Université de Montréal

RÉFLEXION DE COMPORTEMENT ET ÉVALUATION PARTIELLE EN PROLOG

par

François-Nicola Demers

Département d'informatique
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril 1994

© François-Nicola Demers, MCMXCIV

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé

RÉFLEXION DE COMPORTEMENT ET ÉVALUATION PARTIELLE EN PROLOG

présenté par

François-Nicola Demers

a été évalué par un jury composé des personnes suivantes :

Rachida Dssouli

(président-rapporteur)

Jacques Malenfant

(directeur de recherche)

Michel Boyer

(membre du jury)

Mémoire accepté le :
26 août 1994

SOMMAIRE

Dans ce mémoire, il est question de deux notions centrales: d'abord, la réflexion de comportement et ensuite, l'évaluation partielle.

La réflexion de comportement est un paradigme de programmation qui tente de rendre un langage plus flexible à de nouvelles applications. Nous nous intéressons exclusivement à l'implémentation de la réflexion par méta-interprétation à *la 3-Lisp*. Nous avons restreint notre étude à un sous-ensemble bien défini (presque complètement déclaratif) du langage Prolog. Nous avons conçu quatre langages réflexifs dont chacun rend accessible une entité différente du langage de base. Par ces langages, nous avons démontré la pertinence et la puissance de la réflexion à *la 3-Lisp* en programmation logique dans un but d'ouverture et de simplification de la sémantique du langage.

Concernant l'implémentation de ces langages, nous avons utilisé deux représentations reconnues en programmation logique: la représentation non-close et la représentation close ("ground"). La représentation non-close, traditionnellement utilisée par les programmeurs Prolog est naturelle et simple. Nous l'avons utilisée pour l'implémentation de deux de nos langages réflexifs. Par ailleurs, la représentation close a été reconnue plus puissante que la représentation non-close mais plus complexe. Les deux autres langages ont été implémentés dans cette représentation.

L'implémentation de la réflexion par méta-interprétation est très inefficace que ce soit en représentation close ou non-close. Pour tenter de pallier à ce problème, nous avons appliqué une technique d'optimisation de programmes basée sur la sémantique appelée évaluation partielle. Cette technique a eu particulièrement du succès sur des méta-interprètes. Pour la représentation non-close, l'évaluation partielle a réussi à optimiser très efficacement et, dans bien des cas, à éliminer complètement la couche de méta-interprétation de l'exécution de nos programmes

réflexifs. Par contre, concernant la représentation close, il s'avère que notre algorithme de résolution est inadéquat pour l'obtention d'une bonne évaluation partielle. En effet, dans ce cas, les résultats d'optimisation de nos deux autres méta-interprètes sont faibles ou clairement insuffisants.

Enfin, nous avons pu établir que la représentation close est indispensable pour une bonne implémentation de la réflexion en programmation logique dans le but d'une optimisation par évaluation partielle. De plus, pour que celle-ci soit capable d'optimiser efficacement les méta-interprètes en cette représentation, il est nécessaire de concevoir un algorithme de résolution beaucoup mieux adapté pour l'application de l'évaluation partielle.

REMERCIEMENTS

J'aimerais remercier quelques personnes qui m'ont aidé ou supporté directement ou indirectement au cours de ma maîtrise et au cours de la rédaction de mon mémoire.

D'abord, il est indiscutable que mon directeur de recherche, Jacques Malenfant, a contribué directement à la réalisation de cette maîtrise. Je le remercie donc grandement pour l'aide et le soutien tout au long des deux années de recherche accomplie.

Je remercie Dan Sahlin, concepteur de l'évaluateur partiel Mixtus. Il a eu la gentillesse de corriger certaines erreurs d'implémentation de son système à ma demande.

Je remercie Corin Gurr qui vient de terminer sa thèse de doctorat sur l'évaluation partielle méta-circulaire dans le langage Gödel. Ce chercheur, par l'entremise de quelques discussions électroniques, a aidé au développement de mes recherches, aux conclusions tirées de mes résultats et à la compréhension des tendances actuelles en évaluation partielle. Sa thèse en elle-même a été une grande source d'inspiration.

Je remercie Nicolas Anquetil, le co-administrateur système, pour son aide et sa patience sans limite à répondre à mes requêtes même les plus loufoques.

Je remercie Michel Gagnon et Stéphane S. Somé, amis et collègues de travail, entre autres, de m'avoir écouté expliquer en long et en large mes théories sur l'évaluation partielle quand nous étions à souffrir sous la pluie en plein bois.

Je remercie enfin tous les autres membres du laboratoire Incognito.

TABLE DES MATIÈRES

LISTE DES FIGURES

CHAPITRE 1

Introduction et motivation

La réflexion est un paradigme de programmation récent en informatique. Elle offre une manière élégante d'augmenter la puissance d'un langage en représentant dans le langage lui-même les programmes ainsi que l'état de leur exécution en tant qu'entités de plein droit, c'est-à-dire manipulables au même titre que les autres valeurs. Dans le cas de la promotion de l'état d'exécution en tant qu'entité de plein droit, on parle plus spécifiquement de réflexion de comportement [Mae87].

Nous nous intéressons particulièrement à l'implémentation *à la 3-Lisp* [Smi84] de la réflexion de comportement. Cela consiste en l'utilisation d'un empilement de méta-interprètes (appelé tour réflexive) tous s'exécutant en même temps. Ces méta-interprètes doivent respecter certaines conditions. Par exemple, ils doivent être écrits dans le langage qu'ils interprètent. C'est la condition de méta-circularité. Dans le cas de Prolog, pour respecter cette condition, il est suggéré qu'une rationalisation du langage soit d'abord bien définie de façon à simplifier l'écriture des méta-interprètes réflexifs.

Nous allons donc caractériser un sous-ensemble du langage Prolog. Entre autres, nous exclurons les prédicats méta-logiques tels que `var/1`, `==/2`, `clause/2` et le coupe-choix n'ayant pas de sémantique déclarative. À partir de ce sous-ensemble, nous allons écrire plusieurs petits méta-interprètes réflexifs. Chaque langage (méta-interprète) rendra accessible une entité du comportement de l'exécution.

À titre d'exemple de la puissance de l'intégration de la réflexion en Prolog, il sera possible de définir déclarativement de façon réflexive des prédicats méta-logiques de manipulation des unifications tels que `var/1` et `==/2` (précédemment exclus par rationalisation). De plus, le prédicat `clause/2` d'accès à la base de règles pourra aussi être défini réflexivement dans le langage.

L'implémentation de la tour réflexive par couche de méta-interprétation coûte cher en temps d'exécution. Il est bien connu que chaque couche de méta-interprétation ralentit l'exécution d'un programme d'un ordre de grandeur. Or, en réflexion *à la 3-Lisp*, nous sommes confronté à des empilements de méta-interprètes potentiellement infinis, mais qui comporte tout de même, en pratique, au moins quelques niveaux. L'évaluation partielle, une technique de transformation de programmes qui a eu particulièrement du succès dans l'élimination de telles couches de méta-interprétation. Par contre, elle a ses limites. En plus de concevoir et d'implémenter quelques langages réflexifs *à la 3-Lisp* en Prolog, un second mandat de ce mémoire est de proposer une première étude des capacités d'optimisation de l'évaluation partielle dans le cas de méta-interprètes réflexifs.

Nous appliquerons un évaluateur partiel automatique déjà existant sur nos différents méta-interprètes réflexifs. Nous mesurerons les gains d'optimisation obtenus pour tenter de déceler les limites de l'évaluation partielle et de déterminer (s'il y a lieu) les changements d'implémentation à apporter sur nos méta-interprètes.

Ce mémoire se divise en deux grandes parties: la réflexion et l'évaluation partielle. Le chapitre 2 contient une section introductive aux définitions de base de la réflexion. Dans le chapitre 3, il est question de la description du sous-ensemble du langage Prolog auquel nous nous sommes intéressé. Dans la deuxième partie, il est question d'évaluation partielle. Le chapitre 4 contient un compte-rendu historique des recherches dans le domaine. Le chapitre 5 présente plusieurs résultats d'optimisation de l'évaluation partielle sur les langages réflexifs présentés au chapitre 3. Enfin, le chapitre 6 conclut et offre quelques perspectives de recherches.

CHAPITRE 2

Réflexion et langages réflexifs

La définition du terme réflexion dans le dictionnaire est la suivante: la réflexion est un retour opéré par la pensée sur elle-même en vue d'une conscience plus nette et d'une maîtrise plus grande de ses processus. Dans cette définition, nous retrouvons la notion fondamentale du retour sur soi-même. En informatique, nous nous intéressons particulièrement à la réflexion en rapport aux langages de programmation. Par analogie, le programme en exécution écrit dans un langage de programmation se compare à la pensée d'un homme. Dans ce cas, le retour de la pensée sur elle-même se traduit par la possibilité pour un programme en exécution de lire et de modifier l'information relative à lui-même, c'est-à-dire à sa représentation dans le langage et à son exécution. L'information dite réflexive doit être représentée à l'aide de structures de données du langage. Cette restriction oblige à ce que le langage contienne tous les outils dont un programme a besoin pour analyser son propre comportement. La réflexion a donc pour effet d'ajouter de la puissance à un langage sans modifier ce qui est déjà existant dans le langage.

En d'autres mots, la réflexion donne au langage une plus grande maîtrise de son évaluateur ou plus généralement de sa sémantique. La sémantique d'un langage se caractérise souvent par l'implémentation d'un interprète du langage. La plupart des langages ont suffisamment de puissance d'expression pour offrir les moyens d'écrire un interprète du langage dans le langage même. Ces interprètes particuliers se nomment **méta-interprètes**. L'écriture de méta-interprètes est encore aujourd'hui le moyen le plus simple et le plus rapide d'implémenter une sémantique particulière. En effet, il est possible de modifier la sémantique d'un langage en modifiant le comportement d'un méta-interprète. De ce point de vue, la méta-interprétation a pour objectif de permettre la modification de la sémantique du langage tout comme la réflexion. Le désavantage de l'approche de la méta-interprétation, c'est d'être obligés de réécrire tout un méta-interprète pour d'infimes modifications de la sémantique du langage. Comment modifier la

sémantique du langage sans devoir implémenter de façon répétitive la majorité de l'algorithme de contrôle de leur interprète? C'est une question de réutilisation.

En fait, la solution que propose la réflexion est d'ajouter des outils puissants pour accéder et modifier l'information relative à l'évaluateur du langage au cours son exécution. Le programmeur qui désire concevoir un outil de programmation qui exige une modification du comportement du langage, n'a qu'à se concentrer sur cette modification sans s'attarder aux aspects relatifs à la méta-interprétation comme, par exemple, l'analyse syntaxique. La réflexion a néanmoins ses limites: son utilisation n'est pertinente que pour des modifications ou des ajouts relativement mineurs au langage. La réflexion, dans ce sens, est un compromis pour éviter la méta-interprétation complète. Néanmoins, la réflexion ne règle pas tout. Il reste tout de même des questions de structuration et de réutilisation. C'est ce qui constitue une gamme de défis pour les chercheurs s'intéressant à la réflexion.

1. Réflexion et réification

Premièrement, nous allons introduire les concepts fondamentaux en reprenant la comparaison du processus de la réflexion de la pensée humaine avec celle des langages.

Que faisons-nous lorsque nous réfléchissons? À la base, la pensée opère au niveau du cerveau en traitant des informations recueillies à l'aide des sens et, compte tenu d'objectifs à atteindre, elle établit un plan visant à rendre la réalité extérieure conforme à ses objectifs. La pensée n'agit pas elle-même sur la réalité. Elle agit plutôt sur une représentation mentale de la réalité. Pour mettre au point un plan, la pensée étudie cette représentation mentale, la modifie successivement et teste les résultats d'une ébauche de plan en simulant ses effets sur sa représentation mentale. Ensuite, quand le plan est complet, la pensée traduit le plan en influx nerveux donnant ordre aux membres d'agir sur la réalité extérieure.

La réflexion consiste à étudier le processus de pensée en lui-même. Pour ce faire, elle utilise un autre processus de pensée que nous supposons similaire au processus de pensée de base. Comme dans le cas de la réalité, la pensée réflexive n'agit pas directement sur le processus de pensée. Elle agit sur une représentation de la pensée de base. La pensée réflexive, elle aussi, étudie et modifie sa représentation pour mettre au point un plan. Enfin, le plan terminé, la pensée réflexive doit appliquer le plan sur la pensée de base conformément aux modifications désirées. Pour que la pensée réflexive mène à des résultats applicables, il faut que la représentation du phénomène sur lequel elle veut agir soit non seulement correcte et dans la mesure du possible complète mais aussi constamment remise à jour. Un besoin se fait sentir: la connexion causale.

La **connexion causale** [Smi84] d'une réalité et de sa représentation c'est-à-dire de la description symbolique de cette même réalité est une propriété nécessaire. Cette propriété signifie que la réalité et sa représentation sont intimement liées de telle façon que si l'une des deux changent, ceci amène un changement correspondant dans l'autre. Nous définissons alors un **système réflexif** [Mae87] comme étant celui qui possède une représentation de lui-même en connexion causale avec la réalité qu'elle représente. Ce système est aussi capable de "raisonner" et agir sur cette représentation pour examiner plus à fond son état et modifier son comportement futur.

