Classification de la réflexion de comportement basée sur les monades

Examen Prédoc
deuxième partie

François-Nicola Demers

Juillet 1996*

Table des matières

1 Introduction **3**

2 Réflexion **6**

2.1 Langages à *la 3-Lisp* 6

2.2 Langages extensibles 9

2.3 Système de réécriture et tour réflexive 11

3 Programmation monadique **15**

3.1 Monades computationnelles 18

3.2 Retour à l'exemple 21

3.3 Implantations, méthodologie et applications 22

4 Problématique et cadre du travail **24**

*Version corrigée après mars 1997.

4.1 Algèbre de la réflexion 24
4.2 Démarche de la recherche 28

5 Théorie des catégories et monades **30**
5.1 Introduction 31
5.2 Théorie du calcul de Moggi 32

6 Mise en perspective **34**
6.1 Monades et réflexion : les origines 35
6.2 Autres formalismes sémantiques 37

7 Conclusion **37**
7.1 Contributions 38
7.2 Echéancier 38

A Un exemple en programmation : les adjonctions **39**

B Concepts de base : catégorie, foncteur, etc. **45**

C Monades et triplets de Kleisli **49**

D Monoïdes **51**

1 Introduction

> Good general theory does not search for the maximum generality, but for the right generality. [Lan71]
>
> *Sanders Mac Lane*

Dans le domaine de la conception de langages de programmation, la réflexion de comportement cherche à donner aux programmeurs des outils permettant de changer la sémantique même d'un programme en cours de son exécution.

Le concept de réflexion a ses origines les plus lointaines en fondement des mathématiques et en philosophie du langage. En mathématique, il a été relié entre autres au problème de cohérence des systèmes logiques qui tentaient de devenir les fondements universels des mathématiques. Le célèbre paradoxe de Russell (dont on retrouve plusieurs variantes explicatives comme le paradoxe du menteur) est peut-être une des premières manifestations formelles de la réflexion et des problèmes qu'elle engendre. Par la suite, plusieurs recherches ont été mises en route pour tenter de se débarrasser des paradoxes en gardant (si possible) la puissance des définitions circulaires découvertes, c'est-à-dire la réflexion. Plus tard, les fameux **principes de réflexion** [Kre68] ont été définis et reconnus importants. Ils mettent en évidence une dichotomie nécessaire pour éviter les paradoxes. Elle caractérise la séparation entre un niveau qu'on dit de base et un méta-niveau qui a généralement un certain pouvoir d'action sur le niveau de base. Les principes de réflexion expriment justement les actions possibles entre les deux niveaux. Cette dichotomie est similaire à celle qui sert à distinguer les logiques dites du premier ordre des logiques d'ordre supérieurs.

En informatique, la réflexion a pris son essor par les travaux de Brian Smith [Smi82, Smi84] sur le développement d'une terminologie propre au domaine. Ses travaux lui ont permis la création d'un langage appelé 3-Lisp qui est basé principalement sur le concept de **tour réflexive**. Celle-ci est en quelque sorte une extension de la dichotomie expliquée plus haut où il est question d'un niveau de base et d'un empilement de méta-niveaux, tous ayant un pouvoir d'action sur les niveaux immédiatement inférieurs. 3-Lisp est plus précisément une implantation directe de la tour réflexive infinie, c'est-à-dire un empilement de méta-interprètes en nombre fini mais non borné à l'avance[1], tous écrits dans un même langage où chaque in-

[1] Dans la littérature sur la réflexion, on parle généralement d'un empilement *poten-*

terprète évalue l'interprète immédiatement en dessous de lui. Chaque interprète représente exactement un niveau de la tour réflexive. Par la suite, les recherches se sont orientées vers le problème de l'implantation de tels langages[2] et de leur formalisation mathématique.

Une implantation naïve de 3-Lisp s'avérerait très peu efficace, car chaque méta-niveau nécessaire pour l'exécution d'un programme donné correspond à un interprète complet du langage et cet interprète doit interpréter tous les niveaux qui se trouvent en dessous de lui. Les chercheurs ont donc tenté d'optimiser leur implantation 3-Lisp par des techniques générales comme l'évaluation partielle [DKM91], mais sans grand succès. Des techniques d'optimisation plus fines doivent être expérimentées. Du côté théorique, des chercheurs ont tenté de formaliser la tour réflexive par la sémantique dénotationnelle. Les résultats n'ont pas été satisfaisants car la propriété de compositionnalité chère à cette sémantique n'a pu être respectée.

Les travaux sur la réflexion, bien que populaires, n'ont pas été suffisants pour démontrer l'utilité de cette nouvelle discipline en méthodologie de programmation et en implantation de langages. Les recherches sont donc parties dans plusieurs directions. Aujourd'hui, nous retrouvons des systèmes réflexifs dans presque tous les paradigmes de programmation [DM95]. Il y en a tellement qu'il est difficile de reconnaître une terminologie commune sur la réflexion reconnue par tous les chercheurs du domaine. De plus, les questions et problèmes essentiels sont encore méconnus par plusieurs ou ils sont malencontreusement mélangés à des concepts de moindre importance.

Du côté de la méthodologie, un besoin de classification se fait sentir dans le but de découvrir ce qu'il y a de commun entre les systèmes réflexifs actuels pour simplifier la terminologie, s'attaquer à l'essence des problèmes et éviter la duplication du travail [DM95]. Et d'un côté plus formel, il est apparu une volonté de développer une théorie générale de la réflexion dans laquelle les chercheurs pourraient décrire leurs résultats théoriques [MF96] à l'aide d'outils de raisonnement adaptés à cette discipline. Nous pensons évidemment au raisonnement équationnel qui est une sorte de technique de réécriture de termes de haut niveau qui présente les cas où la substitution de

tiellement infini. Bien que le qualificatif "infini" ne soit peut-être pas adéquat, il met en évidence le fait que l'implantation correspond "du mieux qu'elle peut" (des choix d'implantation sont nécessaires) à la tour réflexive qui est, elle, infinie. Nous dirons qu'une tour est *non-bornée à l'avance* s'il est impossible (dans tous les cas) de déterminer le nombre maximal des niveaux dans la tour seulement en examinant le programme avant son évaluation. Dans ce cas, le nombre de niveaux peut dépendre des entrées du programme.

[2] Nous nommerons quelques fois ceux-ci langages à *la 3-Lisp*.

termes dans les programmes ne changent pas la sémantique de ces derniers. Cette technique est utile au programmeur pour mieux comprendre le comportement de ses programmes. Elle est aussi utile au concepteur de langage pour l'aider à optimiser les programmes avant la compilation.

Le problème de la classification, que cela soit méthodologique ou théorique, est une tâche difficile et de grande envergure. Pour ce faire, des outils pratiques et théoriques doivent être utilisés. Ce doctorat contribuera à ce projet de classification. Nous utiliserons un concept aujourd'hui reconnu en langage de programmation qui se nomme **monade**. Le but sera alors d'établir une classification précise de la puissance d'expression de la réflexion dans les langages réflexifs et des méthodes de raisonnement sémantique possible pour chaque classe. Les langages réflexifs considérés seront principalement ceux dont une implantation raisonnable efficace est, en principe, réalisable.

Les monades sont des constructions algébriques venant de la théorie des catégories. A la fin des années 80, elles ont été reconnues utiles par Moggi et Spivey pour représenter de façon uniforme certains comportements computationnels (effets de bord, continuations, etc.) en sémantique dénotationnelle. À partir de ce moment, plusieurs applications ont été développées et elles ont eu un impact non-négligable sur la structuration des programmes et l'extension de la puissance des langages. Il a été reconnu que les monades pouvaient jouer un rôle important dans le développement d'une théorie sémantique pour la réflexion.

Ce document se divise en sept sections. La section 2 présente la terminologie de la réflexion et les tendances de recherche d'hier et d'aujourd'hui. La section 3 présente la programmation monadique. Dans la section 4, il sera question de la problématique de notre projet de doctorat. En partant d'un article de Mendhekar & Friedman [MF96], nous élaborerons des stratégies de recherche pour décrire la réflexion dans un cadre monadique et, plus généralement, dans le langage de la théorie des catégories. Nous poursuivrons par la section 5 dans laquelle nous introduirons la théorie des catégories et son importante implication en sémantique des langages de programmation. Enfin, une mise en perspective (section 6) et une conclusion (section 7) suivront.

2 Réflexion

> Semantics can be a tool with which to judge systems, not merely a method of describing them. [Smi84]
>
> *Brian C. Smith*

Dans cette section, nous expliquons brièvement la réflexion telle que pensée par Brian Smith et par d'autres depuis plus de 15 ans. Nous verrons entre autres que nous distinguons deux classes de langages réflexifs : les langages *à la 3-Lisp* et les langages extensibles. Le première classe de réflexion a l'effet de 'grimper' dynamiquement dans la tour réflexive pour insérer du code réflexif et la seconde a principalement l'effet de modifier l'interprète d'évaluation statiquement[3]. Nous nous intéresserons particulièrement aux langages extensibles.

2.1 Langages *à la 3-Lisp*

Un système est dit **réflexif** si celui-ci incorpore des structures le représentant et permet l'accès à ces structures en lecture, certainement, et en écriture, le plus possible. Il y a une connexion causale entre le système et la représentation de lui-même. Cela signifie que le système a toujours une représentation exacte de lui-même quels que soient les changements dans le système ou dans sa représentation.

Un langage de programmation possède une architecture réflexive s'il fournit des outils qui permettent au programmeur de conceptualiser les programmes dans ce langage comme des systèmes pouvant être réflexifs. Nous parlons ici de conceptualisation car la réflexion doit être vue avant tout comme une méthodologie de programmation facilitant ou, en tout cas, modifiant le travail du programmeur.

Deux types de réflexion ont été identifiés. Le premier est dit de **structure**. Il permet une représentation des structures de données du langage et, au mieux, une représentation du programme faisant de celui-ci une entité de plein droit. Le deuxième type est une **réflexion de comportement**. Elle consiste en une représentation manipulable plus ou moins détaillée (dépendant de la réflexion de structure disponible) de la sémantique des

[3]Précisons ici qu'il est possible que certaines modifications se fassent dynamiquement. Par contre, la plupart se feront statiquement.

programmes du langage.

Conceptuellement, la réflexion de comportement implique la capacité d'un programme 'd'observer' et de modifier des structures de données représentant, à différents degrés de détails, l'état de calcul d'un interprète abstrait (hypothétique ou non dépendant des choix d'implantation : interprétation, compilation, etc.) qui remplace, dans la conception du programmeur, l'interprète de base du langage. L'implémenteur du langage réflexif doit imposer au programmeur de façon syntaxique ou sémantique (explicitement dans la syntaxe ou implicitement dans l'évaluation des structures de contrôle réflexives) des moyens d'éviter le problème du **recouvrement introspectif**. En tenant compte de la connexion causale, le recouvrement introspectif est un phénomème d'inconsistance qui se produit lorsque des mises à jour de l'état de calcul (ou de l'une de ses représentations) faites par du code réflexif viennent en conflit avec des mises à jour du même état de calcul faites par l'interprète abstrait ou plus précisément faites par l'interprète de base du langage.

Brian Smith a trouvé une solution au recouvrement introspectif en imaginant le modèle de la **tour réflexive** qui est, rappelons-le, un empilement de méta-niveaux tous en connexion causale les uns avec les autres, où chaque méta-niveau est vu comme un interprète évaluant les niveaux inférieurs. La réflexion s'active par des **procédures réflexives**. Pour éviter le conflit du recouvrement introspectif, ces procédures sont insérées en cours d'exécution dans un des méta-niveaux de la tour réflexive. Ces procédures ont accès aux mêmes structures de données de l'état de calcul que l'interprète dans lequel elles ont été insérées. De plus, ce type de réflexion est dit **ponctuelle** [MJD96], car les changements de comportement ne sont que temporaires c'est-à-dire que durant l'évaluation des procédures réflexives.

Après sa thèse, Brian Smith s'est intéressé aux problèmes d'implantation de son langage 3-Lisp. Avec l'aide de des Rivières [dRS84], il a développé une implantation compilée d'une tour réflexive avec création paresseuse et destruction dynamique des niveaux.

Wand & Friedman ont été les premiers à vouloir formaliser les langages réflexifs *à la 3-Lisp*. Ils ont voulu donner la sémantique dénotationnelle de la tour réflexive évitant les définitions méta-circulaires des premières définitions de 3-Lisp (en effet, 3-Lisp a été défini en 3-Lisp et donc n'est pas bien fondé). Pour ce faire, ils ont suivi la contrainte d'un seul flux de calcul actif à tout moment de l'évaluation de la tour réflexive (cela est aussi vrai en 3-Lisp).

De là est née la notion de **méta-continuation**, une sorte de pile d'états de calcul qui mime la tour réflexive non-bornée. Seul l'état de calcul au dessus de la pile est actif à tout moment (contrairement à la tour réflexive où tous les niveaux sont actifs en même temps). Pour illustrer leur modèle, ils ont développé un langage fonctionnel réflexif appelé Brown [WF88]. Leurs travaux sont basés sur une décomposition de la réflexion en deux actions distinctes : **réification** et **réflexion**. La réification est l'action de passer les structures réifiées de l'interprète d'évaluation au niveau du programme. De son côté, la réflexion est le processus qui prend, du programme, des structures représentant un état de calcul et installe ces structures au niveau de l'interprète. Cette terminologie est aujourd'hui grandement utilisée par les chercheurs en réflexion. En tout cas, Brown est une implantation de la tour réflexive beaucoup moins "juste" que celle de 3-Lisp car un simple flux de calcul est imposé au modèle, empêchant les effets de bord (qui constituent une part importante des applications de la réflexion). Ils se sont éloignés du modèle théorique pour obtenir un modèle de calcul unifilaire formalisable par la sémantique dénotationnelle. En somme, comme ils l'ont mentionné eux-mêmes, ils ont décrit une spécification *non-réflexive* (ou non-circulaire) de la tour réflexive.

Danvy & Malmkjær ont approfondi les travaux de Wand & Firedman et ils ont développé leur langage fonctionnel réflexif Blond [DM88]. Ils arrivent tant bien que mal à présenter une sémantique dénotationnelle moins puissante que leur langage Blond mais plus approfondie que celle dans les articles de Wand & Friedman. La particularité de leur travail se trouve au niveau de la formalisation de la relation des domaines sémantiques de chaque niveau de la tour. De plus, ils généralisent le concept de méta-continuation.

Par la suite, certains chercheurs du domaine ont migré vers d'autres aspects des langages. Par exemple, Danvy & Filinski se sont intéressés à des problèmes de contrôle et de contexte d'évaluation en développant l'idée de la méta-continuation hors du monde de la réflexion [DF90, DF92]. Les recherches sur les langages *à la 3-Lisp* se sont vues délaissées [DM95] sauf pour quelques unes [JF92] [AMY93] [Dem94]. Cela s'explique en partie par le fait que presque aucune application de cette réflexion n'a été montrée.

Parmi ceux qui ont continué à s'y intéresser, Jefferson & Friedman [JF92] ont présenté un langage *à la 3-Lisp* appelé $\mathcal{I}_{\mathcal{R}}$, plus simple que les autres et se voulant adéquat pour l'expérimentation. Le modèle de ce langage est une tour réflexive finie et *bornée*, contrairement à 3-Lisp et les autres. Il est remarquable de voir que $\mathcal{I}_{\mathcal{R}}$ est une des premières implantations importantes

d'une tour réflexive *bornée* qui est maintenant un modèle plus populaire. Jefferson & Friedman ont donc été des précurseurs des résultats actuels en réflexion.

En somme, au niveau théorique, l'échec de la formalisation des langages *à la 3-Lisp* est dû au fait qu'aucun des modèles dénotationnels proposés n'a respecté l'hypothèse de compositionnalité sans restriction forte sur les langages étudiés [DM95] [MDC96]. La violation de cette hypothèse rend l'induction structurelle impossible.

Ces insertions dynamiques dans les méta-niveaux de la tour par des opérateurs qui ont accès à l'état de calcul constitue une façon de modifier le code de l'interprète d'évaluation du programme (le comportement du programme). Cette approche est, nous l'avons vu, la conception originelle de la réflexion où il y a présence évidente d'une tour réflexive. Par ailleurs, ce n'est pas la seule façon. Il est possible d'ajouter une nouvelle contrainte à la réflexion comportementale en encourageant les changements de comportement réalisés *avant* l'exécution et non pas pendant. Dans ce cas, nous avons affaire à des modifications *permanentes* sur la définition du langage. Ce type de réflexion est dit **continue** [MJD96] (en opposition à la réflexion **ponctuelle** des langages *à la 3-Lisp*).

Dans la section suivante, nous discuterons des langages extensibles, sortes de langages réflexifs qui regroupent en bonne partie les langages à réflexion structurale et à réflexion comportementale.

