu donner une vue plus générale de ce qu'est la réflexivité, mettre en évidence les problèmes de la réflexivité ainsi que les liens entre modèle, implémentation et efficacité dans les systèmes réflexifs.

L'étude de la réflexivité a porté sur les modèles de réflexivité et leurs implémentations. La plupart des interprètes réflexifs de langages fonctionnels ont été analysés. Le rôle de la tour d'interprètes a été clarifié et la distance entre l'implémentation de la tour et l'approche philosophique soulignée. Lors de cette étude, il est apparu que ces interprètes réflexifs ne disposaient pas de la réflexivité globale alors que leur modèle de réflexivité le permettait et que l'efficacité de l'exécution reste le problème majeur qui fait obstacle à une large utilisation de la réflexivité.

$I_{r\infty}$ montre que la réflexivité globale relève non pas du modèle mais de l'implémentation. Aucune construction linguistique spécifique n'est nécessaire à l'introduction de ce type de réflexivité. La réflexivité globale est introduite à partir d'une implémentation complète de la connexion causale qui lie un interprète aux définitions contenues dans l'environnement global. L'interprète, basé sur une tour active, est assez inefficace. Mais c'est aussi celui dont les capacités réflexives sont les plus étendues et les plus dynamiques: tout ce que la réflexivité permet d'y faire peut y être fait à tout moment et la réflexivité globale permet de modifier l'intégralité des mécanismes de l'interprète.

Les exemples d'utilisation de la réflexivité globale ont montré le degré de souplesse supplémentaire qu'elle introduit dans la programmation réflexive. Elle permet de s'abstraire plus facilement des limites qu'impose la représentation de l'interprète. Des applications que l'utilisation de la seule réflexivité locale n'autorise pas sont ainsi rendues possibles.

Enfin, le modèle de la tour active est plus proche de la philosophie initiale de la réflexivité. Il réalise pleinement l'empilement de plusieurs niveaux d'interprétation. Ce n'est plus un unique flot de calcul qui parcourt l'ensemble d'une tour fictive au gré des appels de lambdas réflexives, mais plusieurs interprètes qui s'interprètent l'un l'autre.

La réflexivité globale est rarement disponible dans les interprètes réflexifs. Il nous semble qu'elle devrait être mieux étudiée. Des techniques d'optimisation adaptées à cette technique doivent cependant être développées afin de pallier au manque d'efficacité dû à l'existence de plusieurs niveaux d'interprétation.

L'évaluation partielle a permis de réduire une tour d'interprètes à un niveau unique d'exécution. L'application de cette technique à la réflexivité se traduit par un gain très important en efficacité. Lors de ce processus, les modifications induites par les constructions réflexives sont figées. Le résultat de l'évaluation partielle d'un interprète réflexif est très encourageant sur le gain en performance qui devient comparable à celui de l'évaluation d'expressions équivalentes écrites dans le langage qui sert à l'implémentation.

L'application de l'évaluation partielle à la réflexivité globale devrait permettre de concilier les avantages de ce type de réflexivité avec une certaine efficacité de l'exécution. Les modifications que suppose la programmation réflexive basée sur l'utilisation de la réflexivité globale sont plus lentes que celles dues à l'utilisation des lambdas réflexives. Cependant, l'utilisation de l'évaluation partielle dans le cadre de la réflexivité globale suppose un traitement adéquat des effets de bord.

L'utilisation de cette technique sur des mécanismes autres que l'interprétation des expressions du langage devrait permettre d'améliorer plus largement l'efficacité des systèmes réflexifs. Les méta-objets sont des candidats idéaux. L'intérêt de l'évaluation partielle dépend, là encore, du rapport entre son coût et la rapidité des modifications effectuées par le programme réflexif.

L'objectif principal de cette thèse était de définir et d'implémenter un modèle de réflexivité pour un langage d'acteurs. LActE/R, interprète achevé et exploité par un utilisateur averti, J. C. Pazzaglia, en constitue la matérialisation. Les exemples d'utilisation fournis dans cette thèse, couplés à ceux présents dans la thèse de J. C. Pazzaglia [Pazzaglia 97], participent à la validation de notre approche.