Dans les langages de programmation, la réflexion est la capacité des programmes d'observer et de raisonner à propos de leur propre état, et de modifier leur comportement d'exécution ou plus profondément, d'altérer leur propre interprétation ou leur propre signification. Ces deux aspects requièrent un mécanisme pour encoder l'état du programme comme données du langage, la réification.

On définit la **réification** [LO91] comme étant le mécanisme par lequel des informations d'un langage de programmation L qui seraient autrement invisibles parce que fixées dans son implémentation, sont représentées à l'aide de structures de données ou de procédures du langage lui-même en connexion causale avec l'exécution du programme. Ces informations rendues accessibles à un programme de l'usager peuvent ainsi être observées et manipulées dynamiquement. Nous appelons les structures de données du langage représentant l'information précédemment invisible, **l'information réifiée**.

Supposons un langage réflexif L. Le mécanisme de la réification correspond en quelque sorte à une "traduction"[1] automatique de l'information (représentée à l'extérieur du langage) à réifier vers sa représentation dans le langage. En effet, les informations au départ inaccessibles dans le langage, ont nécessairement une représentation mais dans un autre langage L' généralement celui qui a servi de support pour l'implémentation du langage L. La connexion causale s'assure que cette traduction soit continuelle c'est-à-dire que si l'information inaccessible dans le langage L change dans le temps, il faut que l'information réifiée (sa représentation dans le langage L) se conforme le plus rapidement possible à ce changement. De même, si nous modifions l'information réifiée, sa représentation extérieure (celle dans le langage L') doit s'altérer en conséquence.

Le **degré de réflexion** [Mae87] d'un langage est une mesure qualitative de l'ampleur des portions du langage rendues visibles au niveau du programme par réification. Cette mesure est importante car elle est un outil de comparaison de la puissance de réflexion de différents

[1] Ce terme ne sous-entend pas la conception d'un programme indépendant qui traduit effectivement d'une structure à l'autre. Il sous-entend plutôt un changement intrinsèque de la structure causé par le transfert d'un langage à un méta-langage.

langages. Nous verrons que plus le degré de réflexion d'un langage est élevé, plus son implémentation efficace est difficile à réaliser.

On distingue en général deux grands types de réflexion [Mae87]:

- la **réflexion structurelle** est la capacité d'un langage à réifier les programmes et la capacité à manipuler sans restrictions les structures de données du langage.
- la **réflexion comportementale** est la capacité d'un langage à réifier l'état d'exécution des programmes ainsi que l'évaluateur du langage lui-même.

Il est vrai que ces définitions semblent relativement imprécises. De ces définitions, certains pourraient argumenter que, dans un certain sens, la réflexion comportementale peut tout aussi bien être considérée de la réflexion structurelle et vice versa. En fait, le choix de la réflexion comportementale ou structurelle est qualitatif. Il se pourrait qu'un aspect de la réflexion soit structurelle et comportementale en même temps, dépendant de l'angle sous lequel on l'examine. Comprenons en tout cas que ces deux aspects de la réflexion sont dissociés parce qu'ils impliquent des efforts très différents pour être réalisés. La réflexion de structure est actuellement assez bien maîtrisée. Plusieurs langages offrent la réification des programmes, et tout un corpus de connaissances a été développé pour réaliser la réflexion de structure de façon compréhensible et efficace. La réflexion de comportement pose des problèmes énormément plus difficiles à résoudre [Mae87], en particulier parce qu'il ne suffit pas de représenter l'évaluateur du langage dans le langage lui-même mais parce qu'il faut aussi avoir la connexion causale entre cette représentation et l'état d'exécution en cours. Une certaine connexion causale est aussi nécessaire en réflexion de structure. Néanmoins, elle est beaucoup plus facile à réaliser.

En approfondissant les notions de réflexion comportementale, des questions nous viennent à l'esprit: si l'évaluateur du langage L est réifié à un certain degré de réflexion, ne faudrait-il pas réifier l'évaluateur de cet évaluateur? Et à propos de celui-là, ne faudrait-il pas le réifier lui aussi? En bout de piste, nous obtenons une liste infinie d'évaluateurs écrits dans le langage L qui sont chacun la réification de l'évaluateur précédent. Il est conceptuellement impossible d'éviter cette régression infinie. Pour ce faire, il nous faudrait un évaluateur écrit dans le langage L et, pour qu'il évalue une expression du langage, il faudrait qu'il puisse s'interpréter lui-même, en même temps qu'il évalue l'expression. C'est en fait un problème similaire au problème de l'auto-référence.

Si nous ne voulons pas limiter arbitrairement la réflexion comportementale du langage réflexif, nous devons concevoir cette réflexion le plus conformément possible à cette régression infinie de réifications. Ainsi, pour un programme P, nous définissons une **tour réflexive**[Smi84] comme étant un empilement d'interprètes où P se trouve au niveau 0 et où chaque niveau n

est formé d'un interprète I_n, lui-même interprété par l'interprète I_{n+1}. Chaque niveau de la tour réflexive respecte la connexion causale avec son niveau adjacent. La figure 2.1 aide à visualiser la chose. En pratique, une tour réflexive infinie n'est pas réalisable. Nous aurons donc à concevoir une tour réflexive finie qui pourra facilement varier en hauteur. Le dernier interprète dans la tour devra être évalué simplement par l'évaluateur de base du langage L. Pour aider à cela, il est préférable de concevoir les interprètes I_i méta-circulaires. Un **interprète méta-circulaire**[Smi84] pour un langage L est un interprète pour L écrit dans ce même langage et qui est capable de s'exécuter lui-même, comme n'importe quel autre programme de L. Le qualificatif méta-circulaire est significatif. En effet, *méta* renvoie le sens d'action sur et *circulaire* informe de la propriété de l'interprète d'effectuer un retour sur lui-même. Le mot nous informe aussi que le comportement de l'interprète (représenté explicitement) doit être connu d'avance pour comprendre les résultats de cette interprétation. La solution concernant l'implémentation de la tour réflexive est d'effectuer une **création paresseuse des niveaux**. Par exemple, pour une tour réflexive de hauteur m, il sera possible d'ajouter un étage d'interprétation au niveau $m+1$ en utilisant simplement, par exemple, l'interprète $I_{m+1} = I_m$ puisque I est méta-circulaire.

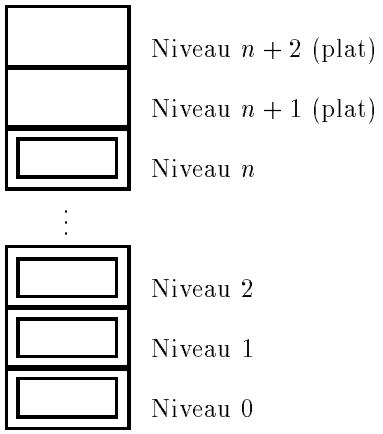

FIGURE 2.1. Tour réflexive.

Maintenant, essayons de comprendre ce qui se passe lorsqu'un programme veut accéder à son état d'exécution. Soit I, notre interprète méta-circulaire. Examinons le cas où le programme en question est un interprète I_k de la tour réflexive. Alors, si celui-ci inspecte son propre état de calcul, il inspecte conséquemment quelque chose qui doit représenter les structures de données de l'évaluateur qui l'exécute à ce moment même. Il se produit donc un **recouvrement introspectif** [Mae87] c'est-à-dire le phénomène manifesté lorsqu'une procédure ou une instruction qui inspecte l'état de calcul inspecterait en fait sa propre exécution. Pour éviter ce recouvrement

introspectif, on introduit des **procédures** dites **réflexives**[Smi84] qui, lorsqu'invoquées au niveau k, s'exécutent au niveau $k+1$. Le recouvrement instrospectif est évité car les instructions inspectant l'état de calcul sont maintenant évaluées par l'interprète I_{k+2}. Il faut absolument que les procédures réflexives soient invoqués, à l'origine, non pas par le méta-interprète mais plutôt par le programme interprété appartenant à l'usager situé au niveau 0 de la tour. Mais rien n'empêche d'appeler une deuxième procédure réflexive à l'intérieur d'une première. Et une troisième à l'intérieur de la deuxième et ainsi de suite. Il n'y a pas de limite à priori au nombre de niveaux dans la tour réflexive pour que nous puissions évaluer toute procédure réflexive quelle qu'elle soit. La création paresseuse des niveaux est donc indispensable.

Un niveau d'interprétation i dans la tour réflexive est **plat** si à aucun moment durant l'exécution du programme P, ce niveau n'exécute une procédure réflexive appelée par le niveau $i-1$. Un niveau plat est un niveau dont l'exécution est tout simplement l'interprète méta-circulaire du langage L. Notons que si le niveau $n+1$ est plat, les niveaux $n+2$, $n+3$ etc. sont nécessairement plats également car la seule façon d'exécuter une procédure réflexive au niveau $n+2$ est d'en appeler une au niveau $n+1$. Nous disons alors que le niveau n est le **niveau d'introspection** [Mae87] du programme P. À titre d'illustration, dans la figure 2.1, les niveaux plats sont les rectangles simples et les niveaux non-plats, donc qui recoivent éventuellement une procédure réflexive, sont les rectangles doubles.

Un programme réflexif P bien conçu est un programme qui a un niveau d'introspection fini pour toutes les entrées légales du programme. Nous pouvons comparer cela aux procédures récursives. Une procédure récursive bien conçue est celle qui ne va s'appeler qu'un nombre fini de fois. Ainsi, pour exécuter P, il suffit de créer uniquement n niveaux de sa tour réflexive. En pratique cependant, le niveau n dépend des entrées du programme et n'est déterminé que dynamiquement. C'est pour cette raison que nous devons être capable d'ajouter des niveaux de la tour à la volée.

Nous venons de voir les principaux concepts modernes de la réflexion. Nous discuterons maintenant des travaux sur les langages réflexifs qui ont été réalisés en programmation fonctionnelle. En effet, la programmation fonctionnelle a été le premier paradigme de programmation à étudier la réflexion comportementale telle que vue jusqu'à présent. Cependant, la réflexion s'est introduite légèrement plus tôt en programmation logique ([Wey80, Wey82][Bow85]) qu'en programmation fonctionnelle ([Smi82, Smi84]). De plus, les premières tentatives de formalisation de la réflexion ont vu le jour dans le cadre de la logique symbolique [Fef62][Per85, Per88]. En programmation logique, nous verrons dans les sections ultérieures que la réflexion n'est pas

vu tout à fait d'un même angle que le nôtre. Les exigences d'obtention de la réflexion sont différentes. Nous reviendrons plus loin sur ce sujet.

2. 3-Lisp

3-Lisp est un langage réflexif conçu par Smith[Smi82, Smi84]. Celui-ci est le premier chercheur à avoir développé la réflexion comportementale avec l'idée fondamentale de tour réflexive. Comme son nom l'indique, 3-Lisp a été écrit en Lisp. Ce langage respecte une rationalisation sémantique du langage Lisp que l'auteur a appelé 2-Lisp. Il est implémenté par un interprète méta-circulaire. Théoriquement, 3-Lisp se veut un empilement infini d'interprètes 2-Lisp, tous s'exécutant l'un l'autre et en même temps. Ce langage a pour fonction de rendre accessible et modifiable l'état de calcul à tout point d'exécution. En programmation fonctionnelle, l'état de calcul est généralement représenté par trois structures de données: l'expression, l'environnement et la continuation. L'accès aux entités réflexives se fait à l'aide de procédures réflexives c'est-à-dire des procédures qui, lorsqu'elles sont invoquées, sont évaluées non pas au niveau auquel elles ont été appelées mais plutôt à un niveau plus haut dans la tour d'interprétation. Ces procédures sont définies telle une simple procédure Lisp comme suit:

```
(define NOM
    (lambda reflect [ARGS, ENV, CONT] CORPS)
```

où `[ARGS, ENV, CONT]` est la liste des arguments de la procédure. `ARGS`, `ENV` et `CONT` sont les paramètres formels qui recevront comme paramètres réels les arguments, l'environnement courant et la continuation courante respectivement. La connexion causale se fait comme suit. Quand il y a un appel d'une procédure réflexive, la fonction (qui est considérée sous forme currifée) se retrouve sans ses arguments `ENV` et `CONT`. L'appel de la procédure a la forme suivante: `(NOM ARGS)`. L'interprète 3-Lisp ajoute les arguments manquants (`ENV` et `CONT`) à la fonction avant de l'appeler au méta-niveau. Après que l'exécution de cette procédure se soit terminé, le calcul reprend à l'endroit où il a été laissé mais cette fois-là avec la nouvelle continuation et le nouvel environnement lesquels ont été construits par la procédure réflexive.

3-Lisp implémente une tour potentiellement infinie. Pour remédier au problème d'appels de procédures réflexives récursives, le concepteur de 3-Lisp propose la création paresseuse des niveaux [Smi84].