2.2 Langages extensibles

Les langages extensibles d'autrefois [SY74] et d'aujourd'hui [LHJ95] sont, par définition, des langages pour lesquels il est possible de modifier la sémantique par ajout ou retrait de blocs syntaxiques qui représentent certaines fonctionnalités utiles aux programmeurs. Evidemment, le but des langages extensibles est de faciliter la redéfinition du langage en évitant les modifications manuelles et fastidieuses dans le code même de l'interprète ou du compilateur du langage.

Généralement, il est plus difficile de reconnaître la présence d'une tour réflexive[4] dans les langages extensibles que dans les langages *à la 3-Lisp*.

[4] Il est question ici d'une tour réflexive arbitrairement grande car, dans les langages extensibles, il y a toujours au moins deux niveaux : le niveau du programme et le niveau de l'interprète.

Parfois même, aucune tour réflexive n'est distinguable. Dans ce cas, nous avons affaire à un langage extensible avec réflexion de comportement faible ou avec réflexion de structure seulement. Les chercheurs qui ont travaillé sur ces langages s'intéressent fondamentalement à la relation entre niveau de base (le programme) et méta-niveau (l'interprète) plutôt qu'à la *réflexion* à proprement dit. Cela permet de comprendre pourquoi ces chercheurs ne disent pas faire de la recherche sur la réflexion et souvent, ne connaissent pas la terminologie et les questions de la réflexion.

Dans un article de Malenfant, Jacques & Demers [MJD96], la notion de réflexion comportementale **statique** a été mise en évidence (en opposition à la réflexion **dynamique**). La réflexion comportementale statique est une forme restreinte de réflexion où une quantité "suffisante" d'information est connue statiquement (avant même l'exécution du programme) pour permettre la compilation efficace du langage. Il est clair que les langages extensibles sont aptes à offrir une réflexion statique beaucoup plus que les langages *à la 3-Lisp*. La réflexion ponctuelle faite surtout par les langages *à la 3-Lisp* est souvent dépendante de choix dynamiques empêchant d'en extraire efficacement l'information statique pour la compilation.

Remarquons que ces termes ne s'utilisent pas dans les mêmes contextes. La terminologie de langage extensible vient du domaine de la méthodologie de programmation, tandis que la terminologie de langage avec réflexion statique se retrouve dans le contexte de l'implantation et la compilation de langages réflexifs [MJD96]. Nous nous intéresserons dans notre projet essentiellement à la méthodologie. Les aspects d'implantation de la réflexion seront approfondis par Marco Jacques [Jac96].

Les langages extensibles découlent souvent de travaux formels sur la sémantique. Il existe une bonne quantité de résultats théoriques sur les langages extensibles. Nous retrouvons entre autres la sémantique des actions de Mosses [Mos92], la sémantique directe extensible [CF94], les continuations composables [DF90], la sémantique des moniteurs [KHC91] et la sémantique monadique [Mog89b]. Nous discuterons plus en détails de ces sémantiques dans les sections 4 et 6.

Concernant la sémantique monadique, Mendhekar & Friedman [MF96] ont montré qu'il est possible de lier cette sémantique et la réflexion pour former ce qu'ils ont appelé la **réflexion monadique**. Nous ne rentrerons pas dans les détails maintenant. Pour l'instant, mentionnons que les auteurs montrent comment les monades peuvent servir à organiser le méta-niveau.

Ce genre d'organisation permet d'obtenir des propriétés souhaitables pour le raisonnement équationnel (raisonnement fait à l'aide de règles de réécriture de termes). Par contre, elle restreint fortement le pouvoir de la réflexion de comportement, car elle exclut toute présence de tour réflexive (arbitrairement grande) et elle se spécialise sur la relation entre niveau de base et méta-niveau. Cela rend ces travaux intimement liés à ceux sur les langages extensibles.

Somme toute, les langages *à la 3-Lisp* sont construits toujours à partir d'un modèle d'une tour réflexive (souvent non-bornée). Cela les rend a priori complexes et difficiles à simplifier, car il est ardu de les redéfinir sans être méta-circulaire (hors du modèle de la tour réflexive), tandis que les langages extensibles peuvent plus naturellement se simplifier jusqu'à ne laisser qu'une réflexion comportementale simple ou même qu'une réflexion de structure. Il est possible d'étudier des modèles simplifiés jusqu'à graduellement se rendre au modèle principal. Ce n'est pas le cas de la plupart des langages *à la 3-Lisp* dans lesquels leur simplification peut être drastique et n'offrir aucune graduation intéressante du plus simple au plus complexe. Dans le cas des langages extensibles, la présence d'une tour réflexive (s'il y en a) n'est pas toujours évidente dans ce cas et elle s'exprime plus facilement par un système de réécriture *indépendant* de l'interprète (ce qui n'est pas le cas des langages *à la 3-Lisp*). Enfin, nous verrons plus en détails que ces langages offrent un meilleur encadrement pour le raisonnement équationnel, contrairement aux langages *à la 3-Lisp*.

Dans la section suivante, nous nous intéresserons à un modèle réflexif appartenant à un langage extensible comportemental.

2.3 Système de réécriture et tour réflexive

Il a été reconnu que les logiques de réécriture et les systèmes de réécriture sont appropriés pour la formalisation logique et sémantique de la réflexion. Entre autres, Mendhekar & Friedman [MF93] ont développé une logique de programmation basée sur le λ-calcul typé pour les langages 3-Lisp, Brown et $\mathcal{I}_{\mathcal{R}}$. Ils ont utilisé un système de réécriture pour définir la sémantique du langage.

De plus, les travaux de Clavel & Meseguer sur la théorie générale de la réflexion ont été inspirés du développement d'un langage logique de réécriture [CM96]. Cela fait suite, entre autres, à un article de Goguen &

Meseguer [GM87] dans lequel ils unifient différents paradigmes de programmation à l'aide d'une base logique. Les travaux de Watanabe [Wat95] ont aussi mis en évidence encore d'autres résultats intéressants montrant que la réflexion et les systèmes de réécriture font bon ménage.

Nous allons, dans cette section, expliquer brièvement le modèle réflexif basé sur le système de réécriture de Malenfant, Dony & Cointe [MDC92, MDC96]. À l'aide de ce modèle, nous pourrons plus tard illustrer un exemple avantageux de l'utilisation des monades pour la réflexion.

Le modèle de Malenfant et al. fait suite aux travaux sur la réflexion en programmation par objets et la modélisation de la réflexion de structure par les classes et les méta-classes. Il fait partie d'une des grandes familles de réflexion comportementale dans les langages à objets : le protocole introspectif $rechercher \circ appliquer$. Nous n'avons pas discuté de la réflexion dans le paradigme de la programmation par objets d'abord parce que nous n'avons pas l'espace de revoir ce vaste monde et aussi parce que nous ne croyons pas cela nécessaire. Nous nous intéressons seulement au mécanisme réflexif qui peut être expliqué hors d'un paradigme particulier. Nous expliquerons quand même brièvement le protocole de réflexion dans son contexte.

Ils se sont intéressés plus précisément à la réflexion de comportement en développant un modèle à base de prototypes en utilisant des méta-objets. L'idée est de représenter les propriétés comportementales des objets dans des méta-objets qui leur sont associés. Le passage de messages est précisément le mécanisme fondamental de toute programmation par objets. Il est naturel de réifier ce mécanisme pour chaque objet. Les méta-objets décrivent donc les façons dont les objets agissent quand ils reçoivent un message.

La réification de ce mécanisme est réalisée en rendant visible les deux opérations principales de l'évaluateur durant un envoi de messages : la **recherche d'une méthode** et l'**application de la méthode trouvée**. Ces deux opérations sont implémentées dans le modèle par deux méthodes appelées *rechercher* et *appliquer*. Ces opérations sont réifiées en deux objets de base BL et BA respectivement. Le programmeur a donc le moyen d'écrire ses propres objets ayant pour sélecteur *rechercher* ou *appliquer*. La redéfinition de ces opérations constitue le mécanisme de réflexion du langage.

Dans les détails, supposons un message $o.s(a_1, \ldots, a_n)$ où o représente l'objet receveur du message, s, le sélecteur du message et a_1 à a_n, les arguments de l'envoi du message. Pour chaque envoi de message, un mécanisme d'introspection s'active et décompose ce message en trois phases : **(1)** trou-

ver le méta-objet du receveur o, **(2)** envoyer un message *rechercher*, que nous écrirons par le symbole $\gamma(s, o)$, au méta-objet trouvé par (1) avec pour paramètres le sélecteur et l'objet, et **(3)** envoyer le message *appliquer* à la méthode retournée par (2) qu'on écrira $\alpha(o, [a_1, \ldots, a_n])$. Nous ne faisons aucune hypothèse sur “l'ordre” d'évaluation des trois phases (l'ordre paresseux ou strict est possible comme dans les langages fonctionnels). Mises ensemble, ces trois phases se lisent sous la forme d'une règle de réécriture :

$$o.s(a_1, \ldots, a_n) \Rightarrow \mu(o).\gamma(s, o).\alpha(o, [a_1, \ldots, a_n])$$

Nous appelons cette règle l'**introspection réflexive** [MDC92]. Cette règle fait partie d'un ensemble de règles appelé les **hypothèses du modèle**. Ces règles constituent les fondements mêmes du mécanisme de la réflexion dans ce langage. Les autres hypothèses du modèle sont

$$\begin{aligned} \mu(o) &= mo && (1) \\ BMO.\gamma(\gamma, BMO) &= BL && (2) \\ BMO.\gamma(\alpha, BL) &= BA && (3) \\ \mu(BMO) &= BMO && (4) \\ \pi(ROOT) &= ROOT && (5) \end{aligned}$$

L'équation (1) nous dit que chaque objet o a un méta-objet unique mo[5]. L'équation (2) nous dit qu'il existe un objet unique BL qui contient une méthode *rechercher* qu'on dit de base. De plus, cet objet a pour méta-objet BMO. Dans le cas de l'équation (3), il existe un objet unique BA qui contient une méthode *appliquer* qu'on dit de base. Cet objet a pour méta-objet BMO. L'équation (4) dit que le méta-objet BMO s'admet lui-même comme son propre méta-objet. Enfin, la dernière équation dit qu'il existe un objet $ROOT$ qui est son propre parent (π est la méthode qui retourne le parent d'un objet). Cet objet $ROOT$ a pour méta-objet BMO.

Nous remarquons que ces règles sont nécessaires (mais non suffisantes) pour engendrer une réécriture *finie*. En effet, toutes ces règles ont pour objectif d'arrêter la réécriture expansive que génère la règle d'introspection réflexive.

Dans ce modèle, il existe des tours réflexives. Pour les voir, il suffit de remarquer que les méthodes *appliquer* des différents méta-objets du modèle

[5] L'équation nous dit aussi que ce méta-objet doit être trouvé d'une manière indépendante du système de réécriture.

peuvent être vues comme des interprètes d'évaluation des méthodes. Des tours réflexives apparaissent, car les méthodes *appliquer* sont des méthodes qui doivent avoir leur propre méthode *appliquer* et ainsi de suite. Cela crée justement un empilement de méthodes *appliquer*. Ces tours sont de longueur arbitrairement grandes et en nombre fini de niveaux, mais bornées à l'avance[6] dans les programmes comme pour le langage $\mathcal{I}_{\mathcal{R}}$. Ce sont justement les hypothèses du modèle qui nous assurent de la finitude de la tour. Il est possible de voir une similitude entre l'opération de *réification* des langages *à la 3-Lisp* avec la règle d'introspection réflexive et l'opération de *réflexion* avec la règle (3). Nous sommes bien devant un modèle de réflexion comparable en puissance à la réflexion *à la 3-Lisp*. Il n'est donc pas question ici de simple réflexion structurale mais de réflexion de comportement de puissance significative.

De surcroît, nous sommes devant une sorte de langage extensible, car le code réflexif n'est pas inséré toujours dynamiquement dans les 'méta-niveaux' de la tour (comme cela est réalisé en 3-Lisp). Il est 'installé' principalement avant l'exécution d'un message par 'redéfinition' des méthodes *rechercher* et *appliquer*. Bien sûr, dans ce modèle, il est toujours possible d'effectuer dynamiquement les changements comme en 3-Lisp, mais cela est plutôt rare. Dans un programme, plus il y a de ce type de changements dynamiques, plus la compilation de ce programme s'en ressent [MJD96, Jac96]. Il est donc préférable de restreindre le plus possible les changements dynamiques.

Evidemment, pour obtenir un système de réécriture opérationnel pour l'évaluation d'envois de messages, il faut ajouter d'autres règles de réécriture de moindre importance et déterminer un ordre d'application des règles (pour les machines séquentielles). Nous n'expliquerons pas la partie opérationnelle du modèle. Nous ne présentons que les règles les plus importantes, car elles nous permettront de voir dans la section suivante comment elles peuvent être facilement agencées aux monades. Pour ceux qui aimeraient des détails d'implantation, ils peuvent consulter l'article de Malenfant et al. [MDC96] dans lequel ils présentent un système de réécriture à priorité.

Enfin, la modélisation de la réflexion par un système de réécriture permet de séparer le processus de réflexion du processus d'évaluation du langage rendant la compréhension et la maintenance du langage plus faciles [MDC96]. Par ailleurs, Mendhekar & Friedman [MF96] ont montré que le

[6]Il est possible de déterminer le nombre maximal de niveaux d'une tour seulement en examinant le programme avant son exécution.

phénomène de **traitement distinctif** ("separation of concerns") est important pour le raisonnement équationnel sur les programmes. Cette caractéristique des systèmes réflexifs consiste en la possibilité de séparer le raisonnement des parties non-réflexives d'un programme du raisonnement des parties réflexives. La modélisation de la réflexion par un système de réécriture aide possiblement au respect du traitement distinctif. Par contre, cela reste à être approfondi. Nous approfondirons cette question.

3 Programmation monadique

> Monads are everywhere. [Wad94][7]
>
> *Hypothèse de Moggi*

En 1989, Moggi [Mog89b, Mog91] et indépendamment Spivey [Spi89, Spi90] ont introduit les monades en conception de langages de programmation pour structurer leur sémantique dénotationnelle. Plusieurs aspects sémantiques des langages peuvent être vus en tant que monades incluant les états, les exceptions, les continuations, le non-déterminisme et même l'interaction. En 1990, Wadler [Wad90] a montré que les monades peuvent être utiles de deux façons : **horizontalement**[8], comme une technique de structuration des programmes d'un langage et **verticalement**, pour étendre la puissance d'un langage. Ces deux aspects des monades vont nous servir à la classification de la réflexion.

Les monades viennent de la théorie des catégories mais elles ne demandent aucune notion de celle-ci pour les comprendre et les utiliser en programmation [JD93]. Pour ce faire, il suffit de voir la monade comme une structure algébrique ou un type abstrait qui s'exprime dans un langage fonctionnel typé. Nous croyons que la compréhension des monades hors de leur contexte historique (la théorie des catégories) est important pour le développement d'une méthodologie de programmation facile à expliquer à tout programmeur. Par contre, sorti de son contexte, le concept de monade est amputé de toute sa richesse et de son importance théorique. Pour certains, il peut même paraître simpliste et insignifiant. Pour une bonne introduction à la programmation monadique, il faut consulter les articles de Wadler [Wad92] et de Hutton & Meijer [HM96]. Pour une introduction aux

[7] Cette conjecture dit que toute structure de contrôle peut être modélisée par une monade.

[8] Traduction libre de "internally", venant d'un article de Wadler [Wad95].

monades dans leur contexte théorique, le lecteur peut consulter l'article de Hill & Clark [HC94] ou encore le cours de Moggi [Mog89a].

Nous allons illustrer la programmation monadique par la présentation de la sémantique monadique d'un langage fonctionnel minimal applicatif dont la syntaxe est définie par l'équation BNF suivante[9] : $e ::= v \mid \lambda v \rightarrow e \mid (e_1 e_2)$. Nous nous sommes inspiré des travaux de Liang & Hudak [LHJ95, LH96].

En sémantique dénotationnelle, la sémantique est une fonction qui associe un terme abstrait et quelques données représentant l'état de calcul (ou d'autres éléments comme l'environnement, la continuation, etc.) à une réponse. En sémantique monadique, la fonction sémantique est plus générale et associe les termes abstraits à des **calculs**, c'est-à-dire à une structure de données dont la forme n'est pas spécifiée à priori. Les détails tels que l'environnement et la continuation sont cachés et deviennent "visibles" après l'évaluation, quand les *calculs* deviennent des **valeurs abstraites**, c'est-à-dire des valeurs terminales (statiques) accompagnées de structures de données contenant les informations de comportement. Plus spécifiquement, la fonction sémantique monadique a les types suivants : $\mathcal{E}$: **Terme** $\rightarrow M$ **Valeur** où **Valeur** est le domaine des réponses. Le constructeur de type M est appelé une **monade**. Cette monade a l'effet de cacher les détails des valeurs à calculer montrant seulement le type résultant. M est aussi équipée de deux fonctions de base :

$$\begin{array}{rcl} \textbf{then} & : & Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb \\ \textbf{return} & : & a \rightarrow Ma \end{array}$$

Nous pouvons imaginer la monade M comme un type abstrait contenant deux opérations de manipulation de structures de données (généré par M). D'ailleurs, Hughes [Hug95] utilise les monades de ce point de vue.