LActE/R se trouve à l'intermédiaire des deux approches présentées précédemment par ses performances et ses capacités réflexives. Il montre que l'approche basée sur les interprètes reste valide même sans technique particulière d'optimisation. Le lien entre modèle, implémentation et efficacité est beaucoup plus fort que dans les autres interprètes. Le problème des performances est pris en compte dès la définition du modèle de réflexivité essentiellement sous la forme d'un principe fondateur, le principe de localité. L'implémentation ne fait que traduire les principes issus du modèle et n'utilise pas de technique d'optimisation spécifique. Le résultat de notre approche est un interprète simple à utiliser, dont le modèle couvre un large champ d'applications et dont l'efficacité reste compatible avec un usage régulier (Cf. annexe).

En ayant pris en compte les critiques d'un utilisateur averti, l'interprète LActE/R a atteint un certain degré de maturité. Certaines questions sont pourtant restées ouvertes:

- Le premier travail à effectuer sur LActE/R est d'affiner le modèle de réflexivité. Le découpage de l'interprète peut être amélioré. Le choix de ce découpage est d'ailleurs une des questions les plus délicates de la définition d'un système réflexif.

- La version actuelle de LActE/R n'est pas distribuée. Il serait intéressant de la distribuer sur un réseau de stations de travail et de voir quels sont les problèmes spécifiques à la distribution d'un modèle réflexif. De nombreuses questions surgissent comme le traitement d'une tour de méta-acteurs ou la répartition des acteurs par rapport aux réificateurs.

- Le modèle de réflexivité basé sur les composants semble être une approche souhaitable de LActE/R. Il nécessite une étude plus approfondie afin d'en déterminer les limites et la validité. Cependant, les composants présentent une architecture idéale pour la mise en place d'optimisations. Dans ce modèle, tout ce qui n'est pas explicitement réifié par l'utilisateur peut être implémenté par des fonctions. Et les composants qui procèdent par ajout de fonctionnalités ne nécessitent pas la réification de leur dénotation mais un simple envoi de message de la part de l'interprète LActE/R.

- L'application de l'évaluation partielle à LActE/R est une des voies possibles pour en améliorer les performances. Plutôt que d'appliquer l'évaluation partielle à un interprète de LActE/R elle serait appliquée aux méta-acteurs. Chaque tour du système pourrait ainsi être réduite. Une telle approche soulève de nombreux problèmes. L'évaluation partielle n'a presque pas abordé le problème de la concurrence et le modèle de calcul de LActE/R fait appel de façon intensive à la concurrence. D'autre part, l'application de l'évaluation partielle à LActE/R suppose que les méta-acteurs changent lentement. Or les acteurs sérialisés modifient continuellement leur comportement. Un graphe de l'ensemble des comportements adoptés par un acteur peut cependant être déterminé. Il semble donc possible d'appliquer cette technique aux méta-acteurs. Des langages à objets moins dynamiques peuvent eux en bénéficier plus facilement.

Modèle, implémentation et efficacité ont constitué le carrefour de cette thèse. Dans chacun de nos interprètes ces questions sont régulièrement revenues. Et, lorsque nous avons choisi de mettre en avant l'un de ces points, les choix effectués ont toujours eu une influence plus ou moins directe sur les autres. Il apparaît donc difficile de les dissocier.

En généralisant le système de description de la relation méta/dénotation mis en place dans l'exemple sur les composants, l'antagonisme qui existe entre les différentes architectures pourrait être résolu. Un même interprète réflexif accepterait ainsi aussi bien une relation bijective entre le méta et sa dénotation, que d'autres approches. La manipulation de la relation méta/dénotation par le programmeur augmenterait notablement les possibilités offertes par la réflexivité.

En rendant le système extensible, la réflexivité est appelée à jouer un rôle important dans la conception des langages informatiques. À partir d'un langage réduit à un noyau essentiel, des concepts plus élaborés peuvent être ajoutés et les mécanismes de base modifiés. L'utilisateur d'un système réflexif dispose des outils qui lui permettent de l'adapter à ses besoins spécifiques. Aussi, il nous paraît important de renverser la démarche adoptée jusqu'ici: plutôt que d'introduire la réflexivité dans des systèmes existants, ne devrions-nous pas concevoir des systèmes qui soient réflexifs dès leur naissance et qui prennent la réflexivité comme point de départ de leur conception?