Plusieurs applications sont possibles. Par exemple, en programmation fonctionnelle, il est possible de définir naturellement les fonctions bien connues `throw/catch`, la fonction "call with current continuation" (`call/cc`) et certaines procédures de déverminage. Il existe d'autres types d'application propre à ce genre d'implémentation de la réflexion dans d'autres paradygmes de programmation telle que la programmation par objets.

3. Brown et Blond: descriptions non-réflexive de la tour réflexive

Wand et Friedman ont développé à la suite des travaux de Smith, un langage fonctionnel réflexif appelé Brown[WF86] qui n'a pas les inconvénients d'inefficacité du système 3-Lisp car la tour réflexive est implémentée en n'exécutant qu'un seul interprète à tout moment, donc qu'un seul flux de calcul. Le but fondamental de cette approche est le développement d'une sémantique dénotationnelle des tours réflexives. Le développement d'une telle sémantique n'est pas simple car la réflexion a l'effet de détruire la compositionnalité du calcul, propriété fondamentale de la sémantique dénotationnelle en général. De même que pour le système 3-Lisp, Brown réifie l'environnement et la continuation. Il offre aussi l'accès à ces structures à l'aide de procédures réflexives. Les concepteurs de Brown décomposent la réflexion en deux actions distinctes qu'ils appellent **réification** et **réflexion** ([FW84]). La réification est l'action de passer les structures réifiées de l'interprète (l'environnement et la continuation) au niveau du programme. Tandis que la réflexion est le processus qui prend du programme un environnement et une continuation et réinstalle le tout au niveau de l'interprète. Respectivement, ces actions sont réalisées par les identificateurs `reifier` et `meaning` en Brown.

La différence importante entre Brown et 3-Lisp se trouve au niveau de l'implémentation. En 3-Lisp, l'exécution se fait par plusieurs flux de calcul ([Smi84]) tandis que Brown utilise une forme d'exécution qu'on dit unifilaire ("single-thread" [Dr88]). Pour éviter d'exécuter plus d'un interprète à la fois, Brown manipule une structure de données appelée la méta-continuation. Une méta-continuation est une pile de continuations de chaque niveau d'interprétation de la tour réflexive. Intuitivement, la méta-continuation peut être imaginée comme une pile d'interprètes. Au départ, la méta-continuation est vide. Au moment de l'action de réflexion, la continuation courante est insérée dans la méta-continuation. La nouvelle continuation courante est construite statiquement. Et au moment d'une réification, la nouvelle continuation courante est retirée de la méta-continuation et l'ancienne continuation est détruite. Plus formellement, la tour réflexive est construite en utilisant le combinateur de point fixe Y (de Church).

Peu de temps après, Danvy et Malmkjær ont développé leur langage fonctionnel réflexif qu'ils ont appelé Blond[Dr88]. Leur implémentation en Scheme est légèrement plus importante que celle de Brown, elle est plus riche et plus consistante en rapport à leur description formelle. Ils offrent une formalisation de la relation des domaines sémantiques de chaque niveau de la tour, une identification formelle de la "réification" et de la "réflexion" telles que définies par Wand et Friedman et une généralisation de la méta-continuation. À ce titre et à l'encontre de Brown, leur méta-continuation est une pile de paires, continuation et environnement. Ceci

permet une meilleure manipulation de la méta-continuation évitant que les évaluations d'expressions se fassent dans des environnements incorrects. Ils utilisent deux concepts relatifs aux manipulations de continuations c'est-à-dire la distinction entre continuation "pushy" et "jumpy". Généralement quand une continuation capturée est appliquée, le contexte dans lequel cela se fait c'est-à-dire la continuation courante, est détruite. Ce dernier phénomène correspond à l'application d'une continuation "jumpy". En Brown et Blond (par défaut), les continuation sont "pushy": quand une continuation réifiée est appliquée, la continuation courante est insérée sur la méta-continuation. Blond offre les deux possibilités par l'utilisation d'abstraction définie par les auteurs: la première avec la η-abstraction et la seconde avec la σ-abstraction.

Par une implémentation de la tour réflexive avec méta-continuation (comme Brown et Blond), tout effet de bord est exclu dans les étages supérieurs. En effet, puisqu'un seul interprète s'exécute à chaque moment, les effets de bord des interprètes de la méta-continuation ne pourraient être appelés au bon moment. Intuitivement, on explique qu'il n'est pas nécessaire d'exécuter tous les niveaux de la tour en même temps car ils effectuent tous des opérations sur des structures identiques et bien connues (par exemple, l'interprète qui est en train d'exécuter). Tandis que si un interprète, à la différence des autres, effectue des effets de bord successivement (une trace, par exemple), on ne peut plus dire que les interprètes opèrent sur des structures connues. Lorsqu'on se limite à la programmation fonctionnelle pure, cette restriction ne pose pas de problèmes parce que justement les effets de bord sont exclus. Néanmoins, en pratique, la possibilité de définition d'outils de déverminage d'un langage est indispensable. Il est donc préférable de chercher à garder cette puissance expressive en réflexion qu'est l'utilisation d'outils d'entrée/sortie.

En somme, la comparaison de Brown et Blond nous amènent à voir que Brown prend son origine d'un modèle réflexif sans tour réflexive pour en construire une réflexive tandis que Blond part d'une tour non-réflexive pour la rendre réflexive.

Nous terminons cette étude en rapport à la programmation fonctionnelle en discutant brièvement d'un dernier langage. D'autres langages fonctionnels réflexifs ont été pensés (comme Stepper [Baw88]) mais, pour les besoins de notre étude, leur explication est de moindre importance.

$I_{\mathcal{R}}$ [JF92] a été conçu en Scheme. À la différence de Brown et Blond, ce système ne suppose pas une sémantique unifilaire. L'originalité de ce système est sa grande simplicité d'implémentation. Contrairement à tous les systèmes réflexifs précédents (sauf 3-Lisp), son modèle est une tour réflexive finie plutôt qu'infinie. Cette tour réflexive est exécutée par le langage Scheme plutôt que par le langage $I_{\mathcal{R}}$. Clairement, ce langage a eu sa raison d'être car il s'avère que

relativement peu d'implémentations très simples de langages réflexifs ont été conçues en programmation fonctionnelle. Justement, nous avons retenu de ce langage sa représentation simple et directe d'une tour réflexive finie. Cela nous a aidé à concevoir notre premier langage réflexif déclaratif, 3-Prolog, que nous verrons plus loin. Auparavant, examinons quelques tentatives antérieures importantes de l'application de la réflexion dans le monde de la programmation logique.

4. Amalgamer langage et méta-langage

Dans la section précédente, nous n'avons présenté que des systèmes développés en rapport à la programmation fonctionnelle et à son formalisme, le lambda-calcul. Pour introduire la réflexion comportementale en programmation logique, il nous faut donner un aperçu des concepts de réflexion en programmation logique et aussi présenter quelques tentatives d'implémentation de la réflexion dans les langages déclaratifs[2].

Dans les langages déclaratifs, la distinction entre la programmation et la méta-programmation est beaucoup plus accentuée que pour tout autre paradigme de programmation. Dans la littérature, on retrouve très souvent une séparation plus ou moins forte des concepts de déduction et de théorie. La méta-programmation est attribuée au concept de déduction et la programmation au concept de théorie. Pour un même langage déclaratif, il peut donc y avoir deux sous-langages: un langage logique de base qui permet l'implémentation de différentes théories et un méta-langage qui permet l'implémentation de différents mécanismes de déduction. Il est donc raisonnable de se poser les questions suivantes: que doit être le rapport entre ces deux langages? Quelles sont les éléments essentiels pour leur cohabitation, leur amalgame? Nous répondrons à ces questions en examinant ce qui s'est dit sur le sujet.

Tout d'abord, il existe deux articles bien connus qui discutent spécifiquement de l'auto-référence dans les langages basés sur la logique du premier ordre. Ces articles ont été tous deux écrit par Perlis [Per85][Per88]. Dans le premier, il présente une théorie de la dénotation ("quotation") pour l'atteinte d'un langage tentant d'éviter les paradoxes (les plus célèbres, les paradoxes de Russell) de l'auto-référence des systèmes de logiques formels. Dans le deuxième, il propose l'utilisation de logiques modales. Malheureusement, pour des raisons d'espace, nous ne discuterons pas des travaux de Perlis car, somme toute, ils s'éloignent des nôtres. Il fallait néanmoins les citer.

FOL ("First Order Logic") [Wey80, Wey82] a été une première tentative reconnue de caractérisation du concept de réflexion dans un langage déclaratif (extension de Prolog). Ce

[2]Dans ce texte, les mentions de "langages déclaratifs" se réfèrent directement à la famille des langages du type Prolog excluant tous les autres paradigmes.

langage se base sur le concept fondamental de théorie. Pour caractériser une théorie, l'auteur, Weyhrauch, introduit la notion de **structures de simulation** qu'on doit comprendre comme étant la partie mécanisable d'un modèle (référence à la logique mathématique). Intuitivement, une structure de simulation est la relation entre les objets sur lesquels nous raisonnons dans une théorie et les mots que nous utilisons pour nommer ces objets. Simplement, la structure de simulation peut être vue comme la description "mécanique" ou fonctionnelle d'une théorie. Une théorie est un triplet $\langle L, SS, F \rangle$ où L est un langage, SS une structure de simulation et F une base de faits. En principe, FOL a la possibilité de manier autant de théories que l'on veut. Il est capable de changer son "attention" d'une théorie à une autre. Pour ce faire, Weyhrauch introduit le concept de méta-théorie. Une méta-théorie diffère des autres théories en ce sens qu'elle exprime des propositions à propos d'autres théories. Dans FOL, ce nouveau concept apparaît avec l'introduction d'un triplet spécial appelé META. META est une théorie générale de tout triplet de la forme $\langle L, SS, F \rangle$. Cette théorie contient des fonctions et des prédicats spéciaux qui permettent de "parler" par exemple, des expressions bien formées, des termes, des dérivations, des structures de simulations, etc. Puisque META est un triplet, il est raisonnable de se demander quelle est la forme de sa propre structure de simulation. C'est là qu'on retrouve le problème du recouvrement introspectif. En effet, le langage FOL, lui-même, sert de structure de simulation au triplet META. Il existe une implantation du langage FOL et il existe aussi, dans cette implémentation, une représentation naturelle et complète du langage FOL lui-même. Cette représentation se retrouve *dans* le langage FOL par l'intermédiaire de la théorie META. Weyhrauch ne fait qu'indiquer que l'implémentation FOL et sa représentation sont deux mêmes entités. Il ne présente donc aucune solution au problème du recouvrement introspectif. Il admet néanmoins que son approche et la réflexion tendent à concevoir l'image de tours de méta-théories qui s'apparente à l'idée de tour réflexive.

Deux ans plus tard, dans un article maintenant fréquemment cité, "Amalgamating Language and Metalanguage in Logic Programming" [Bow82], Bowen et Kowalski caractérisent certaines composantes de base d'un langage réflexif déclaratif. Ils disent que pour qu'il y ait "amalgame" d'un langage L et d'un méta-langage M (le méta-langage de L) il faut la présence sous une forme ou sous une autre des outils suivants:

(i) Une relation de dénotation ("naming") qui associe toute expression linguistique de L à au moins une variable libre de M. Une seule expression de L peut avoir plusieurs noms dans M mais chaque variable dans M est associée à une unique expression dans L.

(ii) Une représentation explicite du moteur d'inférence $\vdash_L$ par un symbole de prédicat `Demo` dans le contexte des propositions venant de la théorie Th (représentation explicite du moteur d'inférence) du langage M.

(iii)
$$\frac{Th \vdash_M Demo(A', B')}{A \vdash_L B}$$

(iv)
$$\frac{A \vdash_L B}{Th \vdash_M Demo(A', B')}$$

Les deux dernières composantes sont appelées, ensemble, les **principes de réflexion** par Weyhrauch ([Wey80]). À l'origine, la notion de principe de réflexion a été introduite en logique symbolique par Feferman ([Fef62]) qui signifiait la description d'une procédure qui avait pour effet d'ajouter à un ensemble d'axiomes A, de nouveaux axiomes A' dont la validité de ceux-ci (A') dépendait de la validité de ceux-là (A). Ceci avait des conséquences sur l'affirmation que tous les théorèmes déduits de A sont valides.

Nous retrouvons, en principe, dans toutes les implémentations subséquentes d'un système réflexif déclaratif, une ou plusieurs de ces dernières exigences en plus des autres notions développées dans la section 1. Pour les besoins de brièveté, nous nous référerons à ceux-ci en les appelant les **exigences d'amalgame de Bowen et Kowalski**.

Dans la prochaine section, nous verrons quelques tentatives plus récentes d'implémentation de la réflexion en programmation logique. En programmation logique, le débat sur l'efficacité des systèmes réflexifs se fait entre ceux qui pensent utiliser l'évaluation partielle pour spécialiser des méta-interprètes réflexifs et ceux qui pensent qu'il suffit de modifier les implémentations de langages déclaratifs pour leur incorporer efficacement certaines notions d'introspection.