La fonction **then** se comprend mieux sous la forme "infixe". Intuitivement, "c_1 **then** $\lambda v \rightarrow c_2$" correspond à l'évaluation qui, d'abord, évalue c_1, lie le résultat à v et évalue ensuite ("then") c_2. Le terme " **return** v" est une évaluation triviale, c'est-à-dire que la fonction **return** ne fait que retourner la valeur triviale (non-décomposable) qu'on nomme un *calcul* (en s'opposant aux *valeurs abstraites* qui sont des valeurs décomposables). En d'autres mots, en programmation, **then** s'occupe de rendre explicite

[9] La notation $\lambda_ \rightarrow _$ sera toujours utilisée pour désigner l'abstraction fonctionnelle.

l'ordre de l'évaluation des expressions composées et **return** s'occupe des expressions atomiques.

Comme dans toute sémantique, les abstractions fonctionnelles et les applications ont besoin d'un accès minimal à un environnement **Env** qui associe les variables à des *calculs*[10]. Nous nous donnons donc une fonction qui retourne à la volée l'environnement courant : **readEnv** : M **Env**. Voici maintenant la sémantique de notre mini-langage fonctionnel :

$$\begin{aligned}
\mathcal{E}[\![v]\!] &= \textbf{readEnv then} \\
&\qquad \lambda\rho \to \rho\, v \\
\mathcal{E}[\![\lambda v \to e]\!] &= \textbf{readEnv then} \\
&\qquad \lambda\rho \to \textbf{ return}\,(\lambda c \to (\rho\,[c/v])\,\mathcal{E}[\![e]\!]) \\
\mathcal{E}[\![(e_1\, e_2)]\!] &= \mathcal{E}[\![e_1]\!]\ \textbf{then} \\
&\qquad \lambda f \to \mathcal{E}[\![e_2]\!]\ \textbf{then} \\
&\qquad\qquad \lambda v \to f(\textbf{ return}\, v)
\end{aligned}$$

Cette sémantique ne rend compte que de l'ordre de certaines opérations sémantiques sans expliciter les opérations. En effet, dans le cas de l'application, l'évaluation de $(e_1\, e_2)$ consiste en l'évaluation e_1, ensuite, de e_2 et de l'application de la fonction trouvée sur le résultat de l'évaluation de e_2 (qui est une valeur triviale expliquant la présence de la fonction **return**).

La sémantique n'est pas complète puisque le constructeur M et les fonctions **then** et **return** ne sont pas encore définis. Nous n'avons en main qu'un *squelette* de la monade (les types de M, **return** et **then**). Si nous définissons la monade et nous l'ajoutons à la sémantique monadique, nous obtenons une sémantique *complète*. Il y a plusieurs définitions monadiques possibles. Cela dépend des fonctionnalités sémantiques que nous désirons. Par exemple, si nous voulons une sémantique simple en style direct sans aucun fonctionnalité particulière, nous allons fournir la monade appelée **identité** M_I, c'est-à-dire qui ne différencie pas un *calcul* d'une *valeur abstraite*. C'est la monade la plus simple qui existe. Notons aussi que le langage pour définir les monades n'est pas nécessairement le même langage que celui qui est défini par la sémantique monadique. Ce langage est plutôt

[10] Nous supposons pour des raisons de simplicité que toute variable (appelée environnement) de ce type accepte en paramètre tout terme du langage. Si le terme est un identificateur, le résultat de l'application retournera la valeur associée à celui-ci. Sinon, l'environnement se comportera comme une fonction identité.

un méta-langage fonctionnel d'ordre supérieur. Dans ce cas-ci, nous supposons que le méta-langage est un langage fonctionnel applicatif avec structure **Let_ = _in _.**

$$\begin{array}{rcl} \textbf{Type}\ M_I\ a & = & a \\ \textbf{return}_I\ x & = & x \\ c\ \textbf{then}_I\ f & = & f\ c \end{array}$$

Nous pouvons facilement démontrer que cette monade rend la sémantique monadique (après quelques β-réductions) pareille à la sémantique en style direct de notre mini-langage.

Dans la section suivante, nous discuterons de la monade computationnelle et, par la suite, de deux monades qui nous permettent d'obtenir les sémantiques style par état et style par passage à la continuation.

3.1 Monades computationnelles

Les monades computationnelles ont été présentées pour la première fois par Wadler [Wad90] pour montrer comment la notation des listes par compréhension de la programmation fonctionnelle peut être généralisée à une monade arbitraire. Dans son premier article sur le sujet, il généralise le constructeur de listes $*$[11] à la monade où M est un constructeur de type quelconque. Il généralise aussi les fonctions bien connues $map : (\alpha \rightarrow \beta) \rightarrow (\alpha* \rightarrow \beta*)$ qui applique une fonction sur chaque élément d'une liste et $join : (\alpha*)* \rightarrow \alpha*$ qui concatène toutes les listes d'une liste de listes. Voici la première définition de la monade qui est aujourd'hui délaissée, mais qui est plus facile à comprendre que celle qui sera présentée en section 3.2.

3.1 Définition.
*La monade computationnelle M est un quadruplet $(M, map, unit, join)$ où $map : (\alpha \rightarrow \beta) \rightarrow (M\alpha \rightarrow M\beta)$, $unit : \alpha \rightarrow M\alpha$ appelé l'***unité***, et $join : M(M\alpha) \rightarrow M\alpha$, appelé le* **multiplicateur***, obéissent aux lois algébriques suivantes :*

[11] Nous utiliserons toujours cette notion pour désigner le type liste pour un type donné. Par exemple, le type $\alpha*$ correspond au type des listes dont les éléments sont de type α.

$$\begin{array}{rcll} map\,\mathtt{Id} & = & \mathtt{Id} & (1) \\ map\,f \circ map\,g & = & map\,(f \circ g) & (2) \\ unit \circ f & = & map\,f \circ unit & (3) \\ join \circ map\,(map\,f) & = & map\,f \circ join & (4) \\ join \circ unit & = & \mathtt{Id}_{M\alpha \to M\alpha} & (5) \\ join \circ map\,unit & = & \mathtt{Id}_{M\alpha \to M\alpha} & (6) \\ join \circ map\,join & = & join \circ join & (7) \end{array}$$

Les quatres premières équations sont moins importantes et quelquefois, elles peuvent être mises de côté[12]*. Les trois dernières sont les plus importantes et elles sont appelées les lois* **unité gauche**, **unité droite** *et* **associativité** *respectivement.* □

Si $M = *$ (le constructeur de liste $*$), nous obtenons les lois des fonctions de liste bien connues *map*, *unit* et *join*. Il est facile de se convraincre que les équations précédentes sont vraies pour $M = *$. Dans ce cas, ces équations sont exactement celle de la théorie des liste de Bird [Bir87]. Nous nommerons la monade liste $M_l = (*, map_l, unit_l, join_l)$ où $M = *$, le constructeur de liste.

Deux ans plus tard, Wadler [Wad92] adopte la définition plus appropriée suivante. Celle-ci est plus courte, mais beaucoup moins conforme à la notion de monade catégorique. En fait, elle se rapproche plus de la notion de *triplet de Kleisli* (voir l'annexe C pour les détails catégoriques).

3.2 Définition.
Une **monade computationnelle** *est un triplet* (M, **return**, **then**) *où* **then** : $M\alpha \to (\alpha \to M\beta) \to M\beta$ *et* **return** : $\alpha \to M\alpha$ *doivent respecter les équations suivantes :*

$$\begin{array}{rcl} \mathbf{return}\,a\ \ \mathbf{then}\,(\lambda b \to n) & = & n[a/b] \\ m\ \ \mathbf{then}\,(\lambda b \to\ \mathbf{return}\,b) & = & m \\ m\ \ \mathbf{then} & & \\ ((\lambda a \to n)\ \ \mathbf{then}\,(\lambda b \to m)) & = & (m\ \ \mathbf{then}\,(\lambda a \to n)) \\ & & \quad \mathbf{then}\,(\lambda b \to m) \end{array}$$

[12]Les deux premières servent à ce que M et *map* forment ensemble un foncteur (voir annexe B). La troisième et la quatrième nous assurent que l'unité (*unit*) et le multiplicateur (*join*) sont des transformations naturelles (voir annexe B).

Ces équations ont les mêmes noms que ceux donnés aux équations de la définition 3.1 et dans le même ordre. La loi **unité gauche** *dit que, de calculer la valeur a, lier la réponse à b et calculer n est égale au résultat n avec la valeur a substituée à la variable b. La loi* **unité droite** *dit que le calcul de m, la liaison du résultat à la variable b et le retour de b est égale à m. La loi* **associativité** *dit que l'ordre des parenthèses dans la composition des calculs ne change rien.* □

Nous voyons qu'une monade est une notion simple (un constructeur et seulement 3 équations !) et très générale (M n'a pas de sémantique pour l'instant). Cette simplicité et cette généralité facilitent sa correspondance à un grand nombre de structures sémantiques de programmation. Par exemple, voici deux définitions de monade, l'une nous permet d'avoir le style par état et l'autre, le style passage à la continuation.

$$\begin{aligned} \textbf{Type}\ M_e\, a &= e \rightarrow (a, e) \\ \textbf{return}_e\, a &= \lambda x \rightarrow (a, x) \\ c\ \textbf{then}_e\, f &= \lambda s_0 \rightarrow \textbf{Let}\ (a, s_1) = c\, s_0\ \textbf{in}\ f\ a\ s_1 \end{aligned}$$

Expliquons quelque peu cette définition de la monade "état" écrite dans notre méta-langage. Le constructeur de type M_e est un type d'ordre supérieur qui prend en paramètre un type "état" (représenté par la variable e). Donc, $(M_e E)a = \lambda x \rightarrow (a, x)$ où a est un *calcul* et "attend" un état e (l'abstraction x) de type E. Nommons le tout, une **valeur abstraite**.

Nous savons déjà que **return** est une fonction qui prend un calcul (α) et retourne une *valeur abstraite* $M\alpha$ alors $\textbf{return}_e$ prend un *calcul* (calcul "sans effet") et retourne un couple contenant le *calcul* abstrait de l'état courant (en attente d'un état). Dans le cas de la fonction $\textbf{then}_e$, $c : M\alpha$ est une *valeur abstraite*. f est une fonction qui prend un *calcul* et retourne une *valeur abstraite*. On applique la *valeur abstraite* c sur l'état courant s_0 pour récupérer, ensuite, la nouvelle *valeur abstraite* (a, s_1). On applique enfin f sur le *calcul* a et on y ajoute le nouvel état courant s_1.

$$\begin{aligned} \textbf{Type}\ M_\kappa\, r &= (a \rightarrow r) \rightarrow r \\ \textbf{return}_\kappa\, a &= \lambda k \rightarrow k\ a \\ c\ \textbf{then}_\kappa\, f &= \lambda k \rightarrow c\, (\lambda a \rightarrow f\ a\ k) \end{aligned}$$

Concernant la monade "continuation" M_κ, la structure est plus complexe. Nous omettons son explication détaillée. Nous la présentons seulement pour démontrer que le style monadique est effectivement plus général que le style passage à la continuation car on obtient le deuxième à partir du premier, mais l'inverse n'est pas possible (sans ajouter de nouvelles fonctions de contrôle comme `call/cc` de Scheme).

3.2 Retour à l'exemple

Dans cette section, nous montrerons comment il serait possible d'incorporer les monades dans le système de réécriture de Malenfant et al. [MDC92, MDC96] dont nous avons montré les règles principales en section 2.3.

Cet amalgame a un but intéressant. En bout de ligne, nous aimerions obtenir une implantation générique du système de réécriture à priorité développé par Malenfant et al. [MDC96] de façon à pouvoir intégrer facilement certaines fonctionnalités utiles comme la manipulation d'un état global, des mécanismes de non-déterminisme, la manipulation de la continuation du calcul, etc. Malenfant et al. [MDC96] ont déjà fait l'expérience d'intégrer "manuellement" des mécanismes de contrôle de la continuation. Pour ce faire, ils ont dû repenser complètement leur premier modèle sans continuation [MDC92] pour y intégrer les continuations et ensuite, l'implanter [Jac94]. Nous aimerions développer un modèle plus général qui ne demanderait au programmeur qu'un minimum d'ajouts syntaxiques à son implantation pour obtenir les fonctionnalités voulues. Par exemple, il suffirait d'utiliser la monade M_κ "passage à la continuation" (déjà présentée) et d'ajouter quelques objets et méta-objets de base au modèle pour obtenir le même résultat que pour le modèle de Malenfant et al. [MDC96].

Voici les règles de réécriture de notre système générique. Remarquons que ces règles sont obtenues à partir de celles de la section 2.3 en ajoutant adéquatement les termes **then** et **return**.

$$\begin{aligned} o.s(a_1, \ldots, a_n) &= \mu(o) \ \textbf{then} \\ & \quad \lambda mo \rightarrow mo \,.\, \gamma(s, o) \ \textbf{then} \\ & \quad \lambda m \rightarrow m \,.\, \alpha(o, [a_1, \ldots, a_n]) \qquad (1) \\ \mu(o) &= mo \qquad (2) \end{aligned}$$

$$BMO.\gamma(\gamma, BMO) = \textbf{return}\, BL \quad (3)$$
$$BMO.\gamma(\alpha, BL) = \textbf{return}\, BA \quad (4)$$
$$\mu(BMO) = \textbf{return}\, BMO \quad (5)$$
$$\pi(ROOT) = \textbf{return}\, ROOT \quad (6)$$

Les changements par rapport aux règles de la section 2.3 sont minimes. En réalité, ces nouvelles règles ne font que rendre explicites deux particularités d'implantation : **(1)** la distinction entre un message qui n'est pas encore complètement réduit (par exemple, $mo.\gamma(s, o)$) et un message complètement réduit (par exemple, BMO ou $ROOT$), et **(2)** l'ordre de réduction des envois de messages.

Evidemment, nous n'en sommes qu'aux préliminaires de la tâche que nous voulons accomplir. Il serait intéressant d'approfondir et d'implanter complètement ce nouveau système de réécriture. Nous croyons que ce travail est important, car il constitue un premier pas pour l'intégration des avantages de deux domaines de recherche (dont nous avons montré les intérêts communs) : les langages extensibles et les langages réflexifs comportementaux (dans lesquels il y a présence de tours réflexives). En effet, il faut remarquer que nous avons obtenu un système de réécriture plus général au sens où il généralise un style d'interprétation voulu qui est le style passage par continuation déjà introduit par Malenfant et al. [MDC96] dans le modèle de base. Il est plus général aussi parce qu'il fait abstraction de certaines informations de comportement. Nous savons déjà que ces informations, en général, sont utiles aux programmeurs qui veulent appliquer des méthodes réflexives pour changer la sémantique de leurs programmes. Il est donc possible que l'abstraction du système de réécriture de certaines informations utiles mais non-essentielles au modèle mette en évidence une *séparation* des entités de la réflexion offertes aux programmeurs. Il y aurait, d'une part, un modèle monadique de base (comparable au premier modèle [MDC92] comme nous venons de l'expérimenter) qui incarne la tour réflexive du langage et, d'autre part, les extensions comportementales comme la continuation explicite.

3.3 Implantations, méthodologie et applications

Haskell a servi aux premières applications de la correspondance entre le comportement computationnel et les monades. L'aspect *horizontal* des monades a été exploité pour l'implantation du compilateur Glasgow Haskell

[Jon96], pour l'implantation d'état modifiable ("updatable state") [Wad90], pour les entrées/sorties [PJW93], l'interaction [Wad95], l'analyse sémantique [HM96], la substitution [BH94], l'impression de bonnes qualités ("pretty printing") [Hug95], la réflexion [MF96] et bien d'autres.

Du côté de l'aspect *vertical* des monades, il est reconnu que les systèmes d'inférence de type "classiques" (comme celui des langages Haskell et ML) ne suffisent pas à la modélisation méthodologique des monades. Des langages comme CLEAR [GB84a], un langage de spécification algébrique qui peut être vu comme un langage de description de monades [RB85], Quest [Car91] de Cardelli proposé par Espinosa [Esp95] ou Gofer [Jon94] utilisé par Liang [LHJ95] sont plus adéquats parce qu'ils offrent un système de type plus riche (par exemple, les types dépendants de Quest et le polymorphisme sur les types de Gofer).