Bibliographie

[Abelson 91] H. Abelson et al. – *Revised 4 Report on the Algorithmic Language Scheme*, édité par William Clinger et Jonhathan Rees, Novembre 1991. Dedicated to the memory of ALGOL 60.

[Agha 86] Gul Agha. – *Actors: A Model of Concurrent Computation in Distributed Systems.* – Thèse de PhD, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1986, 144p. Published by MIT Press.

[Asai 93] Kenichi Asai, Satoshi Matsuoka et Akinori Yonezawa. – Duplication and Partial Evaluation to Implement Reflective Languages. *OOPSLA Workshop on Reflection and Meta-Level Archictectures.* ACM. – 1993.

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

[Bondorf 93] Anders Bondorf. – *The Similix 5.0 Manual.* – Rapport technique, Department of Computer Science, DIKU, University of Copenhagen, Denmark, Mai 1993.

[Briot 87] Jean-Pierre Briot et Akinori Yonezawa. – Inheritance and Synchronisation in Concurrent OOP. *Proceedings of ECOOP'87*, éd. par Springer-Verlag, pp. 33–40. – 1987.

[Briot 89] Jean-Pierre Briot. – Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. *Proceedings of the ECOOP'89*, pp. 109–129. – Nottingham, England, Juillet 1989.

[Carle 92] Patrice Carle. – *Mering IV: un langage d'acteurs pour l'IAD intégrant objets et agents par réflexivité compilatoire.* – Thèse de Doctorat d'Université, Université Paris VI, Avril 1992.

[Cointe 84] Pierre Cointe. – *Implémentation et interprétation des langages orientés objets. Applications aux langages Smalltalk, ObjVlisp et Formes.* – Thèse de Doctorat d'État, Université de Paris VI, Décembre 1984.

[Cointe 87a] Pierre Cointe. – The ObjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System, pp. 155–176, dans *Meta-level Architectures and Reflection.* – North-Holland, Amsterdam, édité par P. Maes et D. Nardi, 1987.

[Cointe 87b] Pierre Cointe. – MetaClasses are First Class: The ObjVlisp Model. *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87).* ACM, pp. 156–167. – Orlando, Florida, 1987.

[Consel 90] Charles Consel. – *The Schism Manual, Version 1.0.* – Yale University, New Haven, Connecticut, Décembre 1990.

[Consel 93] Charles Consel et Olivier Danvy. – Tutorial Notes on Partial Evaluation. *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993.* ACM, pp. 493–501. – New York: ACM Press, 1993. (tutorial presented at the conference).

[Consel 96] Charles Consel et François Noël. – A General Approach for Run-Time Specialization and its Application to C. *popl96.* – 1996.

[Danvy 88] Olivier Danvy et Karoline Malmkjaer. – Intensions and Extensions in a Reflective Tower. *ACM Conference on Lisp and Functional Programming*, pp. 327–341. – 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.

[Dourish 92] Paul Dourish. – *Computational Reflection an CSCW Design.* – Technical report EPC-92-102, Rank Xerox EuroParc, Mars 1992.

[Ferber 89] Jacques Ferber. – *Objets et agents: une étude des structures de représentation et de communications en Intelligence Artificielle.* – Thèse de Doctorat d'État, Université de Pierre et Marie Curie, Paris IV, Juin 1989.

[Ferber 91] Jacques Ferber et Patrice Carle. – Actors and Agents as Reflective Concurrent Objects: A Mering IV Perspective. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, n° 6, Novembre/Décembre 1991, pp. 1420–1436.

[Friedman 84] Daniel P. Friedman et Mitchell Wand. – Reification: Reflection without Metaphysics. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 331–347. – Août 1984.

[Futamura 71] Y. Futamura. – Partial Evaluation of Computation Process - An approach to Compiler-Compiler. *Systems, Computers, Controls*, vol. 2(5), 1971, pp. 45–50.

[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.

[Giroux 93] Sylvain Giroux. – *Agents et sytèmes, une nécessaire unité*. – Thèse de PhD, Département d'informatique et de recherche opérationnelle, Faculté des Arts et Sciences, Université de Montréal, Mars 1993.