5. Introspection et réification

Avant de présenter quelques implémentations, nous précisons ici un point important. La réflexion telle que nous l'entendons se fait principalement par *réification*, c'est-à-dire que la modification d'une entité réflexive passe par une manipulation d'une structure qui représente de façon unique une structure de données inaccessible autrement. Néanmoins, les recherches en programmation déclarative ont montré qu'il était possible d'obtenir une réflexion structurelle et comportementale efficace sans réification en offrant, à la place, une forme d'introspection réduite et fixe dans le langage. Pour simplifier la compréhension, nous distinguerons ici une introspection sans réification de celle avec réification en qualifiant la première d'introspection implicite et la seconde d'introspection explicite.

À titre d'exemple, le langage Prolog utilise une introspection implicite pour obtenir une certaine réflexion structurelle et comportementale. En effet, en Prolog, le programme n'est pas en lui-même un objet de première classe néanmoins il est possible d'y accéder et de le modifier dynamiquement à l'aide des prédicats `retract/1` et `assert/1`([SS86b][BBP+93]). Par analogie à la réflexion comportementale, le coupe-choix peut être vu comme un opération réflexive fixée dans le langage qui permet de modifier la pile de points de choix (de manière restreinte) sans, par contre, utiliser une représentation complète de celle-ci. Le prédicat `call/1` est aussi une opération réflexive fixe qui permet d'accéder minimalement au moteur d'inférence.

Nous allons discuter plutôt d'une implémentation souvent citée comme point central dans l'histoire de l'implémentation de la réflexion en programmation logique.

metaProlog[Bow85] a été conçu par Bowen en 1985. metaProlog traite de manière plus déclarative les prédicats méta-logiques `assert/1` et `retract/1` en introduisant le traitement de théories (bases de règles) en tant qu'objet de première classe. Il innove aussi en ayant un mécanisme qui distingue les variables du niveau-objet (niveau 0 dans la tour réflexive, figure 2.1) et les méta-variables. Ce dernier mécanisme assigne pour chaque objet syntaxique (clauses, buts et variables) dans le langage une constante unique (expression de dénotation). Ces mêmes constantes peuvent elles aussi être identifiables par d'autres constantes. L'auteur a évité de stratifier le langage en différents méta-niveaux (méta-niveau, méta-méta-niveau, etc.) par ce mécanisme. Le prédicat `demo/3` offre la possibilité aux programmes du niveau objet de passer au méta-niveau par un appel comme le suivant:

```
demo(Theorie, Proposition, Preuve)
```

où `Preuve` est la preuve du but `Proposition` dans la théorie `Theorie` (base de règles). Le mécanisme d'unification est rendu explicite par le prédicat `match(Gauche, Droite, In_Subst, Out_Subst)`. Enfin, metaProlog garde une trace des différentes théories au cours des modifications (`retract/1` et `assert/1`) en gardant en mémoire les noms associés à ces théories.

Ce système tente d'intégrer la première composante d'un langage réflexif (relation de dénotation) en programmation logique en introduisant une relation bijective entre les entités du langage et des constantes (autres entités). Il y a aussi introduction d'un prédicat `demo` qui s'apparente au prédicat du même nom introduit par Bowen et Kowalski pour leur principe de réflexion. Ce prédicat permet d'appeler directement l'évaluateur du langage sur un but et une théorie. En fait, ce langage utilise une instrospection implicite car il permet d'accéder et de modifier le programme dynamiquement sans réification du programme. Il ne donne pas plein accès au programme et à sa représentation dans le langage. En ce sens, le langage n'offre pas de

réflexion de structures autres que celles de Prolog. Enfin, le langage semble être un langage amalgamé au sens de Bowen et Kowalski. Néanmoins, il utilise une introspection implicite au sens où il n'offre pas de représentation explicite et pleinement manipulable des objets syntaxiques du langage.

À la suite de ce langage, il s'est construit d'autres langages réflexifs déclaratifs tentant d'introduire la réflexion (et la modularité) ([LMN91][BCL+88, LMN91][CL89, Cos90]). Néanmoins, ils ont tous utilisé l'introspection implicite (comparable à metaProlog) et tous ne répondent pas à la deuxième composante de Bowen et Kowalski: présence d'une représentation explicite du moteur d'inférence. Ils ne sont donc pas des langages amalgamés. Une première tentative d'introspection explicite s'est faite dans [Sug90]. En effet, ce langage appelé R-Prolog* réifie complètement le programme et la liste des unifications. Il utilise un mécanisme de dénotation pour accéder à une représentation explicite des variables. Une particularité de ce langage se retrouve au niveau de sa sémantique déclarative: l'auteur introduit la notion de méta-continuation (théoriquement similaire à celle des langages Brown et Blond) pour parler d'états d'exécution. Par contre, ce langage ne rend pas explicite le moteur d'inférence ou toute autre structure de données comportementales.

En somme, les langages réflexifs déclaratifs conçus jusqu'à maintenant ou bien, ne répondent pas complètement aux exigences d'amalgame de Bowen et Kowalski ou bien n'utilisent pas l'intropsection explicite des objets syntaxiques du langage. Dans le deuxième cas, la raison est simple: l'inefficacité causée par l'introspection explicite en programmation logique. En effet, contrairement à la programmation fonctionnelle, les chercheurs en programmation logique se sont tous restreints à une réflexion très limitée de peur de perdre une certaine efficacité de leurs langages. Ce qui n'est pas le cas des chercheurs en programmation fonctionnelle. Ces derniers ont conçu leurs langages réflexifs à l'aide de la méta-interprétation sachant bien que leur implémentation était malgré tout assez inefficaces. De plus, il semble que la tendance des recherches a été plutôt portée vers l'étude de la notion de modularité en programmation logique. Ce qui explique peut-être qu'aucune recherche sérieuse ne s'est faite en réflexion comportementale.

De ce fait, les chercheurs en réflexion en programmation logique ont délaissé assez rapidement la deuxième exigence d'amalgame de Bowen et Kowalski: la représentation explicite du moteur d'inférence. C'est dans cette situation que nos recherches trouvent leur raison d'être, comparativement aux recherches précédentes. En effet, nos langages réflexifs ont tous été implémentés à l'aide de la méta-interprétation avec introspection explicite. Ce qui nous a permis aussi de respecter les exigences d'amalgame de Bowen et Kowalski beaucoup plus précisément. Le reste du chapitre parle justement de ces nouveaux langages réflexifs.

6. Langages 3-Prolog

Dans l'article [MD93], nous avons développé un langage réflexif simple nommé 3-Prolog. Ce langage a la syntaxe et la sémantique du langage Prolog pur. Il offre aussi la possibilité au programmeur de définir des prédicats réflexifs examinant et modifiant à sa guise la résolvante en cours d'exécution. Concernant l'implémentation, nous nous sommes basé sur l'étude de deux langages réflexifs: 3-Lisp[Smi84] pour son contenu conceptuel et $I_{\mathcal{R}}$[JF92] pour sa simplicité d'implémentation. Le langage a été implémenté en Prolog à l'aide de la méta-programmation. Nous avons écrit un interprète méta-circulaire de manière à exécuter une tour réflexive *à la 3-Lisp* [Smi84, MCD91]. Néanmoins, contrairement à 3-Lisp, mais comme pour $I_{\mathcal{R}}$, nous ne pouvons exécuter que des tours réflexives finies.

Le but de la conception de ce langage et de notre exigence de simplicité d'implémentation était de démontrer l'applicabilité de la technique d'optimisation qu'est l'évaluation partielle pour éliminer la méta-interprétation à l'exécution de programmes réflexifs. Ce but a été atteint pour des programmes réflexifs très simples. Néanmoins, par ce langage, rien ne nous permettait de conjecturer que l'évaluation partielle pourrait optimiser efficacement des programmes réflexifs sophistiqués dans des langages déclaratifs réflexifs puissants. Pour tenter de remédier à cette lacune, nous nous sommes penché plus à fond sur la représentation des structures en Prolog de façon à simplifier le problème.

Nous avons abouti à la spécification d'un sous-ensemble bien défini du langage Prolog que nous avons nommé 2-Prolog[3]. À l'aide de ce langage, nous avons conçu plusieurs langages réflexifs tous présentant différents degrés de réflexion. 2-Prolog est une rationalisation du langage Prolog. Cette rationalisation nous a permis de réduire la complexité du problème de l'applicabilité de l'évaluation partielle et d'éliminer certaines faiblesses de celle-ci apparemment incontournables à l'heure actuelle en Prolog.

À la suite de ces recherches, nous avons conçu plusieurs langages réflexifs simples tous présentant différents degrés de réflexion. Ils implémentent presque tous une tour réflexive finie analogue conceptuellement au langage $I_{\mathcal{R}}$ de [JF92]. Ces langages expérimentaux, écrits en 2-Prolog, ont servi d'une part, à développer une façon simple d'implémenter la réflexion en programmation logique et d'autre part, à éclaicir les difficultés d'optimisation en rapport à l'évaluation partielle en Prolog. Un premier langage réflexif a été conçu pour réifier le programme en tant qu'entité de première classe du langage. Il se nomme 3-Prolog$_P$. En fait, ce langage

[3]Ce nom fait référence à la rationalisation du langage Lisp par Smith et appelé 2-Lisp. Mais attention, la rationalisation du langage Prolog n'est pas de même nature que celle de Lisp. Malgré cela, nous gardons le nom 2-Prolog pour signifier que la démarche pour l'intégration des notions de réflexion en Prolog est comparable à celle de Smith: d'abord, il est question d'une "simplification" du langage et, ensuite, la construction de la tour réflexive est abordée.

implémente une réflexion de structure de base. Un autre langage réflexif nommé 3-Prolog$_U$ a été développé pour la réification des unifications faites en cours d'interprétation d'une requête particulière. Le langage 3-Prolog$_R$, comme pour 3-Prolog de [MD93], réifie la résolvante.

Le dernier langage réflexif que nous avons conçu se nomme 3-Prolog*. Il est lui aussi écrit en 2-Prolog. Il est en quelque sorte l'intégration de plusieurs des entités comportementales que nous avons pensé rendre accessible au programmeur dans les langages précédents. En plus de celles-là, il rend disponible une représentation de la pile de points de choix. À la différence des autres langages 3-Prolog$_i$[4], ce langage est l'implémentation d'une tour réflexive sans fin comparable au langage 3-Lisp[Smi84]. Il a été conçu pour deux objectifs: démontrer la puissance d'implémentation par la méta-interprétation de la réflexion comportementale *à la 3-Lisp* en programmation logique et illustrer nos observations concernant les résultats de l'évaluation partielle sur celui-ci.

En conclusion, dans ce chapitre, nous avons vu les concepts de base de la réflexion en programmation fonctionnelle et en programmation logique. En programmation fonctionnelle, nous avons discuté du concept fondamental de tour réflexive et des différentes façons d'en faire l'implémentation. En programmation logique, nous avons discuté du rapport entre langage et méta-langage pour une bonne réflexion. Dans le chapitre suivant, nous discuterons plus à fond des langages 3-Prolog$_i$ et 3-Prolog*.

[4]La variable i représente les lettres P, U, ou R.

CHAPITRE 3

Langages réflexifs en programmation logique

Dans ce chapitre, nous implémenterons la réflexion comportementale à l'aide de la méta-interprétation en Prolog. Notre but est de démontrer la puissance de la réflexion par l'interprétation. Pour ce faire, nous montrerons qu'il est possible d'obtenir la puissance d'expression (et même plus) du langage Prolog à l'aide de la réflexion avec seulement un sous-ensemble déclaratif du langage Prolog.

Les implémentations de quatre langages réflexifs bien définis seront expliqués en détail. Tout d'abord, motivons le choix de ces langages. En Prolog, il y a trois actions comportementales importantes du moteur d'inférence:

- Sélection d'un but B dans la résolvante,
- Sélection d'une clause Cl dans le programme dont la tête se réduit au but B,
- Processus d'unification du but B avec la tête de la clause Cl,
- Création d'un point de choix dans la pile de points de choix.

Par la réflexion, nous aimerions offrir au programmeur les moyens de modifier ces actions comportementales ponctuellement ou pour un certain nombre de réductions du moteur d'inférence. Pour ce faire, nous allons réifier certaines structures de données manipulées régulièrement par ce dernier.

Sachant comment s'effectue les manipulations d'une entité réifiée, le programmeur pourra accéder à cette même entité pour la modifier à sa guise. Par exemple, le programmeur sait que le moteur d'inférence "choisit" toujours le premier but dans la résolvante. De ce fait, si le programmeur veut modifier la sélection d'un but, il devra permuter les buts dans la résolvante de manière à ce que la sélection des buts se réalise comme il le désire. Il est donc vrai que le programmeur en réflexion doit très bien connaître le comportement du moteur d'inférence. En fait, un programmeur qui n'a jamais écrit ou compris la méta-programmation déclarative ne pourra pas utiliser la réflexion pour la conception de ses programmes.

Les structures de données correspondant respectivement aux actions comportementales précédentes sont:

- la résolvante,
- le programme,
- la liste des unifications,
- la pile de points de choix.