L'idée de l'extension des langages par les monades est d'avoir un ensemble de monades déjà écrites dans un langage fonctionnel et de les combiner pour former des comportements computationnels complexes. Par exemple, les monades *état* et *continuation* (de la section précédente) pourraient être combinées pour ajouter au langage extensible un état global *et* un accès à la continuation du calcul. Le problème de la combinaison des monades est complexe. La composition des monades en tant que telle a été montrée impossible dans le cas général [JD93]. Aucune théorie solide n'existe pour solutionner ce problème [Wad94]. Wadler [KW92], Steele [Ste94], Espinosa [Esp95], Barr & Wells [BW85] et Liang et al. [LHJ95] ont travaillé sur le développement de solutions sans aboutir à un consensus. Du côté des langages typés, les solutions proposées par Liang et al. [LHJ95, LH96] sont remarquables. Ils utilisent le concept de **transformateur de monades** de Moggi [Mog89a] pour composer les monades. Un transformateur de monades est une sorte de monade contenant un "trou" pour accueillir un autre transformateur de monades ou une monade. Quand les "trous" sont remplis, l'entité résultante est une monade au même titre que les monades simples. Ils utilisent aussi des **escaliers** ("liftings") pour pouvoir accéder aux informations véhiculées dans les monades. Ils implémentent leurs modèles en Gofer en exploitant à fond les avantages du système de type étendu de ce langage.

Du côté des langages fonctionnels stricts, Scheme a servi de base d'expérimentation pour l'implantation de programmes monadiques [Esp95]. Par contre, relativement peu d'expérimentations ont été réalisées dans ce langage populaire comparativement aux langages paresseux. Cela s'explique

par le fait que les résultats de la programmation monadique peuvent s'obtenir aussi à l'aide des structures de contrôle "impurs" (comme `call/cc`, `shift` et `reset`) des langages fonctionnels stricts [Fil94] [DF92]. Des constructions explicites par interprétation des mécanismes de contrôle "impurs" comme cela est fait en programmation monadique ne sont donc pas nécessaires dans ces langages. Néanmoins, l'implantation des monades en Scheme apporte des avantages pratiques comparativement aux langages paresseux car, dans ce langage, il y a moyen d'optimiser le code monadique par évaluation partielle [DKM91, CDL95], une technique très performante dans le cas des monades et encore aujourd'hui très rare dans les langages paresseux.

Espinosa [Esp94, Esp95] résout le problème de la combinaison des monades en utilisant, lui aussi, des sortes de constructeurs de monades qui transforment les monades en d'autres monades plus complexes. Malheureusement, sa méthode est plus compliquée à utiliser pour l'implantation de certaines fonctionnalités comme `call/cc`. Il a aussi examiné le problème de l'ordre de composition des constructeurs de monades [Esp95].

4 Problématique et cadre du travail

La problématique que nous voulons explorer se situe dans le prolongement des travaux de Mendhekar & Friedman [MF96]. Nous les expliquons en détails avant de décrire le cadre du travail.

4.1 Algèbre de la réflexion

Mendhekar & Friedman ont approfondi la notion de réflexion monadique créée originellement par Filinski [Fil94] (voir section 6.1). Ils se sont intéressés, comme pour les travaux de Danvy & Malmkjær [DM88], à la formalisation de la relation entre niveau de base et méta-niveau. Contrairement à ces derniers (qui ont fondé leur raisonnement sur une base dénotationnelle), ils ont fondé leur raisonnement sur une base logique. En effet, ils se sont inspirés des **bi-conditions de Tarski** [Tar44] qui définit précisément le prédicat de vérité T. Ils obtiennent :

$$\frac{p \,:\, A}{\Uparrow p \,:\, T(\ulcorner A \urcorner)} \; T\text{-introduction}$$

$$\frac{p\ :\ T(\ulcorner A \urcorner)}{\Downarrow p\ :\ A}\ T\text{-élimination}$$

Les opérateurs $\Uparrow$ et $\Downarrow$ correspondent précisément aux opérateurs de réification et de réflexion que nous avons définis en section 2. Les auteurs utilisent l'isomorphisme de Curry-Howard [How80] pour associer les formules logiques aux types, les preuves de ces formules aux programmes de ces types et les réductions sur les preuves à des exécutions de programmes. De plus, si nous avons une preuve (programme) p, le résultat de la réduction de cette preuve est dit **témoin** de la proposition (ou du type du programme p) dont elle est la preuve. Les réductions possibles sur p sont aussi appelées des **représentations** de la preuve p.[13]

Intuitivement, pour la règle T-introduction, si nous avons une preuve p pour une phrase A, $\Uparrow p$ correspond à une preuve de l'assertion que la preuve de A existe. Et dans le cas de T-élimination, pour une preuve p de l'existence d'une preuve pour A, $\Downarrow p$ doit être un *témoin* de A. Les auteurs déduisent 4 propriétés pour le raisonnement sur la réflexion que doivent respecter les opérateurs $\Uparrow$ et $\Downarrow$. Nous les appellerons les **propriétés de la réflexion** :

- Si p prouve A, alors $p = \Uparrow\Downarrow p$ et $p = \Downarrow\Uparrow p$. Ces équations se nomment respectivement **inverse-1** et **inverse-2**.
- Si s et t sont des preuves qu'il existe des preuves d'une même formule A, alors $\Downarrow s = \Downarrow t$. Cette règle est appelée **propriété de réflexion**.
- Si s et t sont des preuves d'une même formule A, alors $\Uparrow s = \Uparrow t$. Cette règle est appelée **propriété de réification**.

De ces propriétés, les auteurs tirent trois classes de langages réflexifs : la réflexion statique[14], la réflexion de monade et la réflexion comportementale.

- La **réflexion statique** est une puissante réflexion structurelle dans laquelle nous avons deux opérateurs réflexifs qui se complètent : `quote` et `eval`. L'opérateur `quote` offre un mécanisme pour réifier les programmes et les rendre accessibles sous une structure facilement décomposable. Nous pensons ici évidemment à l'opérateur `quote` de

[13] Cette correspondance et la terminologie associée sont mal expliquées dans l'article. Elles induisent de mauvaises interprétations des définitions qui vont suivre. Possiblement, il faudrait les reconsidérer pour vérifier leur consistance.

[14] A ne pas confondre avec la notion du même nom développée par Malenfant et al. [MJD96] qui n'a aucun lien avec la réflexion statique discutée ici.

Lisp qui représente les programmes réifiés sous forme de listes polymorphiques. L'opérateur `eval` représente l'interprète d'évaluation de programmes du langage. La présence de cet opérateur ne signifie pas qu'il y ait réflexion comportementale, car la sémantique de `eval` n'est pas représentée dans le langage et ne peut pas être changée par le programmeur. Si nous associons sémantiquement `quote` avec notre opérateur de réification $\Uparrow$ et `eval` avec l'opérateur de réflexion $\Downarrow$, nous pouvons vérifier si ces opérateurs respectent les propriétés de la réflexion. Dans l'article de Mendhekar & Friedman, on montre que la propriété de réification n'est pas respectée[15].

- La **réflexion de monade** est d'abord attribuée à Moggi [Mog89b] puis à Filinski [Fil94]. La réflexion de monade *à la Moggi* est l'association entre $\Uparrow$ avec un opérateur $[_]$ et $\Downarrow$ avec un opérateur de la forme `Let _` $\leftarrow$ `_ in _` (voir section 5.2 pour plus de détails sur ces opérateurs). Les opérateurs $[_]$ et `Let _` $\leftarrow$ `_ in _` appartiennent au méta-langage de Moggi (voir section 5.2) et servent à donner une sémantique catégorique (sorte de sémantique abstraite) à un λ-calcul abstrait de certains comportements computationnels. Mendhekar & Friedman [MF96] ne vérifient pas si ces opérateurs suivent leurs propriétés.

  La réflexion de monade *à la Filinski* se rapproche beaucoup de la réflexion *à la 3-Lisp*. En effet, dans ce cas, les opérateurs $\Uparrow$ et $\Downarrow$ sont définis à l'aide des opérateurs `shift` et `reset` (voir section 6.1). Mendhekar & Friedman affirment (sans le démontrer complètement) que ces nouvelles définitions de $\Uparrow$ et $\Downarrow$ respectent les propriétés précédentes de la réflexion. Ils concluent de ce fait que cette *réflexion de monade* est appropriée pour le raisonnement équationnel.

  Il est déplorable de voir que les auteurs ne définissent pas "monadiquement" les opérateurs `shift` et `reset`. Ce manque d'uniformité cache le fait que la continuation peut aussi être vue comme une entité monadique. Rappelons-nous encore une fois que le style passage à la continuation peut être généralisé au style monadique. En conséquence, il est possible de définir plus uniformément cette réflexion de monade en considérant la "combinaison" de **deux** monades : la monade continuation (au-dessus) et la monade générale M (en dessous). A l'aide de la méthode de combinaison des monades de Liang et al. [LHJ95], la monade résultante, combinaison des deux précédentes, peut être

[15] Voici un exemple : $(\lambda x \to x)\ 3 =_\beta 3$ sémantiquement mais (`quote` $((\lambda x \to x)\ 3)) \neq_\beta$ (`quote 3`).

étudiée indépendamment pour y découvrir ses propriétés propres.[16] Les opérateurs `shift` et `reset` se définissent facilement en fonction de la monade résultante. Et a fortiori, les opérateurs $\Uparrow$ et $\Downarrow$ obtiennent une définition formelle. Cette approche nous semble plus uniforme, car elle centralise l'analyse vers la notion de monade principalement. Nous allons approfondir cette approche dans nos recherches ultérieures.

- La dernière classe se compose des langages avec *réflexion comportementale* (*à la 3-Lisp*). Mendhekar & Friedman en disent peu de choses sauf pour mentionner leurs travaux précédents [MF93] sur la caractérisation de la réflexion par un système de réécriture. Ils reconnaissent par contre que leurs propriétés de réflexion n'ont plus de sens dans ce contexte. Ils échouent donc à caractériser les propriétés du raisonnement pour la réflexion comportementale.

Mendhekar & Friedman [MF96] élaborent plus tard dans leur article la notion de **monades réflexives**. C'est une réflexion de monade *à la Filinski* avec des contraintes supplémentaires. Elles sont les suivantes. Le programmeur définit l'opérateur d'extension ($_^*$, voir la définition C.2) sans utiliser les opérateurs de réflexion ($\Uparrow$ et $\Downarrow$). Conceptuellement, la définition de l'extension constitue le seul *méta-niveau* du langage. Il y a donc un langage de base (le λ_v-calcul typé avec les opérateurs $\Uparrow$ et $\Downarrow$) et un méta-langage (le λ_v-calcul typé et des opérateurs pour la manipulation des "représentations" des termes). Le fait qu'il soit interdit d'utiliser les opérateurs de réflexion (dans la définition de $_^*$) empêche donc l'apparition de tour réflexive. La manipulation de "représentations" de termes ne peut se faire qu'à l'intérieur de la définition de l'extension. En d'autres mots, il est défendu de manipuler des "représentations" de termes au niveau de base (la réflexion se passe au méta-niveau). De plus, il est défendu de construire des "représentations" de termes sauf par l'utilisation de l'opérateur $\Uparrow$.

Toutes ces contraintes permettent de *séparer* le raisonnement du niveau de base (équationnel standard du λ_v-calcul) du raisonnement du méta-niveau (dans lequel il y a manipulation de "représentations" sans changer la sémantique des termes représentés). Les auteurs montrent alors que le λ_v-calcul réflexif obtenu par l'introduction des monades réflexives est une extension conservatrice du λ_v-calcul en définissant une *transformation* du lan-

[16] L'étude d'une telle combinaison de monades a été réalisée auparavant avec la monade *liste* [KW92] avec certains succès. Cela constitue aussi une approche de recherche beaucoup plus modulaire, car l'attention est d'abord mise sur la monade pour, ensuite, se diriger vers la relation entre celle-ci et l'interprète qui l'accueille.

gage réflexif vers un méta-langage monadique $\lambda_{\eta,_^*}$ (contenant les opérateurs d'inclusion et d'extension avec leurs axiomes). La *transformation* "traduit" d'abord les opérateurs de réification $\Uparrow$ et de réflexion $\Downarrow$ par leur définition donnée par Filinski (voir section 6.1). Elle traduit ensuite tous les termes en style passage à la continuation pour éliminer les opérateurs `shift` et `reset`.

Il est clair que les contraintes imposées ici sont très fortes. En effet, intuitivement, les opérateurs de réflexion ont simplement le pouvoir d'arrêter temporairement le cours du calcul pour laisser place à une simple manipulation (non-réflexive) des termes et du contenu de la monade T. Ensuite, le calcul est repris à partir d'une continuation du calcul, qui se situe avant le processus de réflexion. Bien que le langage réflexif obtenu offre des techniques de raisonnement équationnel (venant du λ_v-calcul) en plus de mécanismes de raisonnement sur les opérateurs de réflexion (par l'utilisation des propriétés de réflexion), le pouvoir de la réflexion comportementale est faible. Elle se manifeste par l'absence de tour réflexive.

La propriété de réification présentée plus tôt est la règle la plus difficile à respecter pour les langages réflexifs comportementaux. Cette propriété est en réalité en opposition avec une hypothèse admise par tous et souhaitable presque toujours qui dit que les représentations des entités des programmes peuvent être changées sans changer la *sémantique* des programmes. Donc, la violation de propriété de réification est vitale pour que les langages réflexifs soient vraiment utiles. Mais d'un autre côté, cette violation rend le raisonnement équationnel impossible. Il semble donc que l'approche proposée par Mendhekar & Friedman soit inadéquate pour le raisonnement sur la réflexion. La recherche d'un langage réflexif étant une *extension conservatrice* d'un langage non-réflexif ne semble donc pas être une approche appropriée pour la réflexion comportementale.

4.2 Démarche de la recherche

Nous décrivons maintenant notre démarche pour aboutir à la classification. Nous allons d'abord reprendre les travaux de Mendhekar & Friedman présentés dans la sous-section précédente. Comme nous l'avons expliqué, la sémantique monadique proposée dans leur article n'est pas uniforme car la continuation qui est implicite dans la sémantique peut être représentée explicitement par une monade. Nous allons donc uniformiser leur sémantique en utilisant la monade continuation. Nous allons tenter d'obtenir les mêmes résultats qu'eux en étudiant la monade correspondante et sa relation à l'in-

terprète.

Nous tenterons aussi de découvrir de nouvelles techniques de raisonnement équationnel utiles pour la réflexion. Nous savons que cette recherche constitue une tâche de grande envergure. Par contre, nous allons nous limiter au raisonnement possible sur une structure monadique. Cela limite considérablement la recherche de nouvelles techniques. Il est probable que de nouvelles techniques de raisonnement se manifestent si nous adoptons une approche *algébrique* (série d'équations décrivant le rapport entre les opérateurs de réflexion et les aspects non-réflexifs du langage). Cette approche avait déjà été reconnue informellement par Smith lui-même [Smi84]. Wand & Friedman [WF88] avait ensuite parlé de *l'algèbre* des opérateurs de la réflexion. Danvy & Malmkjær ont tenté, par la suite, de caractériser cette algèbre par une approche dénotationnelle [DM88]. Comme nous l'avons montré, Mendhekar & Friedman tentent eux aussi de la définir, mais cette fois-ci par une approche logique et monadique. Rien ne dit que l'algèbre qu'ils décrivent est celle qui convient le mieux à la réflexion.

Ces recherches nous permettront ensuite d'élaborer des variantes de leur langage réflexif de façon à affaiblir ou renforcir le pouvoir de la réflexion. De la même manière, nous étudierons chacune des variantes découvertes pour y déterminer leur capacité de raisonnement équationnel.

Il est probable que certains langages réflexifs qui existent déjà ne puissent pas être formalisé par une sémantique monadique. Il sera donc important de bien déterminer les limites du pouvoir d'expression de la sémantique monadique. Si nous nous rappelons de la terminologie de la section 3, cela est relié à l'aspect *vertical* du pouvoir des monades : le pouvoir d'augmenter la puissance des langages monadiques. Nous avons vu que la combinaison de monades semble donc être une approche prometteuse pour ce problème [MF96].

Après avoir élaboré une série de nouveaux langages réflexifs avec leur sémantique monadique respective, nous classifierons ces langages en fonction de plusieurs critères :

- La capacité à tenir des raisonnements équationnels. Nous avons vu que Mendhekar & Friedman ont créé un langage réflexif dans lequel le raisonnement équationnel du λ_v-calcul est respecté. En partant de ce langage, nous croyons qu'il est possible d'obtenir une gamme de langages réflexifs dans lesquels la puissance du raisonnement équationnel est graduellement limitée pour laisser place à d'autres techniques de

raisonnement équationnel.