[Goldberg 83] A. Goldberg et D. Robson. – *Smalltalk-80, the Language and its Implementation*. – Massachusetts, Addison-Wesley, 1983.

[Hassas 93] Salima Hassas. – *GMAL, Un Modèle Réflexif pour la Conception de Systèmes d'Intelligence Artificielle Distribuée*. – Thèse de Doctorat d'Université, LISI, Université Claude Bernard, Lyon I, Septembre 1993.

[Hewit 69] Carl Hewit. – PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot. *First International Joint Conference on Artificial Intelligence*, pp. 295–301. – Washington D.C., Août 1969.

[Hewit 73] Carl Hewit. – A Universal Modular ACTOR Formalism for Artificial Intelligence. *Proceedings of the third IJCAI*, pp. 235–245. – Stanford, California, 1973.

[Ichisugi 92] Yuuji Ichisugi, Satoshi Matsuoka et Akinori Yonezawa. – RbCl: A Reflective Object-Oriented Concurrent Language without a Run Time Kernel. *Proceedings of IMSA'92 International Workshop on Reflection and Meta Level Architectures*. – 1992.

[Ingalls 78] D. H. H. Ingalls. – The SMALLTALK-76 Programming System Design and Implementation. *Proceedings of the 5th POPL*, pp. 9–17. – Tuscon, Arizona, 1978.

[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.

[Kiczales 91] Gregor Kiczales, Jim des Rivières et Daniel G. Bobrow. – *The Art of the Metaobject Protocol*. – MIT Press, 1991.

[Kiczales 93] Gregor Kiczales, John Lamping et Anurag Mendhekar. – *What a Metaobject Protocol Based Compiler can do for Lisp.* – Rapport technique, Xerox PARC, Embedded Computation Area, Décembre 1993.

[Kleene 52] S. C. Kleene. – *Introduction to Metamathematics.* – Princeton, NJ: D. van Nostrand, 1952.

[Lamping 92] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr. et Erik Ruf. – An Architecture for an Open compiler. *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures.* – 1992.

[Lee 93] Shinn-Der Lee et Daniel P. Friedman. – Quasi-Static Scoping: Sharing Variable Bindings Across Multiple Lexical Scopes. *Twentieth ACM Symposium on Principles of Programming Languages*, pp. 479–492. – 1993.

[Maes 87] Pattie Maes. – *Computational Reflection.* – Thèse de PhD, Vrije Universiteit Brussel, 1987. Technical report 87.2.

[Masini 89] Gérald Masini, Amadeo Napoli, Dominique Colnet, Daniel Léonard et Karl Tombre. – *Les langages à objets.* – Paris, InterEditions, 1989.

[Masuhara 92] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe et Akinori Yonezawa. – Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92).* ACM, pp. 127–144. – Vancouver, Canada, Octobre 1992.

[Masuhara 94] Hidehiko Masuhara, Satoshi Matsuoka et Akinori Yonezawa. – An Object-Oriented Concurrent Reflective Language for Dynamic Resource Management in Highly Parallel Computing. *IPSJ SIG Notes, 94-PRG-18*, pp. 57–64. – Juillet 1994.

[Masuhara 95] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai et Akinori Yonezawa. – Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95).* ACM, pp. 300–315. – ACM SIGPLAN Notices, Octobre 1995.

[Matsuoka 90] Satoshi Matsuoka et Akinori Yonezawa. – Metalevel Solution to the Inheritance Anomaly in Concurrent Object-Oriented Languages. *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming.* – Ottawa, Canada, Octobre 1990.

[Matsuoka 91] Satoshi Matsuoka, Takuo Watanabe et Akinori Yonezawa. – Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. *Proceedings of ECOOP'91.* – Geneva, Switzerland, Juillet 1991.

[Mcaffer 95] Jeff McAffer. – Meta-level Programming with CodA. *in Proceedings of OOPSLA'95.* – Octobre 1995.

[Michel 90] Claude Michel. – *Un Modèle de Raisonnement Distribué.* – Rapport de DEA, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice-Sophia Antipolis, Septembre 1990.