Au cours de ce chapitre, nous présenterons des langages déclaratifs offrant au programmeur l'accès à une ou plusieurs des structures de données précédentes. Ces langages n'ont pas la prétention de fixer définitivement la manière dont la réflexion doit se réaliser en programmation logique. Non, ils ne sont qu'une tentative de démonstration de l'intégration de la réflexion.

Les recherches dans les langages de programmation ont montré que l'uniformité des différents concepts reliés à un langage est très importante pour une intégration saine du concept de réflexion dans celui-ci. Un exemple type supportant cette affirmation se retrouve dans la démarche de conception du langage 3-Lisp. En effet, le concepteur de ce langage a été contraint de rationaliser le langage Lisp avant de concevoir Lisp réflexif. 2-Lisp est le nom donné au langage Lisp rationalisé sur lequel se base 3-Lisp.

Il s'avère que Prolog a aussi besoin d'une rationalisation avant même l'intégration de la réflexion. En effet, Prolog n'est pas un langage dont les concepts sont bien uniformisés. Pour s'en convaincre, il suffit de penser que ce langage n'a pas de sémantique déclarative pour tous ses prédicats de système. Les prédicats méta-logiques n'ont pas de sémantique déclarative naturelle. Quelques tentatives (non explicites) de rationalisation de Prolog dans la but d'introduire la réflexion ont été vues dans les articles [CL89], [Sug90] et [MD93]

Pour les besoins de notre étude, nous avons donc imaginé la conception d'un langage presque complètement déclaratif, sous-ensemble du langage Prolog, excluant plusieurs des prédicats de système de Prolog. Ce langage, nous l'avons nommé 2-Prolog en guise de reconnaissance au langage 2-Lisp de Smith [Smi84]. Nous avons voulu restreindre 2-Prolog au plus petit sous-ensemble non trivial du langage Prolog de telle sorte qu'il soit simple de montrer que la réflexion par interprétation apporte certainement beaucoup à la conception de langages déclaratifs. En fait, nous verrons que la réflexion nous permet de définir tous les prédicats méta-logiques principaux de Prolog dans le langage 2-Prolog.

La suite de ce chapitre se composera des sections suivantes. La section 1 présente la rationalisation du langage Prolog, c'est-à-dire 2-Prolog et la représentation en 2-Prolog des structures de données précédentes. Les sections qui suivent présenteront des méta-interprètes 2-Prolog qui

réifient une ou plusieurs des structures dont nous avons discuté précédemment. Il sera donc question de langages réflexifs bien définis. Pour chacun d'eux, nous présenterons un exemple ou deux de programmes réflexifs intéressants. Dans la section 3, nous étudierons un langage où le programme en entier est une entité de première classe. Dans la section 4, les unifications deviendront accessibles. Dans la section 5, il sera question de la réification de la résolvante. Enfin, dans la section 6, la pile de points de choix deviendra une entité de première classe.

Prenons note que la conception de la réflexion dans les langages développés en 2-Prolog se base sur [Sug90] pour l'implémentation de la connexion causale, et sur [JF92] (à l'exception de 3-Prolog*) au sens où ces langages sont simples et sont l'implémentation d'une tour réflexive **finie**. La conception du langage 3-Prolog* se base significativement sur le langage FOL ([Wey80, Wey82]).

1. 2-Prolog

Le langage 2-Prolog est un sous-ensemble bien défini du langage Prolog. Ce langage admet seulement et exactement la syntaxe du langage Prolog. Comme pour Prolog, il intègre deux types de prédicats: les prédicats de système; des prédicats dont la sémantique ne peut en aucun moment changer (être redéfinie) dans le langage et des prédicats définis; des prédicats dont la sémantique peut être définie et modifiée par le programmeur.

Premièrement, les deux seules structures de contrôle admises en tant qu'entités fixes dans le langage sont le IF-THEN-ELSE[1] et la conjonction dont le prédicat est `','/2`.[2]. Ces structures de contrôle sont vues en tant que prédicats de système du langage.

Les structures de contrôle utilisées pour l'écriture des méta-interprètes réflexifs interprétant des programmes 2-Prolog doivent appartenir au langage 2-Prolog. C'est une des raisons qui ont motivé au cours de nos expérimentations notre choix de la structure de contrôle IF-THEN-ELSE plutôt que le coupe-choix qui est la structure de contrôle la plus utilisée en Prolog. Ceci s'explique par le fait que l'interprétation du coupe-choix est impossible à réaliser simplement. En effet, le coupe-choix a une portée dynamique sur la pile de points de choix tandis que la structure IF-THEN-ELSE a plutôt une portée statique, locale à la clause dans laquelle elle se situe. Son interprétation se fait facilement par la structure elle-même. Pour visualiser cela, il suffit d'examiner la clause d'interprétation suivante.

---

[1] IF-THEN-ELSE sera le nom donné à la structure Prolog de la forme `(A->B;C)` où `A`, `B` et `C` sont des termes 2-Prolog. Lors de l'exécution, le but `A` ne retourne qu'une seule solution. Les solutions alternatives ne sont pas calculées.

[2] Nous n'avons pas incorporé la structure `not/1` (négation) dans le langage en tant que prédicat de système car il est possible de définir celle-ci à l'aide de la structure IF-THEN-ELSE. Par contre, il est impossible de définir la structure IF-THEN-ELSE à l'aide de la structure `not/1`. La structure IF-THEN-ELSE a donc une plus grande puissance expressive.

```
solve((A->B;C)) :- solve(A) -> solve(B) ; solve(C).
```

Cette structure de contrôle nous permettra, comme pour le coupe-choix, de réduire la création des points de choix inutiles. Il est bien évident que nous n'aurions pas pu nous passer de structures de contrôle car l'interprétation de programmes déclaratifs de grande taille est impensable sans un minimum de contrôle sur le retour-arrière.

D'autres raisons nous assurent que notre rationalisation des structures de contrôle de Prolog est raisonnable. Celles-ci sont en lien avec l'évolution du domaine de recherche qu'est l'évaluation partielle. Nous discuterons de ces raisons dans le chapitre 4.

1.1. Prédicats de système de 2-Prolog. Voici les prédicats de système (qui n'ont pas de définition déclarative dans le langage) de 2-Prolog. Nous n'expliquons que très brièvement leur sémantique. Pour de plus amples explications, se référer à [BBP+93].

(i) `true/0` et `fail/0` sont des prédicats dont le premier est toujours vrai et le deuxième toujours faux (échec de résolution).

(ii) `is/2` unifie une variable (à gauche) au résultat d'un calcul arithmétique (sommation, soustraction, multiplication ou division) (à droite). `number/1` vérifie si un terme est un nombre entier. `'<'/2` test si le nombre de gauche est plus petit que celui de droite.

(iii) `write/1` et `read/1` imprime et lit un terme à l'écran d'affichage géré par la boucle "read-eval-print" de Prolog. Ce sont deux prédicats n'ayant aucune sémantique déclarative.[3]

(iv) `=../2` est un prédicat ayant deux paramètres. Le premier prend un but de la forme $p(a_1, a_2, \ldots, a_n)$ ou une variable qui sera unifiée à un but. Le deuxième est ou bien une liste `[` p`,` a_1`,` a_2`,` a_3`, ... ]` composée du prédicat et des paramètres d'un but ou bien une variable qui sera unifiée à une liste de la forme précédente.

(v) `call/1` invoque le moteur d'inférence de base de 2-Prolog sur un terme syntaxiquement correct.[4] [5]

---

[3] Nous aurions pu exclure ces prédicats qui nous empêchent d'avoir un langage 2-Prolog complètement déclaratif. En effet, en principe, ces prédicats pourraient être définis réflexivement dans le langage. Néanmoins, il s'est avéré que leurs exclusions compliquent le choix des exemples pertinents des chapitres ultérieurs. Mentionnons seulement que s'il fallait se donner un langage 2-Prolog purement déclaratif, il faudrait certainement exclure ces derniers.

[4] Il aurait été simple de rendre ce prédicat définissable dans le langage en restreignant sa portée aux prédicats de système seulement. Nous le n'avons pas fait pour simplifier le déverminage et les expérimentations.

[5] Quelques problèmes nous ont donné un certain fil à retordre à l'écriture de nos méta-interprètes 2-Prolog en Prolog. Ceux-ci sont inhérents aux systèmes Quintus Prolog et Sicstus Prolog que nous avons utilisés pour nos expérimentations. C'est un problème relatif au prédicat `call/1`. Le système Quintus Prolog offre aux usagers un procédé d'encapsulation de leurs programmes en modules (ensembles de programmes). Cette fonctionnalité oblige l'usager, dans plusieurs cas, à gérer les noms de modules dans ses méta-interprètes. La présence d'un but du type `call(B)` dans une clause d'un module est modifiée au chargement de la clause. Le but apparaît plutôt de la forme `call(`*module*`:B)` où *module* est le nom du module dans lequel la clause se présente. Toute occurence de la variable `B` dans la clause se voit transformée par le terme *module*`:B`. Ce type de transformation forcé par le système a des conséquences sur la sémantique de la variable `B`. En effet, par exemple, un appel du

Les deux dernières composantes sont importantes pour respecter, en partie, les exigences d'amalgame de Bowen et Kowalski.

Le prédicat `=../2` nous permet de décomposer la structure la plus petite en programmation logique: le but. Les prédicats `is/2`, `number/1` et `'<'/2` nous offrent la possibilité de réaliser des calculs et des tests arithmétiques simples. Le prédicat `write/1` et `read/1` nous permet de dialoguer dynamiquement avec l'usager. Enfin, le prédicat `call/1` nous permet d'invoquer le moteur d'inférence de 2-Prolog. Celui-ci servira aux méta-interprètes dans le but d'exécuter des prédicats de système 2-Prolog.

Prenons note que le choix des prédicats de système 2-Prolog aurait pu être autre. Nous aurions pu ajouter une gamme beaucoup plus grande de prédicats utiles à titre de prédicats de système. Néanmoins, pour les exemples de programmes présentés dans ce mémoire, cela n'était pas nécessaire. De plus, 2-Prolog doit rester un langage simple, uniforme, suffisamment puissant et offrant une grande versatilité. De cette manière, les difficultés et les caractéristiques de l'apport de la réflexion en programmation logique deviennent plus facile à identifier. Aussi, les performances et les incapacités fondamentales de l'évaluation partielle sont plus facilement identifiables.

1.2. Quelques méta-programmes en 2-Prolog. Voici une liste de quelques prédicats définis en 2-Prolog (ceux-ci sont souvent pré-définis dans les implémentations Prolog). Ils sont indispensables pour alléger la lecture du code de nos exemples. La sémantique de ces prédicats est clairement identifiable par leur définition en 2-Prolog.

- Le prédicat `not(+TERME)` (négation par échec de Prolog) est défini en 2-Prolog par le programme suivant.[6]

```
not(Terme) :- (call(Terme) -> fail; true).
```

- `functor(+-TERME, -+PREDICAT, -+ARITE)` retourne le nom du prédicat et l'arité du terme `TERME`.

```
functor(Terme, Pred, Arite) :-
  Terme =.. [ Pred | Arguments ],
  longueur(Arguments, Arite).

longueur(Liste, Nombre) :-
  longueur(Liste, 0, Nombre).
```

type *module*:`B = p(C)` peut échouer si le prédicat `p/1` n'est pas défini dans le module *module* même si `B` est une variable libre. Pour éviter ce problème, nous avons fait toutes nos expérimentations en Sicstus Prolog 0.7 #4. Ce système n'offre pas d'encapsulation par module des programmes. Le problème était évité.

[6] Nous aurions pu décider d'ajouter la structure de contrôle de négation par échec fini en tant que structure système du langage 2-Prolog. Nous l'avons pas fait car il est possible de définir celle-ci en fonction de la structure de contrôle IF-THEN-ELSE. Remarquons que l'inverse n'est pas possible. La structure IF-THEN-ELSE est donc plus puissante que la structure `not/1`.

```
longueur([], Nombre, Nombre).
longueur([_|RListe], Nombre, Rep) :-
  NNombre is Nombre + 1,
  longueur(RListe, NNombre, Rep).
```

Un autre prédicat du même type est défini en 2-Prolog. C'est un sucre syntaxique. Le prédicat a la forme suivante: `functor(+TERME, -PREDICAT/ARITE)`. Il sert simplement à ce qu'un terme IF-THEN-ELSE soit considéré un prédicat de foncteur et d'arité `ifcut/3`. Ce prédicat est utile pour simplifier la lecture et l'écriture des méta-programmes. Son utilisation s'explique par le fait que nous avons voulu garder la syntaxe courante au programmeur Prolog de la structure de contrôle IF-THEN-ELSE de la forme `(A->B;C)`. En voici la définition.

```
functor(Terme, Pred/Arite) :-
  functor(Terme, Pred, Arite),
  not(Pred/Arite = ';'/2).
functor((Terme;_), ';'/2) :-
  functor(Terme, Pred, Arite),
  not(Pred/Arite = '->'/2).
functor((_->_;_), ifcut/3).
```

- `arg/3` est bien connu des programmeurs en Prolog. Il permet d'accéder à un seul argument d'un terme. Il est défini comme suit en 2-Prolog.

```
% arg(+NUMERO, +TERME, -ARGUMENT).
arg(No, Terme, Arg) :-
  Terme =.. [_ | LTerme],
  sous_liste(No, LTerme, [ Arg | _ ]).

sous_liste(1, Liste, Liste).
sous_liste(No, [_ | RListe], NListe) :-
  1 < No,
  NNo is No - 1,
  sous_liste(NNo, RListe, NListe).
```

- `atomic/1` vérifie si un terme est un nombre ou un atome (un prédicat d'arité zéro). Si `Terme` est une variable libre, une erreur sera signalée et l'exécution sera suspendue.

```
atomic(Terme) :- Terme =.. [ Terme ].
```

Les prédicats `'='/2` (unification), `';'/2` (disjonction) et `nl/0` ("newline") sont aussi définissables dans le langage 2-Prolog. Nous avons omis de présenter leur définition pour cause de simplicité.