- La puissance d'expression de la réflexion. Il y a plusieurs moyens de limiter ou d'augmenter la puissance de la réflexion. Par exemple, il est possible de diminuer ou d'ajouter la quantité d'information réifiable relativement au programme (l'environnement, la continuation, une représentation du programme, etc.). Il est aussi possible de limiter l'accès à l'information réifiée (accès en lecture seulement, en lecture/écriture, etc.). Enfin, nous pouvons restreindre le pouvoir de manipulation de l'information réifiée (transformation respectant plus ou moins la sémantique).
- La présence ou non d'une tour réflexive. Nous avons vu que le langage de Mendhekar & Friedman ne contient pas de tour réflexive. Que se passe-t'il si nous obtenons un langage réflexif monadique dans lequel il y a une tour réflexive ? Quel est l'impact de la tour réflexive sur le raisonnement équationnel ?
- La capacité à être implanté efficacement. Les travaux de Malenfant et al. [MJD96] ont élaboré des critères pour déterminer si un langage réflexif est apte à être compilé efficacement ou non. Nous utiliserons ces critères aussi pour notre classification.

Par la suite, il sera nécessaire de *valider* la classification en relation avec les langages réflexifs existants. Cette validation se fera en appliquant les techniques de raisonnement découvertes à l'intérieur des langages réflexifs existants correspondant à leur position estimée dans la classification établie.

Enfin, ayant obtenu une série de sémantiques monadiques pour la réflexion, il sera intéressant de mettre en évidence les monades les plus importantes pour le développement de langages réflexifs monadiques. Intuitivement, il est possible que l'ensemble de ces monades composent une *base* pour engendrer *l'espace* de conception des langages réflexifs monadiques.

5 Théorie des catégories et monades

> Category theory comes, logically, before the λ-calculus. [Mog91]
>
> *Eugenio Moggi*

La théorie des catégories [Lan71] est une branche des mathématiques qui a été développée dans les années 40. Tout a commencé par un article

de Eilenberg et Mac Lane qui répondait au besoin de ce temps de relier des branches disparates des mathématiques pures. En particulier, des sujets tels l'algèbre et la topologie ont été fusionnés pour former la topologie algébrique. La théorie des catégories a permis le développement d'une base solide pour cette fusion.

Dans cette section, nous discuterons d'abord de la théorie des catégories en informatique et ensuite de l'application originelle des monades en sémantique des langages.

5.1 Introduction

John Reynolds a été le premier à appliquer cette théorie en conception de langages de programmation [Rey80]. Par la suite, il y a eu plusieurs recherches pour formaliser l'informatique en théorie des catégories, comme la théorie du λ-calcul [Has95], la sémantique dénotationnelle [Mog89b, Mog91] et la spécification algébrique [GB84a, GB84b, TBG91], mais aussi dans les aspects pratiques de l'informatique comme la conception de langages [Wad92], les techniques d'implémentation [CCM85] [JHHP93], la dérivation de programmes [Spi93] et d'autres.

Malheureusement, cette théorie est difficile à comprendre et demande une grande patience à celui qui veut l'utiliser, car elle offre des concepts très abstraits qui n'ont pas, pour le profane, de significations naturelles autres que pour des exemples plutôt simplistes. Par contre, curieusement, d'après l'expérience d'enseignement de cette discipline, il apparaît que l'appréciation de cette théorie vient plus rapidement pour l'informaticien théorique que le mathématicien parce qu'il est peut-être plus facile de trouver des exemples d'expérimentation en informatique qu'en mathématique.

Au cours de la dernière décennie, plusieurs mathématiciens et informaticiens ont tenté d'expliquer cette théorie. C'est pour cette raison qu'aujourd'hui, on dénombre une grande quantité de rapports de recherche vulgarisant le sujet pour les informaticiens en général [Hoa88a] [RB88] [Pie90] [Fok92b] [Bou93], les théoriciens de l'informatique [GS89] [BW90] [Pie91] [Ten91] [AL91] [Poi92] [Spi93] ou les mathématiciens intéressés aux langages [LS86] [LS91] [Gog91] [vO95].

Mais pourquoi cette volonté d'unir la théorie la plus générale et la plus abstraite des mathématiques [Hoa88b] avec la programmation ? Une réponse est clairement expliquée dans le livre de Rydeheard & Burstall [RB88]. En

effet, il est dit que la théorie des catégories peut être d'un grand intérêt pour les informaticiens, car elle opère sur le même niveau de généralité que la logique et la programmation. La fonction essentielle de cette théorie est de *définir* et construire des définitions. Ce sont justement les tâches principales de l'informaticien.

Il y a une autre raison qui motive l'informaticien à apprendre cette théorie. C'est qu'elle est en grande partie *constructive* [RB85, RB88]. Les théorèmes qui affirment l'existence de certaines entités catégoriques sont prouvés par construction explicite de celles-ci. De ce fait, plusieurs ont développé des méthodes pour extraire les aspects mécaniques du raisonnement catégorique [Hue86] [RB88] [Fok92a]. Aujourd'hui, on retrouve plusieurs langages utiles ou spécialisés à cette théorie : Nuprl [C +86], Mizar [Rud92], LEGO [Pol94], Coq [DFH +93], HOL [MT93] et beaucoup d'autres. D'ailleurs, on peut voir la théorie des catégories comme une collection d'algorithmes. Par contre, ces algorithmes ont un degré de généralité dépassant les niveaux rencontrés normalement en programmation. C'est une raison de plus qui justifie l'effort de compréhension de cette théorie.

Il faut comprendre que cette théorie est avant tout une façon de penser particulière. A première vue, les résultats et les concepts sont simples voire simplistes car, justement, ils sont très abstraits. Mais, sans cette tournure d'esprit développé dans un monde "à part", ces concepts simples n'auraient peut-être jamais été mis en évidence formellement. D'ailleurs, nous avons ajouté en annexe A un exemple qui, nous l'espérons, donne une idée du *raisonnement catégorique* et de sa familiarité avec la programmation. Il s'est avéré que la programmation fonctionnelle utilise abondamment certains de ces concepts (en les appelant simplement des fonctions d'ordre supérieur) sans reconnaître leur caractère distinct l'un de l'autre et leurs relations entre eux, comme a pu le montrer la théorie des catégories. Les monades computationnelles en sont un exemple patent. La sous-section suivante présente l'origine des monades en programmation et leur relation à la réflexion.

5.2 Théorie du calcul de Moggi

Le méta-langage computationnel de Moggi [Mog91] est à l'origine même de la programmation monadique [Wad90]. Il est considéré comme un des développements sémantiques les plus significatifs de la dernière décennie [BW96].

Le méta-langage de Moggi est un langage typé qui met en évidence la distinction entre *calcul* et *valeur* (comme l'avait déjà fait Reynolds [Rey72] avec les termes *sérieux* et *triviaux*). En particulier, il utilise les monades pour rendre explicites certaines informations computationnelles. Si e est une expression *calcul* (ou un terme sérieux) alors $[e]$ retourne la *valeur abstraite* de e. Simplement, cet opérateur a le typage suivant $[_] : \alpha \to M\alpha$. Il correspond donc à notre fonction `return`, tandis que l'expression `Let` $x \leftarrow e_1$ `in` e_2[17] évalue e_1, lie le résultat à x et évalue e_2. Cet opérateur est le même que notre `then`.

Il existe des *transformations* standards du λ_v-calcul (passage par valeur) et du λ_n-calcul (passage par nom) vers le style passage à la continuation [Plo75]. Moggi a généralisé ces *transformations* en développant une sémantique du calcul (qui est une traduction des différents λ-calculs (style direct, appel par valeur, appel par nom, etc.) vers son méta-langage logique. Le méta-langage de Moggi a été reconnu "semblable" à plusieurs autres méta-langages. En voici quelques uns.

Dans un article de Hatcliff & Danvy [HD94], il est montré qu'il existe une correspondance biunivoque entre la transformation de différents λ-calcul vers le méta-langage de Moggi et une transformation "générique" vers le style passage à la continuation. Cela permet de mieux comprendre la relation étroite entre le style passage à la continuation et le style monadique.

Filinski a montré de son côté que le style monadique est de même puissance que le λ_v-calcul avec "continuation composable" [DF90] (continuations de plein droit, état et opérateurs `shift` et `reset`). Pour ce faire, il montre une *transformation* d'un méta-langage à l'autre et vice versa. De plus, il montre que les opérateurs de réification ($\Uparrow$) et de réflexion ($\Downarrow$) peuvent être définis en fonction des opérateurs `shift` et `reset` montrant un lien certain avec la réflexion (voir section 6.1). La relation entre le style monadique et les continuations composables a aussi été reconnue par Wadler [Wad94] et Kieburtz, Agapiev & Hook [KAH92].

Enfin, Benton & Wadler [BW96] montrent que les *transformations* des

[17] Cette notation met en évidence le fait que la monade est reconnue par plusieurs comme une construction *du premier ordre* et non pas d'ordre supérieur comme cela semble être sous-entendu dans sa définition [Mog89b] [Wad95]. Il est donc possible d'implanter la programmation monadique dans un langage du premier ordre. Dans un article récent de Wadler [Wad95], une première implantation du premier ordre est esquissée. De plus, il est montré que, par nature, la monade est limitée théoriquement [RB85] et pratiquement [Wad94] à un certain pouvoir d'expression.

λ-calculs (direct, par valeur et par nom) vers le méta-langage de Moggi correspondent *exactement* à trois *transformations* de la logique intuitionniste vers la logique linéaire intuitionniste de Girard [Gir87]. Pour ce faire, ils utilisent la notion générale d'adjonction (voir section A). Leurs travaux montrent un lien intéressant entre logique monadique de Moggi et logique linéaire de Girard.

Toutes ces correspondances laissent penser que la logique monadique, la logique linéaire, les "logiques" avec continuations de plein droit, la logique modale (reconnue par Moggi [Mog91]) et la logique de la réflexion sont interreliées. Il est remarquable de voir que tous ont deux opérateurs particuliers en opposition : la logique monadique a l'inclusion et l'extension ; la logique linéaire a les opérateurs *pourquoi pas* ("why not") et *bien sûr* ("of course") ; la "logique" des continuations composables a, entre autres, `shift` et `reset` ; la logique modale a les opérateurs de *possibilité* et de *nécessité* et, enfin, la logique de la réflexion a les opérateurs de réflexion et de réification.

De plus, en logique modale, les opérateurs de *possibilité* et de *nécessité* sont souvent modélisés par des monades [Mog91]. En logique linéaire, les opérateurs *pourquoi pas* ("why not") et *bien sûr* ("of course") sont modélisés eux aussi par des monades [Mog91]. Nous avons vu que les opérateurs de réification et de réflexion peuvent être modélisés par des monades. Il y a encore beaucoup de travail à faire de ce point de vue, comme nous l'avons mentionné. Nous allons tenter de contribuer à augmenter la compréhension de tout cela. En somme, la reconnaissance formelle de similitude entre toutes ces logiques est possible grâce à la grande généralité de la théorie des catégories. En plus d'encourager le partage de ces domaines de recherche, nous conjecturons que cela pourra servir au développement de la théorie générale de la réflexion.

6 Mise en perspective

Depuis quelques années, plusieurs approches de classification de la réflexion ont été proposées. Il y en trois qui sont particulièrement intéressantes. Un premier article de Demers & Malenfant [DM95] montre par une approche historique les relations conceptuelles et terminologiques entre les différents développements de la réflexion en programmation fonctionnelle, par objets et logique. Clavel & Meseguer [CM96] utilisent une approche logique pour développer une théorie générale de la réflexion. Ils

décrivent les aspects essentiels des systèmes réflexifs, mais la classification en tant que telle est encore préliminaire. Malenfant, Jacques & Demers [MJD96] décrivent une classification basée sur la compilation pour discriminer les langages réflexifs aptes à être compilés efficacement de ceux qui ne peuvent pas l'être. Leur discrimination est basée principalement sur la notion de *réflexion statique* (expliqué en section 2.3).

Bien que ces trois classifications soient enrichissantes relativement à leur approche respective, aucune ne décrit suffisamment et ne compare les techniques de raisonnement de programmes reliées à chaque classe de langages réflexifs. De ce point de vue, sans aboutir à une classification, certains chercheurs ont travaillé sur le raisonnement de la réflexion, mais se restreignant à certains langages réflexifs précis. Par exemple, Mendhekar & Friedman [MF93] se sont intéressés au raisonnement équationnel possible dans les langages réflexifs *à la 3-Lisp*. Comme nous l'avons expliqué précédemment, ils ont poursuivi leurs travaux du raisonnement en utilisant une approche monadique [MF96]. De son côté, Moggi [Mog91] s'est intéressé aux langages extensibles et au raisonnement équationnel sur ceux-ci. Il utilise une approche formelle basée sur la logique catégorique et les monades (voir section 5.2).

Du côté des langages extensibles, beaucoup d'études sur le raisonnement ont été menées alors que pour les langages *à la 3-Lisp*, on en retrouve peu. Plusieurs extensions possibles de ces langages ont été étudiées particulièrement. Par exemple, des théories sur le raisonnement ont été élaborées en relation aux continuations [SF92], aux états modifiables [SF93] et aux exceptions [Spi90].

6.1 Monades et réflexion : les origines

Concernant le raisonnement et les monades, dans un article de Danvy et al. [DKM91], il est mentionné brièvement qu'il existe une ressemblance étrange entre la relation entre un terme de type α et sa "représentation monadique" $T\alpha$[18], et la reconstruction de Lisp opérée par Smith [Smi82]. Il est dit que durant un certain calcul, un interprète n'accède pas précisément aux valeurs (de type α) à calculer mais plutôt à des *représentations* de ces valeurs (de type $T\alpha$).

C'est un article de Filinski [Fil94] qui établit les premiers liens sérieux

[18] T est un constructeur de types d'une monade quelconque.

entre la réflexion et les monades. Filinski développe alors la notion de **réflexion monadique** par analogie à la réflexion comportementale. Bien que la réflexion monadique ait très peu de liens avec la réflexion comportementale, elles ont quelques ressemblances qui devraient être étudiées en détails.

D'abord, comparativement à la dichotomie presque toujours présente entre niveau de base et méta-niveau de la réflexion de comportement, la réflexion monadique sépare les termes en deux groupes à l'aide d'un constructeur de type T. S'il existe un terme de type α, alors ce terme représente un *calcul* et celui-ci correspond à une *valeur abstraite* de type $T\alpha$. Cette vision quelque peu bizarre à première vue vient précisément des travaux de Moggi sur la vulgarisation de la sémantique du calcul monadique (voir section 5.2). Nous supposons alors que les termes-calculs de type α sont d'un certain "méta-niveau" et les termes-valeurs de type $T\alpha$ sont dans le niveau de base. La réflexion monadique définit donc la relation entre ces niveaux comme suit :

$$\frac{\Gamma \vdash v : T\alpha}{\Gamma \vdash \varepsilon(v) : \alpha}$$

$$\frac{\Gamma \vdash v : \alpha}{\Gamma \vdash [v] : T\alpha}$$

Pour toute valeur $v : T\alpha$, $\varepsilon(v)$ représente la *réflexion* de la valeur de v vers un calcul "avec effet" de type α. De façon contraire, si nous avons un calcul $v : \alpha$, alors $[v]$ représente la réification de v vers une valeur "sans effet" de type $T\alpha$ [Fil94].

De plus, cette réification et cette réflexion monadique doivent respecter les équations suivantes qui se déduisent des lois monadiques de $_^*$ et η (définition C.2) :

$$\varepsilon([v]) = v$$

$$[\varepsilon(v)] = v$$

Filinski offre aussi une "implantation" des opérateurs de réification et de réflexion en fonction des opérateurs `reset` et `shift`[19] comme suit :

$$\Uparrow e = [e] = (\ \texttt{reset}\ \ (\eta\, e))$$

[19]Intuitivement, `reset` installe une indicateur sur la queue d'exécution. De son côté, `shift` prend une procédure en argument. Il prend la continuation du calcul jusqu'au premier indicateur (placé par `shift`), la convertit en une procédure et passe l'argument à la continuation "réifiée".

$$\Downarrow e = \varepsilon(e) = (\texttt{shift} \ (\lambda k \rightarrow (k^* \ e))$$

Par la suite, comme nous l'avons vu, Mendhekar & Friedman [MF96] réutilisent ces définitions pour montrer que $\Uparrow$ et $\Downarrow$ respectent bien leurs lois de la réflexion.

6.2 Autres formalismes sémantiques

La sémantique des actions a été créée par Mosses et Watt [Mos92] pour remédier au manque de modularité et d'extensibilité de la sémantique dénotationnelle. L'unité fondamentale de la sémantique est l'*action* qui permet de décrire des comportements computationnels précis. Ces actions peuvent être composées ensemble pour former des mécanismes de contrôle plus importants. Ce langage de spécification est intéressant pour la description de langages du premier ordre, mais la notation devient lourde quand il est question de décrire des langages d'ordre supérieur ayant des mécanismes de contrôle manipulant la continuation, comme `call/cc` de Scheme.