[Michel 92] Claude Michel et Jean-Pierre Regourd. – Représentation des Connaissances et Programmation par Acteurs. *Acte de SEIGE'92.* – Nice, France, Septembre 1992.

[Michel 93] Claude Michel et Jean-Pierre Regourd. – Demact: Un exemple de raisonnement distribué. *Premières journées francophones IAD&SMA*, pp. 193–216. – Toulouse, France, Avril 1993.

[Mogensen 96] Torben Æ. Mogensen et Peter Sestoft. – Partial evaluation, dans *Encyclopedia of Computer Science and Technology.* – Septembre 1996. Version 1.01 of 1996-04-09.

[Pazzaglia 97] Jean-Christophe Pazzaglia. – *Méthodologie et outils pour la programmation réflexive dans les langages à objets.* – Thèse de Doctorat d'Université, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice-Sophia Antipolis, 1997.

[Rao 91] Ramana Rao. – Implementational Reflection in Silica. *Proceedings of ECOOP 91.* – Geneva, Switzerland, 1991.

[Regourd 94] Jean-Pierre Regourd, Claude Michel et Jean-Christophe Pazzaglia. – *LActE v1.4, Manuel de Référence.* – Rapport technique, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice-Sophia Antipolis, 1994.

[Ruf 93a] Erik Ruf. – Partial Evaluation in Reflective System Implementation. *OOPSLA Workshop on Reflection and Meta-Level Archictectures.* – 1993.

[Ruf 93b] Erik Ruf. – *Topics in Online Partial Evaluation.* – Thèse de PhD, Department of Computer Science, Stanford University, California, Février 1993. Published as technical report CSL-TR-93-563.

[Shibayama 86] Etsuya Shibayama et Akinori Yonezawa. – *ABCL/1 User's Guide*, 1986. Internal Memo.

[Simmons II 92a] John Wiseman Simmons II et Daniel P. Friedman. – *Language extension via First-class Interpreters.* – Technical report #362, Computer Science Department, Indiana University, Septembre 1992.

[Simmons II 92b] John Wiseman Simmons II et Daniel P. Friedman. – *A Reflective System is as Extensible as its Internal Representations: An Illustration.* – Technical report #366, Computer Science Department, Indiana University, Octobre 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.

[Steele jr 90] Guy L. Steele JR. – *Common Lisp : The Language.* – Digital Press, 1990. Second edition.

[Steels 86] L. Steels. – *The KRS Concept System.* – Technical report 86-1, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Brussel, Belgium, 1986.

[Tanaka 90] Tomoyuki Tanaka. – Actor Reflection without Meta-object. *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, pp. 253–272. – Ottawa, Canada, Octobre 1990.

[Taura 94] Kenjiro Taura, Satoshi Matsuoka et Akinori Yonezawa. – ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation -. *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, éd. par G. Blelloch, M. Chandy et S. Jagannathan. pp. 275–292. – American Mathematical Society, 1994.

[Wand 88] Mitchell Wand et Daniel P. Friedman. – The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, 1988, pp. 11–37.

[Watanabe 88] Takuo Watanabe et Akinori Yonezawa. – Reflection in an Object-Oriented Concurrent Language. *Proceedings of Object-Oriented Programming Sys-*

tems, Languages and Applications (OOPSLA'88). ACM, pp. 306–315. – ACM Press, 1988.

[Watanabe 90] Takuo Watanabe et Akinori Yonezawa. – An Actor-Based Metalevel Architecture for Group-Wide Reflection. *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, pp. 275–290. – Ottawa, Canada, Octobre 1990.

[Weise 90] Daniel Weise. – *Graph as an Intermediate Representation for Partial Evaluation.* – Technical report csl-tr-90-421, Computer Systems Laboratory, Stanford University, Stanford, CA, Mars 1990.

[Weise 91] Daniel Weise, Roland Conybeare, Erik Ruf et Scott Seligman. – Automatic Online Partial Evaluation. *Functional Programming Languages and Computer Architecture (LNCS 523)*, éd. par editor J. Hughes. pp. 165–191. – Cambridge, MA, Août 1991.

[Yokote 92] Yasuhiko Yokote. – The Apertos Reflective Operating System: The Concept and its Implementation. *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*. ACM. – Vancouver, Canada, Octobre 1992.