1.3. Représentation en 2-Prolog des entités à réifier. Voyons comment les quatre entités à réifier présentées en introduction, se voient attribuer une représentation en 2-Prolog. Prenons note que ces représentations ne sont pas fixes dans le langage 2-Prolog. Un usager peut modifier à sa guise ces représentations.

- Soit par exemple, la liste de buts suivante: A, B, C, ... La résolvante sera une structure construite par imbrication du prédicat de système `,/2` de telle manière qu'elle aura la forme suivante: `(`A`, (`B`, (`C`, (`D`, ...)))).`[7]
- Soit le programme suivant. T_1 `:-` C_1 T_2 `:-` C_2 T_3 `:-` C_3 ... Le programme sera une liste Prolog comme suit. `[ cl(`T_1`, `C_1`, `r_1`), cl(`T_2`, `C_2`, `r_2`), cl(`T_3`, `C_3`, `r_3`), ... ]`. les termes r_i sont des identificateurs pour chaque clause. Les variables locales pour chacune des clauses auront la forme d'un terme `v(No)`[8] où `No` est un nombre qui identifie de manière unique les variables, localement dans chacune des clauses.
- Soit à titre d'exemple, la liste des unifications suivantes: `A = a, B = b, C = B, D = E` La structure de liste d'unifications sera alors une liste Prolog comparable à `v(`m`, 0) = a, v(`m`, 1) = b, v(`m`, 2) = b, v(`m`, 3) = mv([3,4]), v(`m`, 4) = mv([3,4])`. Les termes `v/2`[9] représenteront des variables 2-Prolog. Les termes `mv/1` ne serviront que pour représenter les liens d'unification entre différentes variables. Dans l'exemple, le terme `v(`m`, 0)` représente la variable Prolog A. Le terme `v(`m`, 1)` représente la variable B, de même pour C, D et E. Puisque D=E, leurs représentations `v/2` sont donc liées dans la liste d'unifications par le même terme `mv([3,4])` où la liste `[3,4]` représente l'ensemble des variables unifiées l'une avec l'autre. Le paramètre m représente un type pour la représentation de la variable. Nous verrons ultérieurement sa raison d'être.[10]
- La pile de points de choix est beaucoup plus difficile à représenter déclarativement. Ceci s'explique par le fait que très peu de chercheurs ont tenté l'expérience. Pour des raisons de simplicité, nous nous sommes arrêté à la représentation suivante. Avant d'expliquer la représentation, nous devons utiliser le concept d'état. Un état sera une structure contenant les informations suivantes:

---

[7] Nous n'avons pas voulu linéariser la représentation de conjonctions de buts en utilisant plutôt la structure de liste. Nous avons trouvé que cette linéarisation n'apporte pas une simplification d'expression significative dans l'écriture des méta-interprètes. Pour que le programmeur Prolog comprenne plus rapidement nos méta-interprètes, nous avons donc gardé la représentation standard de Prolog par imbrication du prédicat `','/2`. Ce choix reste néanmoins arbitraire.

[8] Ces termes `v(No)` seront appelés des **méta-variables locales**.

[9] Ils seront occasionnellement nommés des **méta-variables** de 2-Prolog.

[10] Prendre note que cette représentation est analogue à bien des points de vue à la représentation utilisée dans le langage Gödel[Gur92].

- Un but choisi B.
- Une clause choisie Cl dont la tête s'unifie au but B.
- Une résolvante.
- Une liste d'unifications.

Un point de choix sera alors un état de cette forme. Une pile de points de choix sera une liste Prolog d'états. Il n'est pas nécessaire pour la bonne compréhension de connaître en détails la représentation Prolog d'un état. Il suffit de savoir néanmoins qu'un état aura la forme d'un programme similaire à la représentation précédente d'un programme. Nous omettons ici l'explication de sa représentation complète.

2. Un premier interprète méta-circulaire 2-Prolog

Voyons comment écrire un méta-interprète méta-circulaire (chapitre précédent, section 1) en 2-Prolog. Cela se fait sensiblement de la même manière qu'en Prolog mais il y a des différences. Voici un exemple de méta-interprète 2-Prolog.[11]

```
% solve(+BUT, +PROG).
solve(true, _).
solve(call(A), Prog) :-          % A doit etre instancie a un terme.
  solve(A, Prog).
solve((A,B), Prog) :-
  solve(A, Prog),
  solve(B, Prog).
solve((A->B;C), Prog) :-
  solve(A, Prog) -> solve(B, Prog); solve(C, Prog).
solve(A) :-
  functor(A, Pred_arite),
  not(Pred_arite = true/0), not(Pred_arite = call/1),
  not(Pred_arite = ','/2), not(Pred_arite = ifcut/3),
  (system(Pred_arite) ->
     call(A);
     clause_prog(A, B, Prog),
     solve(B, Prog)).

system('=..'/2).     system(is/2).        system('<'/2).
system(write/1).     system(number/1).    system(call/1).
system(','/2).       system(ifcut/3).     system(fail/0).
system(true/0).
```

Remarquons que le programme à interpréter est un paramètre du méta-interprète représenté par la variable `Prog`. Nous n'avons pas voulu introduire le prédicat de système `clause/2` de Prolog dans notre langage 2-Prolog parce qu'il est possible de le définir dans le langage à l'aide de la réflexion. De plus, de cette façon, le programme interprété est clairement identifiable par rapport au code du méta-interprète. Aussi, il s'avère que l'exécution de cet interprète

[11] Notons que l'exécution des programmes en 2-Prolog suppose que les clauses sont indexées sur le premier paramètre de la tête de celles-ci.

méta-circulaire est aussi efficace en Prolog que son homologue qui utilise le prédicat `clause/2`. En effet, cet interprète peut être complètement compilé avec le meilleur compilateur Prolog existant actuellement. Tandis qu'un interprète méta-circulaire utilisant le prédicat `clause/2` de Prolog ne peut pas être compilé significativement en Prolog car ses clauses doivent rester accessibles pour l'interprétation.

Rappelons que la représentation du programme est celle énoncée dans la section 1.3. Nous ne présentons pas ici le code du programme `clause_prog/3`. Ce code peut être examiné en annexe A section 1.

Le prédicat `call/1` est traité de manière particulière par le méta-interprète. La raison de cela est simple: nous voulons que cet interprète soit pleinement méta-circulaire. Si le programmeur décide d'ajouter des faits de la forme `system(Pred/Arite)` au méta-interprète, la méta-interprétation de ce nouvel interprète sera complète au sens où les prédicats des faits `system/1` ajoutés seront effectivement interprétés et non pas simplement exécutés (à l'aide de `call/1`) par l'interprète de base 2-Prolog. En d'autres mots, un appel de la forme `call(But)` où `But` n'est pas un prédicat de système sera vraiment interprété. De cette manière, le programmeur peut à sa guise modifier la définition du prédicat `system/1` sans conséquence sur la pleine interprétation de la tour d'interprètes.

3. 3-PROLOG$_P$

Le but de cette section est de montrer qu'il est possible de façon simple de réifier le programme pour la conception éventuelle de programmes modifiables dynamiquement. De plus, cette section aborde les problèmes d'amalgame du programme et du méta-programme dans une même entité (base de règles).

En méta-programmation en Prolog, le prédicat `clause/2` sert à accéder au programme contenu dans la base de règles de Prolog. Ce prédicat n'est pas déclaratif. Il n'est pas possible de définir simplement celui-ci sous une forme déclarative comme par exemple les prédicats pré-définis de 2-Prolog (`'='/2`, `functor/3`, `atomic/1`, etc.).

Dans cette section, nous présentons un langage réflexif, sur-ensemble de 2-Prolog qui rend le programme en sa totalité une entité de première classe du langage. Pour alléger le texte, nommons ce nouveau langage 3-Prolog$_P$. Ce langage réflexif implémente une tour réflexive finie. En ce sens, 3-Prolog$_P$ réifie le programme et à fortiori le code du méta-interprète si la tour réflexive contient au moins trois étages.

Avant de passer au code en lui-même, nous expliquons rapidement comment se fait la réflexion en 3-Prolog$_P$. Comme nous l'avons expliqué dans le chapitre 2, un programme réflexif

comporte deux types de prédicats: les prédicats ordinaires activés par des buts ordinaires et les prédicats réflexifs. Aussi, nous identifions deux types de buts réflexifs: les buts réflexifs implicites et explicites. Les premiers sont identifiés par le prédicat `reflexif_implicite_P/1` et les autres par le prédicat `reflexif_explicite_P/1`. On identifie alors les buts réflexifs implicites dans le programme par les faits `reflexif_implicite_P(Predicat/Arite)` où `Predicat/Arite` est un prédicat réflexif. L'identification se fait de manière similaire avec les buts réflexifs explicites. Les buts réflexifs explicites sont toujours explicitement introduits aux endroits où ils doivent être appelés. C'est le programmeur qui décide du moment opportun auquel ils doivent être invoqué. Tandis que les buts réflexifs implicites sont invoqués à chaque itération principale[12] du méta-interprète. Ils ne sont jamais invoqués explicitement dans le programme. En fait, les buts implicites peuvent être pris en tant qu'ajout de traitement sur une entité réifiée introduit au niveau du méta-interprète.

3-Prolog$_P$ exécute les buts réflexifs explicites au niveau immédiatement supérieur dans la tour réflexive finie en lui "passant" en deux paramètres supplémentaires: le programme interprété et une variable représentant le programme éventuellement modifié par le but réflexif. Ces deux paramètres supplémentaires doivent apparaître dans la définition du prédicat réflexif (qu'il soit implicite ou explicite) qui se trouve au niveau du méta-interprète.

Comprenons bien que le code des prédicats réflexifs ne se trouve jamais au niveau du programme interprété. Ces prédicats étant toujours invoqués au niveau du méta-interprète, leurs définitions doivent donc se retrouver parmi les définitions des prédicats du méta-interprète.

Il est évident que, puisque la tour réflexive n'est pas infinie, la demande d'exécution d'un but réflexif explicite au dernier étage de la tour entraînera, en principe, une erreur. En effet, en 2-Prolog, par exemple, le prédicat réflexif `clause/2` n'est pas explicitement défini. Il nous faut donc à tout coup une tour réflexive suffisamment haute pour l'exécution de programmes incluant des appels réflexifs explicites. Pour chaque programme réflexif P, il faudra déterminer à l'avance le nombre fini de niveaux nécessaires à la tour réflexive pour exécuter le programme sur une certaine requête. Dans le cas du langage 3-Prolog$_P$, il ne semble pas qu'il soit difficile de déduire cette information. Néanmoins, dans le cas d'autres langages réflexifs présentés ultérieurement, cette information pourrait devenir difficile à acquérir. Si justement, c'est le cas, il suffira simplement d'exécuter le programme P successivement avec les niveaux d'interprétation 0, 1, 2, ... jusqu'à ce que l'exécution se passe sans plainte de la part de 2-Prolog.

Voici la définition du coeur de méta-interprète 3-Prolog$_P$.

[12] L'itération principale est la boucle de résolution d'interprétation; celle-ci contient la sélection d'un but dans la résolvante, la sélection d'une clause dans le programme, la réduction du but et de la clause et enfin la construction de la nouvelle résolvante.

```
solve_prog(true, P, P).
solve_prog(Resolvente, Prog, NProg) :-
        functor(Resolvente, Pred),
        not(Pred = true/0),
        selectionne_but(Resolvente,But,ResteResolvente),
        appel_reflexion_implicite(Prog, Prog_x),
        reduit_refl(But, Corps, Prog_x, Prog_xx),
        nresolvente(Corps, ResteResolvente, NResolvente),
        solve_prog(NResolvente, Prog_xx, NProg).
```

Le prédicat principal `solve_prog/3` prévoit deux cas: si la résolvante ne contient que le but `true` alors il termine la résolution, sinon, il effectue une première réduction. Cette réduction consiste en plusieurs sous-opérations toujours exécutées dans le même ordre. Premièrement, il y a appel de tous les prédicats réflexifs implicites (s'il en existe). Cela se fait à l'aide du prédicat `appel_reflexion_implicite/2`. Les appels réflexifs implicites auront pour effet de modifier le programme avant la réduction. Ensuite, il y a sélection d'un but B dans la nouvelle résolvante. Il y a réduction de ce but B à une clause C. Il y a mise à jour de la résolvante en y ajoutant le corps de la règle C. Enfin, la résolution se poursuit sur la résolvante reconstituée.