La sémantique directe extensible [CF94] est une façon d'étendre la définition dénotationnelle d'un langage sans en changer les définitions déjà établies. Elle permet de construire des λ-interprètes et de les composer entre eux pour former des λ-interprètes de langages fonctionnels complets.

La sémantique des moniteurs [KHC91] est aussi une extension de la sémantique dénotationnelle. Elle a pour but d'offrir un cadre adéquat pour décrire les mécanismes de contrôle qu'on retrouve dans les programmes de déverminage, de "profilage", de trace d'exécution, etc. L'unité de base de cette sémantique est le *moniteur*, sorte de démon qui "surveille" l'exécution d'un programme. Cette sémantique est particulièrement intéressante pour décrire la réflexion. Cela sera étudié par Marco Jacques dans les prochaines années [Jac96].

7 Conclusion

Le but de ce doctorat est d'explorer l'espace des langages réflexifs dont on peut donner une sémantique monadique et une théorie du raisonnement équationnel.

7.1 Contributions

Les contributions au domaine se composent de plusieurs volets :

- L'élaboration de sémantiques formelles monadiques pour différentes familles de langages réflexifs. Il est possible que certains langages réflexifs ne puissent pas être formalisés d'une façon monadique. Il sera donc intéressant de déterminer plus exactement les limites de l'expression des monades computationnelles.
- Le développement d'extension aux théories du raisonnement équationnel pour traiter la réflexion. Nous savons que le raisonnement équationnel est important au programmeur pour l'aider à comprendre à un haut degré d'abstraction ses programmes et cela est important aussi au concepteur de langage pour l'aider à l'optimisation des programmes. La théorie des catégories peut aussi aider au développement de nouvelles théories du raisonnement équationnel.
- Etablir un ensemble de monades qui caractérise l'espace des langages réflexifs. La combinaison d'une façon particulière de ces monades nous permettra peut-être d'atteindre tout langage réflexif contenu dans l'espace en question.
- La généralisation des propriétés entre les langages réflexifs voire entre les familles de langages réflexifs. La généralisation constitue la force majeure de la théorie des catégories et des monades. Il est reconnu qu'il n'existe pas à l'heure actuelle de théorie générale pour la réflexion. Nous croyons que la théorie des monades peut aider à établir une telle théorie unificatrice.
- Le développement d'une classification des langages réflexifs et du raisonnement équationnel possible dans chaque classe. Cette classification permettra d'évaluer, du moins théoriquement, la *complexité* des langages réflexifs. Cette complexité est indirectement reliée à la difficulté d'être implantés efficacement.

7.2 Echéancier

La première partie de mon doctorat (automne 94 à été 96 dans la figure 1) est constituée de cours, d'examens et d'apprentissage des outils pertinents pour les recherches de la deuxième partie (automne 96 à automne 97 dans la figure 1). Dans cette dernière, nous allons effectuer la recherche proprement dite. Elle se décompose en quatre phases :

- Reformulation de la sémantique de Mendhekar & Friedman [MF96]. Exploration des variantes de cette sémantique et l'élaboration d'une première classification des langages réflexifs basée sur le pouvoir d'expression de la sémantique monadique.
- Exploration des techniques de raisonnement équationnel sur les classes de langages réflexifs auxquels une sémantique monadique a pu être développée grâce à la première phase.
- Expérimentation des résultats obtenus en relation aux langages réflexifs existants. Cette phase permettra de valider les résultats en vérifiant si les raisonnements équationnels mis en évidence sont exacts appliqués aux langages réflexifs correspondants.
- Classification définitive des langages réflexifs caractérisés en rapport à la puissance des techniques de raisonnement obtenues.

Période	Description du travail
Automne 94	Premier examen pré-doctoral et cours
Hiver 95	Cours de logiques d'intelligence artificielle
Eté 95	Introduction et conférence en théorie des catégories
Automne 95	Théorie des monades
Hiver 96	Recherche d'un sujet de doctorat
Eté 96	Deuxième examen pré-doctoral
Automne 96	Elaboration de sémantiques monadiques réflexives
Hiver 97	Elaboration de techniques de raisonnement
Eté 97	Expérimentation sur ces sémantiques
Automne 97	Rédaction de la thèse

FIG. 1: Sommaire de l'échéancier

A Un exemple en programmation : les adjonctions

> ... adjoints occur almost everywhere in many branches of mathematics. ... a systematic use of all these adjunctions illuminates and clarifies these subjects.
>
> *Saunders Mac Lane*

Pour illustrer la théorie des catégories, nous présenterons une application des adjonctions dans un langage fonctionnel typé. Nous nous sommes basé

sur les travaux de Spivey [Spi89][Spi93], Fokkinga [FM94][Fok92b] et Wadler [Wad90, KW92]. L'adjonction est sans aucun doute un des concepts des plus importants des catégories. De plus, les adjonctions apparaissent souvent en mathématiques et en programmation. Les adjonctions sont intimement liées aux monades. En effet, à chaque adjonction, on peut induire une monade. Et inversement, chaque monade vient d'au moins une adjonction.

Nous supposons que le lecteur sait ce qu'est une catégorie, un diagramme commutatif, un foncteur et une transformation naturelle. Sinon, il peut consulter l'annexe B d'abord.

Considérons maintenant les définitions récursives sur les listes suivantes dans un langage fonctionnel typé :

```
Type longueur : [ a ] -> Int.     % "a" est une variable de type.

longueur([]) = 0
longueur(s ++ t) = longueur(s) + longueur(t)
longueur([_]) = 1
```

De même façon, voici une définition d'une fonction qui accumulent les éléments d'une liste dans un ensemble.

```
Type elements : [ a ] -> { a }.

elements([]) = {}
elements(s ++ t) = elements(s) U elements(t)
elements([x]) = { x }
```

Il est clair que ces deux définitions sont similaires. Pourrions-nous exprimer complètement et formellement ce qui les rend semblables ? La théorie des catégories peut le faire un peu comme un programmeur le fait dans son langage fonctionnel favori en définissant des fonctions d'ordre supérieur suffisamment générales utiles aux définitions des fonctions `longueur` et `elements`. Pour le programmeur, les fonctions d'ordre supérieur utilisées dans les définitions des deux fonctions représenteraient ce qu'il y a de semblable entre les deux définitions.

La théorie des catégories "définit" elle aussi des fonctions générales mais, en plus, elle explicitera de façon exhaustive[20] les propriétés qui réunissent ces deux définitions.

[20] Exprimé par des résultats théoriques de complétude.

Pour ce faire, tentons de comprendre ces définitions dans le monde des catégories, c'est-à-dire dans les termes de certains morphismes et leur composition. Soit le type α. Considérons alors le monoïde $\mathit{liste}_\alpha = (\alpha*, +\!\!+_\alpha, [\,]_\alpha)$, le monoïde $\mathit{Ens} = (\{\alpha\}, \cup, \{\})$ et le monoïde $N = (\mathcal{N}, +, 0)$[21]. Il y a un rappel de la notion de monoïde en annexe D.

En fait, il serait facile de montrer que les deux fonctions de programmation précédemment définies sont précisément des homomorphismes entre monoïdes. En réalité, les deux premières équations des définitions de `longueur` et de `elements` nous disent justement qu'ils sont des définitions d'homomorphisme. La dernière équation de la définition de `longueur` exprime précisément la commutativité du diagramme suivant :

$$\begin{array}{ccc} \alpha & \xrightarrow{[\cdot]_\alpha} & \alpha* \\ & {\scriptstyle K_{\{1\}}} \searrow & \downarrow K^{\#}_{\{1\}} = \texttt{longueur} \\ & & \mathcal{N} \end{array}$$

La fonction $K_{\{1\}} : \alpha \to \{1\}$ retourne toujours 1 quel que soit l'argument donné. $^{\#}$ est un symbole pour différencier les noms de fonctions. La nom de fonction $K^{\#}_{\{1\}}$ est donc seulement un autre nom pour désigner la fonction `longueur`. De même, nous avons le diagramme suivant pour la dernière équation de la définition de `elements`.

$$\begin{array}{ccc} \alpha & \xrightarrow{[\cdot]_\alpha} & \alpha* \\ & {\scriptstyle f} \searrow & \downarrow f^{\#} = \texttt{elements} \\ & & \mathit{Ens} \end{array}$$

où f est la fonction définie comme $f(x) = \{x\}$ pour tout x.

Les propriétés homomorphiques des fonctions `longueur` et `elements` avec les propriétés de commutativité des diagrammes précédents correspondent exactement aux définitions dans notre langage. Donc, clairement, dire que notre programme `longueur` et notre programme `elements` définissent bien nos fonctions revient à dire qu'il existe deux *homomorphismes uniques satisfaisant les deux triangles commutatifs précédents.*

[21] L'ensemble $\mathcal{N}$ représente l'ensemble des nombres naturels.

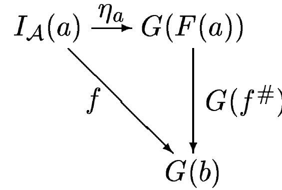
Nous verrons maintenant que nos deux très simples programmes peuvent être associés à deux adjonctions précises.

A.1 Définition.
Une **adjonction** *consiste en*

- *Deux catégories* $\mathcal{A}$ *et* $\mathcal{B}$,
- *Un foncteur* $F : \mathcal{A} \to \mathcal{B}$.
- *Un autre foncteur* $G : \mathcal{B} \to \mathcal{A}$.
- *Une transformation naturelle* $\eta : I_{\mathcal{A}} \to G \cdot F$.

Ceux-ci satisfont la propriété suivante :

Pour chaque objet b *de la catégorie* $\mathcal{B}$ *et* $\mathcal{A}$*-morphisme* $f : a \to G(b)$, *il existe un* **unique** $\mathcal{B}$*-morphisme* $f^{\#} : F(a) \to b$ *tel que le triangle suivant commute.* $I_{\mathcal{A}}$ *est le foncteur identité.*



□

Comment peut-on faire la correspondance entre cette définition générale qui ne signifie rien en première analyse ? D'abord, sachant que la composition d'homomorphismes est un homomorphisme, nous pouvons facilement montrer que si nous prenons les monoïdes comme des objets et les homomorphismes entre monoïdes comme des flèches, nous obtenons bien une catégorie (que nous nommerons $Mono$). De plus, chaque monoïde, en lui-même, peut former une catégorie et les homomorphismes entre celles-ci sont, dans ce cas, des foncteurs entre catégories.[22]

Maintenant, examinons la définition d'adjonction. Prenons le cas où les catégories $\mathcal{A}$ et $\mathcal{B}$ sont le monoïde $Ens = (\{\alpha\}, \cup, \{\})$ et la catégorie des monoïdes $Mono$. Prenons aussi le foncteur F simplement pour $*$. Le foncteur G est pris pour le foncteur "amnésique" ("forgetful functor") $U : Mono \to Ens$ qui, pour chaque monoïde, retourne l'ensemble sous-jacent. La transformation naturelle η est précisément $[\cdot]$ qui a pour effet

[22] Remarquons que si la propriété homomorphique est respectée, il est facile de "voir" des catégories et des foncteurs à plusieurs niveaux d'abstraction. Ce type de "vision" constitue l'outil principal du catégoriste.

de construire la liste singleton à partir de chaque élément. Le morphisme η_a est donc la fonction $[\cdot]_a$. La fonction (morphisme) `longueur` (c'est-à-dire $G(f^{\#})$) se retrouve donc à être l'extension de la fonction f (sur lequel on applique U) qui est la fonction constante $K_{\{1\}}$.

Remarquons qu'en faisant les substitutions adéquatement dans le diagramme de la définition d'adjonction, nous obtenons exactement le diagramme présenté pour la définition de `longueur` !

De même façon, en prenant f définie comme $f(x) = \{x\}$ pour tout x, nous obtenons la fonction `elements`. De plus, nous retrouvons exactement le diagramme commutatif respectivement à la définition de `elements`.

Nous avons donc une entité catégorique qui regroupe l'essentiel des deux programmes informatiques.

Généralement, le foncteur F est appelé **l'adjoint gauche** à G et G, **l'adjoint droit** à F. La transformation naturelle η est appelée **l'unité** de l'adjonction. De plus, à chaque objet b de la catégorie $\mathcal{B}$ et pour chaque morphisme $g : F(a) \to b$, il existe un unique morphisme $g^{\#} : a \to G(b)$. Ce morphisme respecte le triangle commutatif suivant :

$$\begin{array}{ccc} F(G(b)) & \xrightarrow{\epsilon_b} & I_{\mathcal{B}}(b) \\ F(g^{\#}) \downarrow & \swarrow g & \\ F(a) & & \end{array}$$

où ϵ_b est un morphisme construit à partir d'une transformation naturelle $\epsilon : F.G \to I_{\mathcal{B}}$ qu'on appelle le **co-unité** de l'adjonction. Dans les termes des catégoristes, le co-unité est le dual de l'unité. Nous disons que les transformations naturelles η et ϵ forment une adjonction ou que $(F, G, \eta, \epsilon) : \mathcal{A} \rightharpoonup \mathcal{B}$ est une adjonction.

Dans le cas de notre premier exemple, le morphisme $\epsilon_{(\mathcal{N},+,0)}$ ($b = (\mathcal{N}, +, 0)$ relativement au diagramme précédent) correspond à notre fonction `longueur` (ou $+/$), le morphisme $\epsilon_{(\alpha*,++,[\,])}$ correspond à notre fonction $++/$ et le morphisme ϵ_{Ens} correspond à notre fonction `elements` (ou $\cup/$). Plus généralement, la fonction $_/ : \alpha \to \alpha \to \alpha \to \alpha* \to \alpha$ correspond à peu près à la transformation naturelle ϵ. Donc, $[\cdot]$ et $_/$ forme une adjonction et $(*, U, [\cdot], _/) : Ens \rightharpoonup Mono$ est l'adjonction découverte. Ici, $_/$ ne correspond pas tout à fait à la fonction `fold` mais lui ressemble. En fait, elle est plus générale que `fold` (prendre $\alpha = \beta*$ pour obtenir `fold`). Par contre,

$[\cdot]$ correspond précisément à la fonction *unité* de la première définition de la monade liste.

Nous pouvons généraliser comme l'a fait Wadler [Wad90] (voir la section 3.1) et remplacer le constructeur de liste par un constructeur quelconque M. Nous obtenons que $(M, U,$ **return**$, fold)$: $Ens \rightharpoonup Mono$ forme aussi une adjonction où $fold$ est notre fonction $_/$ mais généralisée au constructeur M. Cette fonction $fold$ est appelée un **catamorphisme** [Fok94]. Son type est $fold\ :\ (\alpha \to \alpha \to \alpha) \to M(\alpha) \to \alpha$. En quelque sorte, cette fonction prend en paramètre l'opérateur binaire du monoïde sous-jacent à la monade M (s'il existe) [KW92] pour y construire la fonction $join$ de la monade M.

Remarquons aussi que, dans la définition d'adjonction, il est affirmé qu'il existe un *unique* morphisme respectant le triangle commutatif. En théorie des catégories, quand il est question d'affirmer l'existence d'une et une seule (unique) entité catégorique si certaines propriétés sont respectées, nous disons qu'il y a là une **propriété universelle**. La notion de propriété universelle joue un rôle très important. Cette propriété est dite universelle au sens où tout autre entité respectant les mêmes propriétés peut se "factoriser" en l'entité "canonique" (celle qui a la propriété universelle). En d'autres mots, il y a une sorte d'isomorphisme catégorique entre toutes les entités respectant la propriété universelle.

On en tire que les propriétés de $[\cdot]$ et $_/$ sont suffisantes pour déterminer le monoïde des listes à isomorphisme près. Cela signifie que tout propriété algébrique sur les listes est une conséquence des lois reliées à $[\cdot]$ et $_/$.

De plus, nous pouvons affirmer que chaque homomorphisme sur les listes est uniquement déterminé par sa restriction à la liste singleton (tiré du fait que la fonction singleton est l'unité de l'adjonction) et vice versa. En d'autres mots, nous obtenons une forme normale pour les homomorphismes de listes. Pour montrer que deux homomorphismes sont égaux, il suffit de montrer qu'ils agissent de mêmes façons sur les singletons. Cela correspond au *théorème d'homomorphisme* de Bird [Bir87]. Ce théorème est en fait, une autre façon de voir l'adjonction reliée aux transformations naturelles $[\cdot]$ et $_/$.

Encore une fois, il est possible de généraliser à un constructeur de type quelconque. Toute propriété algébrique sur une monade M doit donc être une conséquence des lois reliées à **return** et le catamorphisme. De plus, chaque homomorphisme sur M est uniquement déterminé par sa restriction au "singleton" du constructeur M (qu'on obtient en appliquant **return**).