[Yonezawa 86a] A. Yonezawa, E. Shibayama, Y. Honda et T. Takada. – Modelling and Programming in a Concurrent Object-Oriented Langage ABCL/1, dans *Object Oriented Concurrent Programming.* – MIT Press, édité par A. Yonezawa et M. Tokoro, 1986.

[Yonezawa 86b] Akinori Yonezawa, Jean-Pierre Briot et Etsuya Shibayama. – Object-Oriented Concurrent Programming in ABCL/1. *Proceedings of the first OOPSLA'86*, pp. 258–268. – Portland, Oregon, 1986.

Annexe A

LActE/R en chiffres

L'objet de cette annexe est de donner quelques valeurs chiffrées qui reflètent le coût de la réflexivité dans LActE/R. Les tests menés à cette fin permettent d'apprécier la relative efficacité de LActE/R en prenant pour point de comparaison un calcul identique effectué sur la version non réflexive de LActE. Le calcul effectué doit donc être identique sur les deux versions de LActE.

Nous avons choisi de prendre comme exemple la factorielle, la suite de Fibonacci ainsi qu'une expression faisant la somme des deux premiers, $(300! + U_{15})$. Ces choix sont partiaux et ne correspondent pas à l'ensemble des programmes possibles en LActE/R. Ils permettent cependant de se faire une idée juste du coût de la réflexivité.

Les calculs ne sont pas optimisés. L'important ici n'est pas de comparer la version la plus rapide mais des algorithmes qui sont similaires dans le monde fonctionnel et dans le monde acteur.

La version Lisp de la factorielle utilisée pour les tests est:

```
(defun fact(n)
  (if (< n 2)
      1
      (* n (fact (1- n)))))
```

La version LActE de la factorielle utilisée pour les tests est:

```
(def-behavior a-fact() ()
  ((n) (if (< n 2)
           (result 1)
          (send [(1- n)] :to self :forward-to
                (continuation ((cont forward-to))
                              ((value) (send [(* n value)] :to cont)))))))
```

```
(new fact a-fact ())
```

La version Lisp de Fibonacci utilisée pour les tests est:

```
(defun fib(n)
  (if (< n 2)
```

```
      1
      (+ (fib (1- n)) (fib (- n 2)))))
```

La version LActE de Fibonacci utilisée pour les tests est:

```
(def-behavior a-fib() ()
  ((0) (result 1))
  ((1) (result 1))
  ((n)
   (let ((adder (jointure ((reply-to forward-to))
                          ((v1) (v2))
                          (send (list (+ v1 v2)) :to reply-to))))
     (send (list (1- n)) :to self :forward-to adder)
     (send (list (- n 2)) :to self :forward-to adder))))
```

```
(new fib a-fib ())
```

La version LActE de l'expression $(300! + U_{15})$ utilisée pour les tests est la suivante:

```
(def-behavior a-test1() ()
  ((:test)
   (let ((join (jointure ((cont forward-to))
                         ((v1) (v2))
                         (send [(+ v1 v2)] :to cont))))
     (send [300] :to fact :forward-to join)
     (send [15] :to fib :forward-to join))))
```

```
(new test1 a-test1 ())
```

La version fonctionnelle de cette expression est obtenue par :

```
(+ (fact 300) (fib 15))
```

Afin de pouvoir apprécier la part due aux processus dans le coût de LActE, nous avons mesuré le temps de calcul d'une factorielle écrite en Common Lisp mais faisant appel aux processus. Le fonctionnement de cette factorielle mime celui des acteurs sans utiliser pour autant l'envoi de message ou la structure acteur. L'analogie entre les acteurs et cette factorielle s'arrête à la mise en œuvre des processus et à la transmission des résultats intermédiaires. Le calcul de la factorielle en Lisp avec processus est basé sur les trois fonctions suivantes:

```
(defun insure-active-process(process)
  (loop
    (unless (mp::process-active-p process) (return))
    (mp::process-allow-schedule))
  (mp::process-enable process))
```

```
(compile 'insure-active-process)
```

```
(defun compute-fact(n res cont)
  (if (< n 2)
      (progn
        (setf (car res) 1)
```

```
        (funcall cont))
      (progn
        (let ((vres '(no-result)))
          (mp::process-run-function "noname" #'compute-fact (1- n) vres
                                    #'(lambda ()
                                        (mp::process-run-function "noname"
                                          #'(lambda ()
                                              (setf (car res) (* (car vres) n))
                                              (funcall cont)))))))))
```

```
(compile 'compute-fact)
```

```
(defun fact(n)
  (let ((res '(no-result)))
    (mp::process-run-function "noname" #'compute-fact n res
                              (let ((pself mp::*current-process*))
                                #'(lambda ()
                                    (insure-active-process pself))))
    (mp::process-disable mp::*current-process*)
    (car res)))
```

```
(compile 'fact)
```