Il ne semble pas y avoir, à priori, de préférence du moment opportun à l'exécution des buts réflexifs implicites. Nous avons choisi d'exécuter ces buts juste avant la réduction. Nous aurions pu choisir de les exécuter un peu après, par exemple juste avant la sélection du but. Pour l'instant, ce choix semble être arbitraire. Voici maintenant comment les buts réflexifs implicites sont construits et appelés.

```
appel_reflexif_implicite(Prog, NProg) :-
  appel_refl_imp([], Prog, NProg).

appel_refl_imp(Liste, Res, NRes) :-
  ((reflexif_implicite_P(P/A),
    not(member(P/A, Liste))) ->
      functor(But, P, A),
      But =.. ListeBut,
      append(ListeBut, [Prog, Prog_x], NListeBut),
      NBut =.. NListeBut,
      metavar(NBut),
      appel_refl_imp([P/A|Liste], Prog_x, NProg);
      NProg = Prog).

metavar(But) :- call(But).
```

Ce programme commence par choisir un premier prédicat réflexif implicite. On reconstitue le but réflexif implicite à l'aide du prédicat et de son arite. On ajoute deux paramètres dont le premier contiendra le programme et le deuxième, une variable qui recevra le nouveau programme modifié par l'appel réflexif implicite. Cette dernière variable permettra au méta-interprète de

garder contact avec la résolvante courante. Ensuite, on appelle le but réflexif construit. Enfin, on fait de même pour les autres prédicats réflexifs implicites.

Un appel explicite de la réflexion se fait de manière similaire à un appel implicite. Dans ce cas, c'est le prédicat `reduit_refl/4` qui s'en occupe. Voici le programme relatif à ce prédicat. Notons que le programme en entier peut être examiné en annexe A section 2.[13]

```
reduit_refl(call(But), But, Prog, Prog).
reduit_refl(But, true, Prog, Prog) :-
  functor(But, Pred),
  not(Pred = call/1),
  system(Pred),
  call(But).
reduit_refl(But, true, Prog, NProg) :-
  functor(But, Pred),
  reflexif_explicite_P(Pred),
  But =.. ListeBut,
  append(ListeBut, [Prog, NProg], NListeBut),
  NBut =.. NListeBut,
  call(NBut).              % Appel du but nouvellement constitue.
reduit_refl(But, Corps, Prog, Prog) :-
  functor(But, Pred),
  not(system(Pred)), not(reflexif(Pred)),
  clause_prog(But, Corps, Prog).
```

Mentionnons que dans [Sug90], un même style de passage de paramètres est implémenté pour réaliser la connexion causale entre les entités réifiées avant et après leur modification.

Voici maintenant à titre d'exemple, la définition du prédicat méta-logiques `clause/2` de Prolog écrit en 3-Prolog$_P$ à l'aide des notions de réflexion explicite.

```
reflexif_explicite_P(clause/2).

clause(Tete, Corps, Prog, Prog) :-
  clause_prog(Tete, Corps, Prog).
```

À l'aide de cette définition, nous retrouvons un peu plus le standard d'écriture des méta-interprètes en Prolog. En exemple, voici le méta-interprète 2-Prolog (vu dans la section précédente) écrit en 3-Prolog$_P$. Nous avons donc affaire ici à un méta-programme réflexif. Nous savons tous que l'utilisation du prédicat `clause/2` facilite la lecture et la compréhension du méta-interprète. Néanmoins, en théorie, il rend le programme interprété difficilement identifiable de façon précise.

```
% solve(+BUT).
solve(true).
solve(call(A)) :- solve(A).
solve((A,B)) :- solve(A), solve(B).
solve((A->B;C)) :- solve(A) -> solve(B); solve(C).
solve(A) :- functor(A, Pred_arite),
    not(Pred_arite = true/0), not(Pred_arite = call/1),
```

[13] Le prédicat `clause_prog/3` est défini en annexe A section 1

```
    not(Pred_arite = ','/2), not(Pred_arite = ifcut/3),
    (system(Pred_arite) ->
       call(A);
       clause(A, B),
       solve(B)).

reflexif_implicite_P(_) :- fail.
reflexif_explicite_P(clause/2).

clause(Tete, Corps, Prog, Prog) :-
  clause_prog(Tete, Corps, Prog).
```

Concernant le méta-interprète 3-Prolog$_P$, l'efficacité de son implémentation pourrait être meilleure en 2-Prolog. Un certain ralentissement est premièrement causé par l'exécution du prédicat `clause_prog/3`. En effet, c'est celui-ci qui s'occupe de choisir et d'exemplariser les clauses avant la réduction. Cette exemplarisation est "paresseuse" dans le sens suivant. Avant la réduction, **tout** le programme est exemplarisé. L'exemplarisation est en fait une transformation de la représentation des clauses ne contenant que des termes `v/1` aux emplacements des variables vers une représentation où tous ces termes sont remplacés par des variables libres. C'est ce qui facilite, ensuite, l'unification du but courant avec la tête de la clause. Néanmoins, cette transformation est très coûteuse en nombre de réductions car toutes les clauses du programme doivent être décomposées et analysées, et ceci, pour chaque réduction au cours de la résolution. Deuxièmement, si le programme contient beaucoup de clauses, la recherche d'une clause qui se réduit au but courant est linéaire dans le programme sans aucune indexation sur le premier paramètre de la tête des clauses comme cela se fait pour la base de règles de Prolog.

Dans le cas de l'exemplarisation, bien sûr, nous avons pensé, en premier lieu, éviter le coût de la transformation en se donnant une représentation du programme avec variables libres et non pas avec les termes de la forme `v/1`. Néanmoins, ce type de représentation exige que nous utilisions, ensuite, le prédicat méta-logique `copy_term/2` de Prolog pour réaliser les exemplarisations. L'utilisation de ce dernier nous aurait obligé à l'ajouter en tant que prédicat de système à notre langage 2-Prolog[14]. Son utilisation va à l'encontre de notre volonté de nous restreindre le plus possible à un langage déclaratif. Il nous fallait donc se donner une représentation des variables du programme sous forme de termes pour éviter d'utiliser `copy_term/2` pour l'exemplarisation.

Une certaine diminution du travail du méta-interprète aurait pu être atteinte en évitant l'exemplarisation paresseuse des clauses. En effet, il suffit d'effectuer la transformation coûteuse seulement sur ces clauses concernées plutôt que sur tout le programme à chaque réduction. Il s'avère que ce type d'optimisation du méta-interprète cause certaines difficultés à l'évaluation

[14] ou plutôt ajouter le prédicat `var/1`.

partielle. En gardant la transformation paresseuse et en utilisant l'évaluation partielle, nous obtenons de meilleurs résultats d'efficacité. Nous en discuterons plus à fond par la suite.

4. 3-PROLOG$_U$

Le méta-interprète réflexif présenté ici permet d'accéder à une structure de liste contenant toutes les unifications qui se sont produites au cours de la résolution. Cette section aborde de ce fait, les questions non-triviales de représentation des variables dans le méta-langage.

Le méta-interprète précédent utilisait en tant que méta-variables des variables libres pour représenter les variables du niveau objet. Si nous voulons une représentation des variables du niveau objet qui nous donne pleine liberté de manipulation sur les unifications, il nous faut représenter les variables du niveau objet autrement que par des variables libres. Nous utiliserons donc une représentation où chaque variable est un terme `var(Niveau, No)` telle que défini dans la section 1. Pour les besoins de la méta-circularité, `Niveau` sera un entier positif qui désignera le niveau dans la tour réflexive et `No` sera aussi un autre entier positif qui, celui-là, identifiera de façon unique la variable que la méta-variable veut représenter. Le paramètre `Niveau` est important car il servira à la différentiation des méta-variables, des méta-méta-variables, des méta-méta-méta-variables, etc.

L'implémentation de la réflexion est sensiblement pareille à celle de l'implémentation du langage 3-Prolog$_P$ en 2-Prolog. Pour alléger le texte, nous omettons son explication en détail. Se référer à l'annexe A section 4.

Voici un exemple simple de programme réflexif dans ce langage. Ce programme est la définition réflexive du prédicat méta-logique `var/1`. Il ne faut pas oublier que la connexion causale se réalise de la même manière que pour 3-Prolog: il y a reparamétrisation du but réflexif (implicite ou explicite) avant son interprétation au méta-niveau. Le langage R-Prolog*[Sug90] offre une redéfinition du prédicat `var/1` similaire à la nôtre.

```
reflectif_explicite(var/1).

var(Terme_var, u(Niv, Unif), u(Niv, Unif)) :-
  functor(Terme_var, v, 2),
  arg(1, Terme_var, Niv),
  ( member((Var=Terme), Unif) ->
    functor(Terme, Pred),
    not(Pred = mv/1);
    fail).
```

Notons que dans [HL88a], Hill et Lloyd ont essayé de trouver une sémantique déclarative pour leur langage de logique typé ("many-sorted"). Ce langage distingue les variables du niveau objet et les méta-variables en utilisant un mécanisme de dénotation comparable au nôtre. Par

contre, ce langage n'utilise pas la réflexion (montée au méta-niveau) pour pouvoir définir `var/1` dans le langage.

Le programme semble peut-être compliqué à première analyse. Essayons de comprendre ce qui se produit. Supposons que le but `var(V)` s'exécute au niveau Niv où `V` est une variable libre. Un appel d'un prédicat `var/3` est exécuté au méta-niveau plutôt qu'au niveau objet. Ce qui implique que la variable `V` "change" de représentation quand elle passe au niveau $Niv + 1$. Puisque le but `var(V)` est monté d'un niveau, son interprétation se fera par le méta-interprète du niveau $Niv + 2$ et non pas par le méta-interprète du niveau $Niv + 1$. Ce qui veut dire que l'entité V (qui était considérée comme une variable libre par le méta-interprète du niveau $Niv + 1$) sera considérée comme un terme de la forme `v(`Niv`, No)`. Pour le méta-interprète du niveau $Niv + 2$, ce terme ne représente pas une variable mais bien un terme clos étant donné que seules les termes de la forme `v(`$Niv+1$`, No)` sont pris pour des variables libres. En d'autres mots, seuls ces termes sont des méta-variables. Nous voyons ici l'utilité du premier paramètre contenant le numéro du niveau auquel appartient la variable.

En somme, une entité représentant une variable libre d'un niveau perd son statut de variable libre quand cette entité change de niveau dans la tour. Cela veut dire que les variables deviennent vraiment des entités de première classe à l'aide de la réflexion.

Voici maintenant d'autres définitions réflexives explicites de prédicats méta-logiques impossibles à (re)définir précisément en 2-Prolog sans la réflexion par méta-interprétation.

```
reflexif_explicite_U('='/2).

'='(Terme1, Terme2, Niv, u(N, Unif), u(N, NUnif)) :-
  unify(Terme1, Terme2, Niv, Unif, NUnif).
```

Le prédicat bien connu `'='/2` a pour fonction d'unifier deux termes. L'algorithme d'unification est défini d'une manière complètement déclarative par le programme `unify/5`. Celui-ci est défini en annexe A section 4.

```
reflexif_explicite_U('=='/2).

'=='(Terme1, Terme2, Niv, u(N, Unif), u(N, Unif)) :-
  unify(Terme1, NTerme1, Niv, Unif),
  unify(Terme2, NTerme2, Niv, Unif),
  NTerme1 = NTerme2.
```

Le prédicat `==/2` vérifie si les deux termes sont égaux au sens où les sous-termes correspondant dans les deux termes sont égaux et les variables libres correspondantes contenues dans les sous-termes sont égales ou liées entre elles par unification. Sans vouloir entrer trop dans les détails, ce programme fonctionne comme suit. Le prédicat `unify/4` unifie le terme `Terme1`

avec la variable libre `NTerme` en tentant de remplacer toutes les méta-variables[15] du terme `Terme1` par leur unification possiblement présente dans la liste `Unif`. Si une méta-variable dans le terme `Terme1` est libre (n'a pas d'unification présente dans `Unif`) celle-ci sera représentée (dans le terme `NTerme`) par une variable libre. Le prédicat `unify/4` est aussi appelé sur le terme `Terme2` et retourne le terme `NTerme2`. Si les termes `NTerme1` et `NTerme2` s'unifient entre eux alors `Terme1 == Terme2` est vraie.

En fait, `unify/4` sert à "transformer" le terme contenant des méta-variables vers un terme similaire contenant des variables libres sans aucune méta-variable.