Par ailleurs, à partir d'une adjonction, il est très simple de construire la monade qui lui est associée. En effet, toute adjonction $(F, G, \eta, \epsilon) : \mathcal{A} \rightharpoonup \mathcal{B}$ met en évidence une monade sur $\mathcal{A}$. Cette monade est le triplet $(FG, \eta, F\epsilon G)$ [Lan71].

Si nous construisons la monade associée à l'adjonction $(*, U, [\cdot], _/) : Ens \rightharpoonup Mono$, nous retrouvons $(*, [\cdot], +\!\!+/)$ qui est la monade liste bien connue [KW92] (ou $(*, map_l, unit_l, join_l)$ correspondant à la définition par compréhension donnée par Wadler). Plus généralement encore, pour chaque adjonction $(M, U, unit, fold) : Ens \rightharpoonup Mono$, nous formons directement la monade générale $(M, map, unit, join)$.

Il faut retenir seulement que l'adjonction et la monade sont des concepts qui offrent différentes vues d'un certain univers formel et abstrait.

B Concepts de base : catégorie, foncteur, etc.

La programmation fonctionnelle typée s'intéresse aux fonctions d'un type à un autre. Il y a plusieurs modèles mathématiques possibles pour représenter les fonctions et les types considérés d'un langage fonctionnel. La correspondance la plus naturelle est de considérer les types représentés mathématiquement par certains ensembles et les fonctions du langage modélisées par des fonctions mathématiques ordinaires entre ces ensembles.

Pour les besoins de la cause, nous supposerons que nous avons en main un langage fonctionnel typé. Dans notre langage, pour chaque type α, nous supposerons qu'il existe une fonction $id_\alpha : \alpha \to \alpha$. De plus, si $f : \alpha \to \beta$ et $g : \beta \to \gamma$ existent, il est toujours possible de construire une fonction $g \circ f : \alpha \to \gamma$ qui est la composition des fonctions f par g. Les lois familières sur la composition sont respectées : $(h \circ g) \circ f = h \circ (g \circ f)$. Les fonctions identité id_α et id_β sont les identités gauche et droite de la composition avec $f : \alpha \to \beta : f \circ id_\alpha = f = id_\beta \circ f$.

Dans les termes de la théorie des catégories, nous dirons que si les types et fonctions d'un langage respectent les propriétés précédentes, alors ils forment une **catégorie**. Une catégorie est donc une sorte de graphe orienté [Bou93] dans lequel il y a des noeuds appelé **objets** et des arcs appelés **flèches** ou **morphismes**.

Pour la compréhension de notre exemple, il faut nous rappeler quelques fonctions d'ordre supérieur de base de la programmation fonctionnelle typée.

A chaque type α, nous pouvons construire le type $\alpha*$ où les éléments sont des *listes* d'éléments de type α. De plus, si $f : \alpha \rightarrow \beta$, nous pouvons former la fonction $f* : \alpha* \rightarrow \beta*$ qui applique la fonction f à chaque élément de la liste de type $\alpha*$. La fonction $*$ correspond précisément à la fonction `map` de Scheme. Remarquons que la notation $*$ est utilisée autant pour la construction de types que pour les fonctions. Ce "polymorphisme" peut-être malvenu pour l'informaticien est nécessaire pour mettre en évidence des propriétés fondamentales sur les listes.

$$
\begin{aligned}
id_\alpha * &= id_{\alpha *} \\
(g \circ f)* &= g * \circ f*
\end{aligned}
$$

La fonction $*$ associe les fonctions identité aux fonctions identité et la composition de fonctions au résultat de la composition de leurs images sous $*$. Dans le langage des catégories, on dit que $*$ est un **foncteur** (de types à types).

Considérons maintenant la fonction $+\!\!+/_\alpha : \alpha ** \rightarrow \alpha*$. Cette fonction prend une liste de listes et forme une liste en concaténant tous les membres de son argument. Cette fonction pourrait correspondre à une fonction de la forme `(fold concat)` écrite en Scheme où `fold` est la fonction habituelle du même nom supposant la loi d'associativité respectée pour `concat`. Nous pouvons facilement voir par simple construction de diagrammes que la fonction $+\!\!+/ : ** \stackrel{\bullet}{\rightarrow} *$ (fonction plus générale où le type (des "étoiles") n'est pas encore spécifiée) forme, dans la terminologie des catégoristes, une **transformation naturelle**. Le point qui chapeaute la flèche indique que nous avons affaire à une transformation naturelle. Reconnaître cette fonction comme une transformation naturelle nous permettra de construire notre adjonction. La reconnaissance de concepts de base est la première étape de cette théorie. C'est un peu comme en programmation fonctionnelle où nous déterminons que certaines fonctions n'ont pas à être écrites, car elles sont déjà définies dans le système. Pour bien comprendre ce concept, montrons que la fonction $+\!\!+/$ est effectivement une transformation naturelle. Pour ce faire, prenons les fonctions :

$$f* : \alpha* \rightarrow \beta*$$

$$f** : \alpha** \rightarrow \beta**$$

De celles-ci, nous pouvons tirer les fonctions :

$$+\!\!+/_\alpha : \alpha** \rightarrow \alpha*$$

$$++/_\beta \ : \ \beta ** \to \beta *$$

Ces quatres fonctions peuvent être facilement mises dans un diagramme comme suit :

$$\begin{array}{ccc} \alpha ** & \xrightarrow{f**} & \beta ** \\ {\scriptstyle ++/_\alpha}\downarrow & & \downarrow{\scriptstyle ++/_\beta} \\ \alpha * & \xrightarrow[f*]{} & \beta * \end{array}$$

Si nous composons le morphisme du bas avec le morphisme du côté gauche, nous obtenons le même résultat que si nous composons le morphisme de droite avec celui du haut du diagramme précédent. En d'autres mots, si nous suivons les chemins de $\alpha**$ jusqu'à $\beta*$ en prenant note des morphismes, nous obtenons l'équation algébrique suivante.

$$f* \ \circ \ ++/_\alpha = ++/_\beta \ \circ \ f**$$

Nous dirons alors, dans les termes des catégoristes, que le diagramme carré précédent **commute**. Si nous généralisons un peu, nous voyons que ce diagramme est un cas particulier du diagramme commutatif suivant :

$$\begin{array}{ccc} G\alpha & \xrightarrow{Gf} & G\beta \\ {\scriptstyle \theta_\alpha}\downarrow & & \downarrow{\scriptstyle \theta_\beta} \\ H\alpha & \xrightarrow[Hf]{} & H\beta \end{array}$$

Le coin gauche en haut du diagramme est obtenu en appliquant le foncteur G sur le type α (dans notre exemple, G est le foncteur[23] $**$). Dans le coin droit en haut, G est appliqué sur le type β. Cela forme la fonction (le morphisme) $Gf \ : \ G\alpha \to G\beta$.

La partie du bas du diagramme diffère du haut seulement sur le fait que cette fois-ci, le foncteur H est appliqué (dans notre cas, le foncteur $*$). Le haut et le bas du diagramme sont connectés par les fonctions (morphismes) $\theta_\alpha \ : \ G\alpha \to H\alpha$ (dans notre exemple, $++/_\alpha \ : \ \alpha ** \to \alpha *$) et $\theta_\beta \ : \ G\beta \to H\beta$ (dans notre exemple, $++/_\beta \ : \ \beta ** \to \beta *$). Nous disons que le diagramme commute si $Hf \circ \theta_\alpha = \theta_\beta \circ Gf$.

[23]La composition de foncteurs retourne toujours un foncteur.

Si cette dernière équation est vraie pour toute fonction (morphisme) $f : \alpha \rightarrow \beta$, alors nous dirons que la famille de morphismes θ est une **transformation naturelle** et nous écrirons $\theta : G \stackrel{\bullet}{\rightarrow} H$ (dans notre cas, $+\!\!+/ : ** \stackrel{\bullet}{\rightarrow} *$).

Nous avons aussi besoin de montrer que la fonction $[\cdot]$ est une transformation naturelle. La fonction $[\cdot]$ a pour effet de construire une liste singleton $[x] : \alpha*$ à partir d'un élément $x : \alpha$. Pour mettre en évidence la transformation naturelle, il est préférable de construire le diagramme commutatif correspondant à partir du diagramme général précédent. Prenons $G\alpha = \alpha$, $H\alpha = \alpha*$ et $\theta = [\cdot]$ dans le diagramme général précédent. Nous obtenons :

$$\begin{array}{ccc} \alpha & \xrightarrow{f} & \beta \\ {\scriptstyle [\cdot]_\alpha}\downarrow & & \downarrow{\scriptstyle [\cdot]_\beta} \\ \alpha * & \xrightarrow[f*]{} & \beta* \end{array}$$

Nous tirons de ce diagramme l'équation algébrique évidente suivante :

$$f * \circ [\cdot]_\alpha = [\cdot]_\beta \circ f$$

Cette équation est triviale car en appliquant x aux deux côtés de l'équation, on obtient $f*[x] = [fx]$. Donc, nous avons vérifié que $[\cdot] : Id \stackrel{\bullet}{\rightarrow} *$ est une transformation naturelle. Ici, Id représente le foncteur identité (qui n'a aucun effet sur les objets et les morphismes).

La fonction $+\!\!+$ peut aussi être vue comme une transformation naturelle. Nous retrouvons une preuve de cette relation dans un article de Spivey [Spi89].

Nous avons besoin d'une dernière transformation naturelle. C'est la fonction $[\,]^\circ$. Nous savons que, pour un type α, la liste vide $[\,]_\alpha$ est un terme de type $\alpha*$. Comme nous l'avons vu, il est impossible de représenter un terme particulier du langage fonctionnel sous forme catégorique. En effet, conformément à notre correspondance types/objets et fonctions/morphismes, un terme est *élément* d'un type particulier et les éléments des types n'ont pas de représentation catégorique. Pour avoir un représentant catégorique, nous devons construire un type ou une fonction qui représentera le terme dans une catégorie. Ce n'est pas tous les termes qui ont le privilège d'être représentés catégoriquement. La liste vide est un terme qui

a ce privilège. En effet, nous pouvons construire une fonction "liste vide" qui correspond à une transformation naturelle ! Nous la nommons $[\,]^\circ$. Ce processus auquel nous associons à certains termes des entités catégoriques constitue une étape importante du processus d'abstraction que la théorie des catégories enseigne. Pour plus de détails sur cette transformation naturelle, il faut consulter aussi l'article de Spivey [Spi89].

C Monades et triplets de Kleisli

Les monades ont été inventées dans les années cinquantes (sous le nom de *constructions standards*) et le concept est devenu populaire dans les années soixantes. Ce concept a une grande importance, car il est reconnu qu'une bonne part de l'algèbre universelle peut être reformulée sous forme de monades [Man76]. Voici sa définition catégorique.

C.1 Définition.
Une **monade** *sur une catégorie* $\mathcal{A}$ *est un triplet* (T, η, μ) *où* T *est un foncteur de* $\mathcal{A}$ *vers* $\mathcal{A}$*, l'***unité** $\eta : \mathtt{Id}_{\mathcal{A}} \stackrel{\bullet}{\to} T$ *et le* **multiplicateur** $\mu : T^2 \stackrel{\bullet}{\to} T$ *sont des transformations naturelles satisfaisant les diagrammes commutatifs suivants*[24] *:*

$$\begin{array}{ccccc} T & \xrightarrow{T\eta} & T^2 & \xleftarrow{\eta T} & T \\ & {\scriptstyle =}\searrow & \downarrow{\scriptstyle \mu} & \swarrow{\scriptstyle =} & \\ & & T & & \end{array}$$

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

□

Cette définition se traduit directement en la définition de la monade, généralisation de la liste (définition 3.1). En effet, le premier diagramme décrit précisément les règles *unité gauche* et *unité droite*. Le deuxième décrit

[24] Les notions de foncteur et transformation naturelle se trouve en annexe B

la règle d'associativité. les transformations naturelles ηT, $T\eta$ et μ correspondent aux fonctions *unit*, (*map unit*) et *join* respectivement dans les lois *unité gauche*, *unité droite* et *associativité* décrites dans la définition 3.1.

La deuxième définition de la monade computationnelle (définition 3.2) vient plutôt de la notion de triplet de Kleisli que Moggi s'est servi pour définir son méta-langage computationnel (section 5.2).

C.2 Définition.

Un **triplet de Kleisli** *sur une catégorie A est un triplet $(T, \eta, _^*)$ où l'opérateur η est appelé* **inclusion** *et l'opérateur $_^*$,* **extension**. *η est de type $\mathtt{Id} \stackrel{\bullet}{\rightarrow} T$ (même opérateur que pour la monade) et, pour chaque morphisme $f : A \rightarrow TB$, $f^* : TA \rightarrow TB$ est le morphisme auquel le domaine A a été relevé au domaine TA. Ces opérateurs doivent respecter les équations* **unité gauche**, **unité droite** *et* **associativité** *suivante :*

$$\eta^* = \mathtt{Id} \tag{1}$$

$$f^* \circ \eta = f \tag{2}$$

$$(f^* \circ g)^* = (f^* \circ g^*) \tag{3}$$

□

Conformément à la deuxième définition de la monade computationnelle (définition 3.2), ici, η est représentée par notre fonction **return** et si nous avons le terme f **then** g (de notre monade computationnel) alors il correspond au terme catégorique $g^* \circ f$.

Comme nous le voyons, les deux définitions précédentes se ressemblent beaucoup, car toutes deux contiennent 3 équations algébriques similaires. En fait, le triplet de Kleisli peut être vu comme une présentation syntaxiquement différente d'une monade puisqu'il existe une correspondance biunivoque entre le triplet et la monade [Lan71]. Le triplet de Kleisli est une présentation qui s'avère très utile pour la réflexion car, comme nous l'expliquons en section 4, l'opérateur d'inclusion s'apparente à l'opérateur de réification.

D Monoïdes

Un **monoïde** est un ensemble muni d'une opération binaire (un semi-groupe) et d'élément neutre représenté par un triplet $(\alpha, \oplus, e)$ où α est l'ensemble, $\oplus$, l'opérateur binaire et e, l'élément neutre.

Un **homomorphisme** $h : \alpha \to \beta$ de monoïdes sont des fonctions qui préservent la structure de monoïdes entre deux monoïdes $(\alpha, \oplus, e)$ et $(\beta, \otimes, d)$ si $he = d$ et $h(x \oplus y) = (hx) \otimes (hy)$. Les équations précédentes sont les **propriétés homomorphiques**. On écrira alors $h : (\alpha, \oplus, e) \to (\beta, \otimes, d)$.

Une monade ressemble beaucoup à un monoïde [RB85]. En effet, il est possible de voir un monoïde comme une monade et vice et versa. Par exemple, la monade liste $M_l = (*, map_l, unit_l, join_l)$ correspond au monoïde liste $(*, +\!\!+, [\,])$ (où le type n'est pas spécifié).

Références

[AL91] Asperti (Andrea) et Longo (Guiseppe). – *Categories, Types, and Structures : An Introduction to Category Theory for the Working Computer Scientist.* – MIT Press, 1991.

[AMY93] Asai (K.), Matsuoka (S.) et Yonezawa (A.). – Duplication and Partial Evaluation to Implement Reflective Languages. *In : Informal Proceedings of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93.*

[BH94] Bellegarde (Francoise) et Hook (James). – *Monads, Indexes, and Transformations.* – Rapport technique nN^o CS-94-008, Oregon Graduate Institute CS, 94.

[Bir87] Bird (R. S.). – An introduction to the theory of lists. *In : Logic of Programming and Calculi of Discrete Design*, éd. par Broy (M.), pp. 3–42. – Springer-Verlag, 1987.

[Bou93] Boucher (Dominique). – *Théorie des catégories en informatique : notions de base et application.* – Thèse, Université de Montréal, 1993.

[BW85] Barr (M.) et Wells (C.). – *Toposes, Triples and Theories.* – Berlin, Springer-Verlag, 1985.

[BW90] Barr (Michael) et Wells (Charles). – *Category Theory for Computing Science.* – London, Prentice-Hall International, 1990, xv+432p.

[BW96] Benton (Nick) et Wadler (Philip). – Linear logic, monads, and the lambda calculus. *In : 11'th IEEE Symposium on Logic in Computer Science.* – New Brunswick, New Jersey, Juillet 1996.

[C +86] Constable (R. L.) et al. – *Implementing Mathematics with the Nuprl Proof Development System.* – Prentice-Hall, 1986.

[Car91] Cardelli (Luca). – Typeful programming. *In : Formal Description of Programming Concepts*, éd. par Neuhold (E. J.) et Paul (M.). – Springer-Verlag, 1991.