La première des fonctions, *insure-activate-process*, active un processus après s'être assuré qu'il est bien désactivé. Elle est utilisée par la continuation de *fact* pour synchroniser le retour du résultat final.

L'essentiel du calcul est réalisé par *compute-fact*. Cette fonction présente de fortes analogies avec le calcul d'une factorielle acteur dont elle reprend le principe en s'affranchissant des envois de messages.

La dernière fonction, *fact*, ne sert qu'à synchroniser l'appel de la factorielle. Elle reproduit un schéma de fonctionnement identique au **ask** des acteurs. Le temps nécessaire au calcul peut être ainsi mesuré.

Lorsqu'un acteur est dit «avec méta-acteur» ou «réifié», il est réifié vers un méta-acteur de comportement *a-meta-actor-unserialized*. La réification de cet acteur est obtenue par l'expression suivante:

```
(ask [:reify-to a-meta-actor-unserialized ()] :to <acteur>)
```

Le réificateur utilisé pour les tests est le suivant:

```
(new my-send a-send ())
(new my-join a-jointure ())
(new my-cont a-continuation ())
(new my-result a-result ())
```

```
(setf my-reifier (make-reifier :send my-send
                               :jointure my-join
                               :continuation my-cont
                               :result my-result))
```

L'installation du réificateur est obtenue à l'aide de l'expression suivante:

```
(ask [:set−reifier my-reifier] :to <acteur>)
```

La réification des acteurs ainsi que le réificateur correspondent aux définitions du protocole. L'utilisation de comportement appartenant au protocole simplifie la démarche sans que le coût du calcul d'un mécanisme modifié par l'utilisateur soit pris en compte.

La plateforme logicielle et matérielle ayant servi aux tests est la suivante: Allegro CL 4.3 sur SparcStation 5 (modèle à 70Mhz avec 16Mo de mémoire vive et 64Mo de swap) sous SunOS 4.1.3_U1.

Lors des tests, un seul utilisateur était connecté à la machine et les seuls logiciels à tourner étaient Allegro CL et SunOS (en particulier, aucun système de fenêtrage n'été en marche).

La version non réflexive de LActE utilisée est la version (1.5) et la version réflexive de LActE/R est la version (0.46).

Chaque résultat donné est un temps en milliseconde et correspond à une moyenne prise sur 10 tests du temps CPU.

Les résultats obtenus pour la factorielle sont les suivants:

factorielle	300	1000	2000
Lisp	25,2	270,0	1371,7
Lisp et processus	730,1	2731,6	6481,6
LActE	909,9	3300,0	7500,1
LActE/R	1276,7	4444,8	9798,4
Avec méta-acteur	3619,7	12291,7	25658,2
Avec réificateur	3100,0	10801,7	22353,4
Avec méta-acteur et réificateur	5636,6	18741,6	38771,5

Les résultats obtenus pour Fibonacci sont les suivants:

Fibonacci	10	15	200
Lisp	0,0	0,0	13,3
LActE	526,7	6050,0	65441,9
LActE/R	716,7	8009,8	88781,7
Avec méta-acteur	2154,9	23405,0	242424,7
Avec réificateur	1886,8	20978,5	217538,5
Avec méta-acteur et réificateur	3238,2	35641,8	367208,4

Le calcul de $(300! + U_{15})$ donne les résultats suivants:

$(300! + U_{15})$	temps (ms)
fonctionnel	21,6
LActE	6918,4
LActE/R	9268,6
test1 réifié	9388,3
fact réifié	11695,0
fib réifié	24871,8
fact et fib réifiés	27360,0
test1 avec réificateur	9393,4
fact avec réificateur	10898,5
fib avec réificateur	22263,2
fact et fib avec réificateur	23989,9
test1 réifié avec réificateur	9209,9
fact réifié avec réificateur	13609,8
fib réifié avec réificateur	37840,0
fact et fib réifiés avec réificateur	41945,0

L'analyse des résultats obtenus révèle que:

1. Le coût d'un calcul effectué sur la version réflexive est de 35% supérieur à celui de la version non réflexive. LActE/R est donc, par ses performances, relativement proche de LActE.

2. Le calcul de la factorielle Lisp basée sur les processus permet de dire que près de 83% du temps de calcul est utilisé par les processus.

3. Lorsque les versions avec acteurs réifiés sont comparées aux versions sans acteur réifié sur LActE/R, il apparaît que le coût d'un méta-acteur est de 2,8. Lorsqu'on dote un acteur d'un réificateur, ce coût est de 2,5. Il passe à 4,3 pour un acteur réifié muni d'un réificateur non standard. Même ce dernier résultat reste acceptable lorsqu'il est comparé à l'interprète réflexif de notre étude sur la réflexivité globale.

4. Le dernier tableau démontre l'effet du principe de localisation. Les temps de calcul de *test1*, qu'il soit réifié ou non, avec réificateur ou non, sont quasiment identiques. L'effet de l'utilisation de la réflexivité reste donc bien localisé à *test1*.

5. Le calcul de l'expression avec la factorielle et Fibonacci réifiées et munies d'un réificateur est de 4,5.

L'ensemble de ces résultats doit être interprété avec précaution. Il est bien entendu possible d'obtenir des coûts supérieurs en utilisant d'autres méta-acteurs ou d'autres combinaisons des entités réifiées et non réifiées. Cependant, ces tests permettent d'apprécier la relative efficacité de LActE/R qui, sans utiliser de technique particulière, arrive à des résultats acceptables pour un interprète réflexif.

Résumé

Dans cette thèse, nous définissons et implémentons un modèle de réflexivité pour un interprète de langage d'acteurs. Notre modèle s'inscrit dans une approche pragmatique de la réflexivité dans laquelle les aspects pratiques et l'efficacité sont des critères primordiaux.

Notre démarche s'appuie sur une étude poussée de la réflexivité et sur un dialogue permanent avec un utilisateur averti.

L'étude de la réflexivité explore les modèles et les implémentations des interprètes réflexifs de langages fonctionnels. Ces derniers sont les premiers à avoir bénéficié du concept de réflexivité et constituent une plate-forme naturelle à l'étude des mécanismes linguistiques.

Ils nous ont permis de clarifier le rôle de l'implémentation de la connexion causale dans la réflexivité globale. Ce type de réflexivité étend les possibilités offertes par la réflexivité en libérant le programmeur des limitations imposées par la représentation.

L'évaluation partielle, technique présentée comme idéale pour résoudre le problème de l'efficacité dans les interprètes réflexifs, est étudiée en détail. L'implémentation d'un interprète réflexif qui l'utilise a permis de mettre en évidence les apports mais aussi les difficultés de mise en œuvre de cette technique.

Notre objectif principal se matérialise par l'interprète réflexif LActE/R. Le modèle de réflexivité de LActE/R a été défini à partir de l'étude de la réflexivité, et d'un principe fondateur, le principe de localité. L'application de ce dernier s'est traduite par l'introduction d'opérations qui contrôlent dynamiquement le nombre de niveaux d'interprétation au sein des tours de méta-acteurs et par une localisation de l'impact de la réflexivité aux seules entités explicitement réifiées. Il a ensuite été affiné et rationalisé à partir des critiques et des propositions d'un utilisateur averti.

LActE/R se caractérise par sa simplicité, son large champ d'application et une efficacité compatible avec une utilisation régulière.

Mots-clés

Réflexivité, modèle de réflexivité, réflexivité globale, connexion causale, évaluation partielle, méta-acteur, efficacité, uniformité linguistique.