[CCM85] Cousineau (G.), Curien (P. L.) et Mauny (M.). – The categorical abstract machine. *In : Functional Programming Languages and Computer Architecture*, éd. par Jouannaud (J.-P.), pp. 50–64. – Berlin, DE, Springer-Verlag, 1985.

[CDL95] Consel (Charles), Danvy (Olivier) et Lee (Peter). – *CMU Summer School on Partial Evaluation.* – Pittsburgh, PA 15213-3890, Carnegie Mellon University, 5–9 juillet 1995.

[CF94] Cartwright (R.) et Felleisen (M.). – Extensible Denotational language specifications. *In : Proceedings of TACS'94.* pp. 244–272. – Springer-Verlag, LNSC.

[CM96] Clavel (Manuel G.) et Meseguer (José). – Axiomatizing reflective logics and languages. *In : Proceedings of Reflection'96.*

[Dem94] Demers (F.-N.). – *Réflexion de comportement et évaluation partielle en Prolog.* – Thèse, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1994. Rapport technique #956.

[DF90] Danvy (Olivier) et Filinski (Andrzej). – Abstracting control. *In : 1990 ACM Conference on Lisp and Functional Programming.* ACM, pp. 151–160. – ACM Press.

[DF92] Danvy (Olivier) et Filinski (Andrzej). – Representing control : A study of the CPS transformation. *Mathematical Structures in Computer Science*, vol. 2, nN° 4, 1992, pp. 361–391.

[DFH +93] Dowek (Gilles), Felty (Amy), Herbelin (Hugo), Huet (Gérard), Murthy (Chet), Parent (Catherine), Paulin-Mohring (Christine) et Werner (Benjamin). – *The Coq Proof Assistant User's Guide.* – Rapport Techniques nN° 154, Rocquencourt, France, INRIA, 1993. Version 5.8.

[DKM91] Danvy (O.), Koslowski (J.) et Malmkjær (K.). – *Compiling Monads.* – Technical Report nN° CIS-92-3, Manhattan, Kansas, Kansas State University, 1991.

[DM88] Danvy (O.) et Malmkjaer (K.). – Intensions and Extensions in a Reflective Tower. *In : Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pp. 327–341.

[DM95] Demers (F.-N.) et Malenfant (J.). – Reflection in logic, functional and object-oriented programming : a short comparative study. *In : Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pp. 29–38.

[dRS84] des Rivières (J.) et Smith (B. C.). – The implementation of procedurally reflective languages. *In : Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 331–347.

[Esp94] Espinosa (David). – *Building Interpreters by transforming stratified monads.* – Rapport technique, New York, NY 10027, Columbia University, Departement of Computer Science, juin 1994. espinosacs.columbia.edu.

[Esp95] Espinosa (David A.). – *"Semantic Lego".* – Thèse de PhD, Columbia University, 1995.

[Fil94] Filinski (Andrzej). – Representing monads. *In : Conference Record of the Twenty-First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

[FM94] Fokkinga (M. M.) et Meertens (L.). – *Adjunctions.* – Memoranda informatica, University of Twente, Juin 1994.

[Fok92a] Fokkinga (M. M.). – Calculate categorically ! *Formal Aspects of Computing*, vol. 4, 1992, pp. 673–692.

[Fok92b] Fokkinga (M. M.). – *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, chap. A Gentle Introduction to Category Theory — the calculational approach. – University of Utrecht, 1992.

[Fok94] Fokkinga (M. M.). – *Monadic Maps and Folds for Arbitrary Datatypes.* – Memoranda Informatica nN° 94-28, University of Twente, Juin 1994.

[GB84a] Goguen (J. A.) et Burstall (R. M.). – Some fundamental algebraic tools for the semantics of computation (i). *Theoretical Computer Science*, vol. 31, 1984, pp. 175–209.

[GB84b] Goguen (J. A.) et Burstall (R. M.). – Some fundamental algebraic tools for the semantics of computation (ii). *Theoretical Computer Science*, vol. 31, 1984, pp. 263–295.

[Gir87] Girard (Jean-Yves). – Linear logic. *Theoretical Computer Science*, vol. 50, 1987, pp. 1–102.

[GM87] Goguen (J. A.) et Meseguer (J.). – Unifying functional, object-oriented and relational programming with logical semantics. *In : Research Directions in Object-Oriented Programming*, éd. par Shriver (B.) et Wegner (P.). pp. 417–477. – Cambridge, MA, 1987.

[Gog91] Goguen (Joseph). – A categorical manifesto. *Mathematical Structures in Computer Science*, vol. 1, nN° 1, Mars 1991, pp. 49–67.

[GS89] Gray (John W.) et Scedrov (Andre) (édité par). – *Categories in Computer Science and Logic.* – Providence, Rhode Island, American Mathematical Society, 1989, *Contemporary Mathematics*, volume 92.

[Has95] Hasegawa (M.). – Decomposing typed lambda calculus into a couple of categorical programming languages. *In : Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS'95)*, éd. par Pitt (D.), Rydeheard (D. E.) et Johnstone (P.). pp. 200–219. – LNCS 953. Springer.

[HC94] Hill (Jonathan M. D.) et Clarke (Keith). – *An introduction to category theory, category theory monads, and their relationship to functional programming.* – Rapport technique nN° QMW-DCS-681, Department of Computer Science, Queen Mary and Westfield College, Aug 94.

[HD94] Hatcliff (John) et Danvy (Olivier). – A generic account of continuation-passing styles. *In : Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*. pp. 458–471. – Portland, Oregon, janvier 17–21, 1994.

[HM96] Hutton (Graham) et Meijer (Erik). – Monadic parser combinators. – 1996. Soumis pour publication.

[Hoa88a] Hoare (C. A. R.). – Approach to category theory for computer scientists. – 1988. Lecture Notes, International Summer School on Constructive Methods in Computing Science, Marktoberdorf 1988.

[Hoa88b] Hoare (C. A. R.). – Notes on an approach to category theory for computer scientists. – 1988. International Summer School on Constructive Methods in Computing Science, Marktoberdorf.

[How80] Howard (W.). – The formulas-as-types notion of construction. *In : To H.B. Curry : Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pp. 479–490. – Academic Press, 1980.

[Hue86] Huet (G. P.). – Cartesian closed categories and lambda-calculus. *In : Combinators and Functional Programming Languages*, éd. par Cousineau (G.), Curien (P.-L.) et Robinet (B.), pp. 123–135. – Berlin, DE, Springer-Verlag, 1986. Lecture Notes in Computer Science 242.

[Hug95] Hughes (John). – The design of a pretty-printing library. *In : Advanced Functional Programming*, éd. par Jeuring (J.) et Meijer (E.). – Springer Verlag, 1995.

[Jac94] Jacques (M.). – *Implantation d'un langage à prototypes avec réflexion de comportement.* – Thèse, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1994. Rapport technique #955.

[Jac96] Jacques (Marco). – *Compilation de la réflexion de comportement statique dans un langage à objets, Rapport pré-doctoral.* – Rapport technique, Université de Montréal, 1996.

[JD93] Jones (Mark P.) et Duponcheel (Luc). – *Composing Monads.* – Rapport technique nN° YALEU/DCS/RR-1004, Department of Computer Science, Yale University, 1993.

[JF92] Jefferson (S.) et Friedman (D.P.). – A Simple Reflective Interpreter. *In : Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*, éd. par Yonezawa (A.) et Smith (B.). RISE (Japan), ACM Sigplan, JSSST, IPSJ, pp. 48–58.

[JHHP93] Jones (S. L. P.), Hall (C.), Hammond (K.) et Partain (W.). – The glasgow haskell compiler : a technical overview. *In : Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference.* – Keele, 1993.

[Jon96] Jones (Simon L. Peyton). – Compiling haskell by program transformation : a report from the trenches. *In : Proceedings of the European Symposium on Programming.* – Linkoping, avril 1996.

[Jon94] Jones (Mark P.). – *The implementation of the Gofer functional programming system.* – Rapport technique nN°

YALEU/DCS/RR-1030, New Haven, Connecticut, USA, Department of Computer Science, Yale University, mai 94.

[KAH92] Kieburtz (R.), Agapiev (B.) et Hook (J.). – Three monads for continuations. *In : ACM SIGPLAN Workshop on Continuations.* – Stanford University, juin 1992. Report STAN-CS-92-1426.

[KHC91] Kishon (A.), Hudak (P.) et Consel (C.). – Monitoring Semantics : A Formal Framework for Specifying, Implementing and Reasoning about Execution Monitors. *Proceedings of PLDI'91, ACM Sigplan Notices*, vol. 26, nN° 6, juin 1991, pp. 338–352.

[Kre68] Kreisel (G.). – Reflection principles and their use for establishing the complexity of axiomatic systems. *In : Zeitschrift Fur Mathematische Logik und Grundlagen der Mathematik*, pp. 97–142. – 1968.

[KW92] King (D.) et Wadler (P.). – Combining monads. *In : Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland.* pp. 134–143. – Springer Verlag.

[Lan71] Lane (Saunders Mac). – *Categories for the Working Mathematician.* – New York, Springer-Verlag, 1971, *Graduate Texts in Mathematics*, volume 5, ix+262p.

[LH96] Liang (Sheng) et Hudak (Paul). – Modular denotational semantics for compiler construction. *In : European Symposium on Programming.* – Linkoping, Sweden, avril 1996. Disponible par ftp : nebula.cs.yale.edu :/pub/yale-fp/papers/mod-sem-draft.ps.Z.

[LHJ95] Liang (Sheng), Hudak (Paul) et Jones (Mark). – Monad transformers and modular interpreters. *In : Conference Record of POPL '95 : 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 333–343. – San Francisco, California, 1995.

[LS86] Lambek (J.) et Scott (P. J.). – *Introduction to Higher Order Categorical Logic.* – CUP, 1986, *Cambridge studies in advanced mathematics.*

[LS91] Lawvere (F. William) et Schanuel (Stephen H.). – *Conceptual Mathematics : A first Introduction to Categories.* – Buffalo Workshop Press, 1991.

[Man76] Manes (E.). – *Algebraic Theories.* – Springer-Verlag, 1976, *Graduate Texts in Mathematics*, volume 26.

[MDC92] Malenfant (J.), Dony (C.) et Cointe (P.). – Behavioral Reflection in a Prototype-Based Language. *In : Proceedings of the International Workshop on New Models for Software Architectures, Reflection and Metalevel Architectures, Tokyo, Japan*, pp. 143–153.

[MDC96] Malenfant (J.), Dony (C.) et Cointe (P.). – A semantics of introspection in a reflective prototype-based language. *In : Lisp and Symbolic Computation*, éd. par Publishers (Kluwer Academic). – Boston, 1996.

[MF93] Mendhekar (A.) et Friedman (D.P.). – Towards a Theory of Reflective Programming Languages. *In : Informal Proceedings of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93.*

[MF96] Mendhekar (A.) et Friedman (D. P.). – An exploration of relationships between reflective theories. *In : Proceedings of Reflection'96.*

[MJD96] Malenfant (J.), Jacques (M.) et Demers (F.-N.). – A tutorial on beharioral reflection and its implementation. *In : Proceedings of Reflection'96.*

[Mog89a] Moggi (E.). – *An Abstract View of Programming Languages.* – Rapport technique nN° ECS-LFCS-90-113, Edinburgh Univ., Dept. of Comp. Sci., 1989. Notes pour le cours CS 359, Stanford Univ.

[Mog89b] Moggi (E.). – Computational lambda-calculus and monads. *In : 4th LICS Conference.* IEEE.

[Mog91] Moggi (E.). – Notions of computation and monads. *Information and Computation*, vol. 93, nN° 1, 1991.

[Mos92] Mosses (P.D.). – *Action Semantics.* – Cambridge University Press, 1992, *Cambridge Tracts in Theoretical Computer Science*, volume 26.

[MT93] M.J.C. Gordon et T.F. Melham. – *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic.* – Cambridge University Press, 1993.

[Pie90] Pierce (Benjamin C.). – *A Taste of Category Theory for Computer Scientists.* – Report nN° CMU–CS–90–113, Carnegie Mellon University, Mars 6 1990.

[Pie91] Pierce (Benjamin C.). – *Basic Category Theory for Computer Scientists.* – The MIT Press, 1991, *Foundations of Computing.*

[PJW93] Peyton Jones (Simon L.) et Wadler (Philip). – Imperative functional programming. *In : Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* pp. 71–84. – Charleston, South Carolina, janvier 10–13, 1993.

[Plo75] Plotkin (Gordon D.). – Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, vol. 1, 1975, pp. 125–159.

[Poi92] Poigne. – Basic category theory. *In : Handbook of Logic in Computer Science, Volumes 1 (Background : Mathematical Structures) and 2 (Background : Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon.* – 1992.

[Pol94] Pollack (Robert). – *The Theory of LEGO : A Proof Checker for the Extended Calculus of Constructions.* – Thèse de PhD, University of Edinburgh, 1994.

[RB85] Rydeheard (D.E.) et Burstall (R.M.). – *Monads and theories : a survey for computation*, chap. 16, pp. 575–605. – Cambridge, Cambridge Universy Press, 1985volume Algebraic methods in semantics.

[RB88] Rydeheard (D. E.) et Burstall (R. M.). – *Computational Category Theory.* – Hertfordshire, Prentice Hall, 1988.

[Rey72] Reynolds (John C.). – Definitional interpreters for higher-order programming languages. *In : Proceedings ACM National Conference*, pp. 717–740.

[Rey80] Reynolds (J. C.). – Using category theory to design implicit conversions and generic operators. *In : Semantics-Directed Compiler Generation*, éd. par Jones (N. D.), pp. 211–2580. – Berlin, Springer LNCS 94, 1980.

[Rud92] Rudnicki (Piotr). – An overview of the Mizar project. – Juin 1992.

[SF92] Sabry (Amr) et Felleisen (Matthias). – Reasoning about programs in continuation-passing style. *In : Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 288–298. – San Francisco, USA, Juin 1992.

[SF93] Sabry (A.) et Field (J.). – Reasoning about explicit and implicit representations of state. *In : Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '93*, pp. 17–30.

[Smi82] Smith (B.C.). – *Reflection and Semantics in a Procedural Language.* – Rapport technique nN° 272, MIT Laboratory for Computer Science, 1982.

[Smi84] Smith (B.C.). – Reflection and Semantics in Lisp. *In : Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 23–35.

[Spi89] Spivey (M.). – A categorical approach to the theory of lists. *In : Conference on the Mathematics of Program Construction*, éd. par van de Snepscheut (J. L. A.). pp. 399–408. – Springer-Verlag, LNCS 375.

[Spi90] Spivey (Mike). – A functional theory of exceptions. *Science of Computer Programming*, vol. 14, nN° 1, Juin 1990, pp. 25–42.

[Spi93] Spivey (Mike). – *Category Theory for Functional Programming.* – Rapport technique nN° PRG-TR-7-93, PRG, Oxford, 1993.

[Ste94] Steele, Jr. (Guy L.). – Building interpreters by composing monads. *In : Conference Record of POPL '94 : 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 472–492. – Portland, Oregon, 1994.

[SY74] Solntseff (N.) et Yezerski (A.). – A survey of extensible languages. *Annual Review in automatic Programming*, vol. 7, nN° 4, 1974, pp. 267–307.

[Tar44] Tarski (Alfred). – The semantic conception of truth. *Philos. Phenomenological Research*, vol. 4, 1944, pp. 13–47.

[TBG91] Tarlecki, Burstall et Goguen. – Some fundamental algebraic tools for the semantics of computation : Part 3. indexed categories. *Theoretical Computer Science*, vol. 91, 1991.

[Ten91] Tennent (Robert D.). – *Semantics of Programming Languages.* – New York, Prentice Hall, 1991, xxi+236p.

[vO95] van Oosten (Jaap). – *Basic Category Theory.* – Rapport technique nN° ISSN 1395-2048, BRICS Lecture Series, Dept. of C.S. University of Ashus, 1995.

[Wad90] Wadler (P.). – Comprehending monads. *In : ACM Conference on Lisp and Functional Programming.*

[Wad92] Wadler (P. L.). – The essence of functional programming. *In : Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM*, pp. 1–14.

[Wad94] Wadler (Philip). – Monads and composable continuations. *Lisp and Symbolic Computation*, vol. 7, nN° 1, 1994, pp. 39–55.

[Wad95] Wadler (Philip). – How to declare an imperative. *In : ILPS'95 : International Logic Programming Symposium*, éd. par Lloyd (John). – MIT Press.

[Wad92] Wadler (Philip). – Monads for functional programming. – Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag, Août 92.

[Wat95] Watanabe (Takuo). – Towards a foundation of computational reflection based on abstract rewriting. *In : IMSA'95*. Information-Technology Promotion Agency, pp. 143–145. – Japan, 1995.

[WF88] Wand (M.) et Friedman (D. P.). – The Mystery of the Tower Revealed : A Nonreflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, vol. 1, nN° 1, 1988, pp. 11–37.