Voici la définition réflexive du prédicat méta-logique bien connu `findall/3` défini en 3-Prolog$_U$. Puisqu'il est question d'interprétation d'une requête dans un *programme* bien déterminé et qu'en 3-Prolog$_U$, le programme n'est pas une entité réifiée, nous avons ajouté un paramètre supplémentaire au prédicat: le programme dans lequel la requête doit être interprétée. Le prédicat est donc de 4 paramètres. Notons que sa définition est courte comme toutes les autres définitions réflexives dans ce langage. C'est un avantage certain. Pour des raisons d'espace, nous omettons ici l'explication de cette définition.

```
% findall(+STRUCTURE, +BUT, -LISTE, +PROGRAMME, +NIVEAU, +UNIF, -NOUV_UNIF).
findall(Struct, But, Liste, Prog, Niv, u(No, Unif), u(No, [(Liste=NListe)|Unif])) :-
        findall(But, Prog, Niv, u(No, Unif), [], Liste_unif),
        construit_liste(Liste_unif, Niv, Struct, NListe).

% findall(+BUT, +PROGRAMME, +NIVEAU, +UNIF, +LISTE, -NOUV_LISTE).
findall(But, Prog, Niv, u(No, Unif), Liste, NListe) :-
        ((solve_refl(Niv, But, No, NNo, Unif, NUnif, Prog),
        not(member(u(NNo, NUnif), Liste))) ->
        NListe = [ u(NNo, NUnif) | NListe_x ],
        Liste_x = [ u(NNo, NUnif) | Liste ],
        findall(But, Prog, Niv, u(No, Unif), Liste_x, NListe_x);
        NListe = []).

construit_liste([], _, _, []).
construit_liste([ u(_, Unif) | RUnifs ], Niv, Struct,[Struct_unif|RStruct_unif]) :-
        unify(Struct, Struct_unif, Niv, Unif),
        construit_liste(RUnifs, Niv, Struct, RStruct_unif).
```

Enfin, voici une redéfinition du prédicat `not/1`. Nous savons que des résultats incohérents peuvent apparaître si un but insuffisamment instancié est appelé négativement. Cette redéfinition aide donc au déverminage des programmes avec négation par échec. Pour alléger le texte, nous ne présentons que partiellement le code de cette redéfinition. Le prédicat `ground/1` bien connu des programmeurs en Prolog est un prédicat méta-logique et réflexif qui vérifie s'il y a

[15] Rappelons qu'une méta-variable est la représentation sous forme d'une terme de la forme `v(Niv, No)` d'une variable du niveau immédiatement inférieur.

présence ou non de variables libres dans le terme en paramètre. En d'autres mots, il vérifie si le terme est clos ("ground").

```
not(But) :-
  ( ground(But) ->
     write('ATTENTION: Un appel de negation par echec est tente'), nl,
     write('sur un but contenant des variables libres!'), nl,
     write('Le but est: '), write(But), nl;
     true),
     (call(But) ->
       fail;
       true)).

reflexif_explicite_P(ground/1).

ground(But, Niv, Unif, Unif) :- ...
```

Notons enfin que cet interprète est tout à fait inexécutable à plus de 1 niveau d'interprétation en Prolog. Il faudra absolument penser à passer à un langage mieux adapté pour ce type de réification. Par exemple, il est possible de conjecturer qu'une implémentation dans le langage Gödel serait suffisamment efficace. En effet, aidé de l'évaluation partielle, la méta-programmation en Gödel est aussi rapide que la méta-programmation standard en Prolog[Gur94a]. En Gödel, la méta-programmation se fait toujours en utilisant une réification complète des unifications. En utilisant les outils de méta-programmation du langage Gödel pour écrire le langage 3-Prolog$_U$, nous obtiendrions sûrement une amélioration des performances d'exécution des requêtes comparable à l'exécution de requête du langage Prolog.

5. 3-PROLOG$_R$

Dans cette section, il sera question de la réification de la résolvante, une composante de l'état d'exécution.

Dans [MD93], nous avons développé un langage réflexif simple pour un sous-ensemble non-limité[16] du langage Prolog. Cet article rendait compte des premiers résultats obtenus de l'étude de l'évaluation partielle comme technique de transformation de programmes destinée à l'optimisation de l'exécution des tours réflexives comme nous l'avons défini dans la section 2.1. Les résultats à l'époque était encourageants car dans des cas simples (programmes-objets simples), l'évaluation partielle réussissait à éliminer toute méta-interprétation pour des tours réflexives finies contenant jusqu'à quatre niveaux.

Le méta-interprète qui va suivre différera de celui présenté dans l'article. Quelques améliorations ont été pensées entretemps. Entre autres, en principe, 3-Prolog$_R$[17] est écrit en 2-Prolog

[16] Dans ce langage, l'ensemble des prédicats de système autorisés n'a pas été fixé.
[17] Le nom du langage dans l'article est 3-Prolog.

et non en Prolog et, de plus, contrairement au langage de l'article, 3-Prolog$_R$ est un langage bien défini par rapport au langage Prolog.

```
solve_refl(true, Prog).
solve_refl(Resolvente, Prog) :-
  functor(Res, Pred),
  not(Pred = true/0),
  selectionne_but(Resolvente, But, RResolvente),
  appel_reflexion_implicite(Resolvente, RResolvente_x),
  reduit_refl(But, Corps, RResolvente_x, TResolvente, Prog),
  nresolvente(Corps, TResolvente, NResolvente),
  solve_refl(NResolvente, Prog).
```

Le code précédent est le coeur du méta-interprète 3-Prolog$_R$. Il est sensiblement pareil aux méta-interprètes réflexifs précédents. La réflexion implicite et explicite est aussi comparable à ce qui s'est fait jusqu'à maintenant. Nous omettons donc l'explication de son fonctionnement.

Voici un exemple de programme réflexif écrit en 3-Prolog$_R$. C'est un programme qui définit une sorte de coupe-choix qui a une portée globale sur l'ensemble de l'exécution plutôt que locale à la clause. Pour les besoins de la cause, nommons ce nouveau coupe-choix, `cc_fg`. La portée de ce coupe-choix n'a pas d'influence sur les points de choix déjà créés mais plutôt sur les points de choix *à venir*. En fait, quand ce prédicat est invoqué, il a pour effet d'empêcher toute création de points de choix ultérieur. Ce type de prédicat "méta-logique" a son utilité quand un programmeur veut que si une certaine clause est appelée, le reste de l'exécution se fasse sans aucune création de point de choix. Voici la définition du prédicat `cc_gf/0` en 3-Prolog$_R$.

```
reflexif_explicite_R(cc_gf/0).

cc_gf(Res, NRes) :-
  NRes = (call(Res) -> true; fail).
```

Nous verrons au chapitre 5 des exemples d'utilisation de ce type de prédicat.

Voici maintenant un exemple de programme réflexif implicite. Ce programme simple réoganise l'ordre des buts dans la résolvante. Il déplace le corps de la règle (si elle existe) qui a été ajouté (en début de résolvante) au cours de la réduction précédente, en fin de résolvante. Par exemple, si la résolvante a la forme suivante `((a, b, c), d, e, f)` où `(a, b, c)` est le corps de la règle de la réduction précédente alors le programme `deplace_corps_fin/2` transformera la résolvante en la suivante: `(d, e, f, (a, b, c))`. Cela a le même effet que l'action d'un méta-interprète qui ajoute le corps des règles en fin de résolvante plutôt qu'en début.

```
reflexif_implicite(deplace_corps_fin/0).

deplace_corps_fin(Res, NRes) :-
  functor(Res, Pred),
  (Pred = ','/2 ->
```

```
Res = (A, B),
NRes = (B, A);       % Permutation des termes.
NRes = Res).
```

Il est facile de concevoir d'autres programmes réflexifs pour ce langage. Néanmoins, nous sommes vite à cours d'exemples utiles. C'est normal car ce langage ne nous offre qu'une seule information réifée: la résolvante. Dans l'article [MD93], d'autres exemples de programmes réflexifs sont présentés. Nous n'y reviendrons pas ici.

6. 3-Prolog*

Maintenant, nous allons passer à un méta-interprète beaucoup plus sophistiqué. Notre motivation de la conception de ce méta-interprète a été soutenue par notre désir, tout d'abord, d'atteindre une limite de réification en programmation logique: la pile de points de choix et, aussi, d'atteindre un niveau de puissance de réflexion comparable au langage 3-Lisp.

Le langage réflexif venant de ce méta-interprète se nomme 3-Prolog*. Il fait suite aux langages 3-Prolog$_i$[18] et 3-Lisp. À propos de ce dernier, 3-Prolog* s'en rapproche encore plus que les langages 3-Prolog$_i$ conceptuellement, car il implémente une tour réflexive potentiellement infinie. 3-Prolog* ne réifie pas les unifications. Ceci s'explique par les expérimentations que nous avons présentées dans la section 4. La réification des unifications rend l'exécution, en Prolog, beaucoup trop inefficace pour réaliser des tests de plusieurs niveaux d'interprétation. Conceptuellement, 3-Prolog* n'est pas implémenté en 2-Prolog mais plutôt en 3-Prolog$_U$ dans lequel nous avons défini le prédicat réflexif `var/1` de telle manière qu'il ait exactement la même sémantique que le prédicat de même nom du langage Prolog.

Il faut dire que nous avons triché. L'expérimentation de notre implémentation du langage 3-Prolog* s'est faite en Prolog plutôt qu'en 3-Prolog$_U$. Ceci est possible étant donné que le prédicat `var/1` a exactement la même sémantique que celui de Prolog. Nous pouvions faire nos essais en Prolog seulement, sans entraîner de modifications conceptuelles. Il est pratiquement impossible d'exécuter 3-Prolog* en 3-Prolog$_U$ implémenté en Prolog. Comme nous l'avons dit, l'implémentation de 3-Prolog$_U$ est beaucoup trop inefficace en Prolog.

Comme pour les langages 3-Prolog$_i$, 3-Prolog* distingue deux sortes de prédicats: réflexifs et non-réflexifs. Une autre distinction est faite entre les prédicats réflexifs à l'aide des prédicats `reflexif_explicite_E/1` et `reflexif_implicite_E/1`. Les buts réflexifs explicites sont "montés" au méta-niveau immédiatement supérieur dans la tour réflexive comme dans les langages 3-Prolog$_i$. Un but réflexif peut être appelé à n'importe quel niveau dans la tour réflexive puisqu'elle est potentiellement infinie. En réalité, la tour réflexive ne s'exécute pas avec un "nombre

[18] L'indice i représente les lettres R, U ou P.

infini" d'étages mais plutôt par le nombre spécifié par l'usager au départ. Ensuite, s'il y a un appel réflexif explicite demandant un étage supplémentaire, 3-Prolog* s'occupe d'ajouter à la volée l'étage en question laissant ensuite l'appel du but réflexif se faire plus haut. [19]

À propos de la connexion causale, 3-Prolog* ne ressemble pas aux langages 3-Prolog$_i$. Par exemple, 3-Prolog$_R$ contenait en paramètre la structure réifiée accessible et il s'occupait de rendre accessible cette structure à la demande. 3-Prolog*, au contraire, est complètement séparé des structures réifiées au sens où il interagit constamment avec une base de règles particulières contenant les informations réifiées. Nous qualifions cette base de règles de **virtuelle** pour la distinguer de la base standard de 2-Prolog. Il y a donc une séparation du moteur d'inférence (méta-interprète 3-Prolog*) et de l'information sur laquelle ce moteur travaille. Cette séparation est en accord avec la philosophie des systèmes experts. Rappelons qu'un système expert se divise toujours en deux entités: le moteur d'inférence et la base de connaissance. De son côté, 3-Prolog* se divise en trois entités distinctes: le moteur d'inférence, la base de règles du programme de l'usager et la base de règles virtuelle. Cette dernière contient plusieurs faits contenant les réifications des structures de données utilisées par le moteur. Voici les principaux.

(i) `goal/1` contient en paramètre le but courant qui participera à la prochaine réduction réalisée par le moteur d'inférence.

(ii) `clauses/1` contient une pile d'identificateurs uniques [20] des clauses contenues dans la base de règles du programme. Ces clauses sont celles qui peuvent se réduire au but `goal/1` courant.

(iii) `goal_stack/1` contient la résolvante courante à qui on a enlevé le but courant.

(iv) `stack/1` contient la pile de points de choix. Cette pile contient en fait des bases de règles virtuelles. Chaque base de règles permet de "redémarrer" la résolution à certains endroits dans l'arbre de recherche.

(v) `num_stack/1` contient un numéro représentant le nombre de points de choix créés.

(vi) `program/1` contient la base de règles avec laquelle le moteur d'inférence cherche à inférer de nouveaux faits.

À l'aide de ces **réifieurs** (c'est le nom que nous leur donnons [WF86, Dr88]), nous pourrons accéder et modifier aux principales informations réifiées.

---

[19] Il y a une autre manière d'exécuter des buts au méta-niveau. Cela se fait par le prédicat de système (de 3-Prolog*) `metacall/1` comparable au prédicat 2-Prolog `call/1` à la différence qu'il exécute les buts que l'on lui donne en paramètre au méta-niveau immédiatement supérieur.

[20] Se rapporter à la définition de la représentation du programme utilisée à la section 1.3.

Comment accéder à la base de règles virtuelle? Il faut tout d'abord définir un prédicat réflexif qui sera résolu au méta-niveau.[21] Dans le programme du prédicat "monté", il suffit d'invoquer les réifieurs précédents. Cette dernière s'intancie à l'entité réifiée au moment de sa réduction. En fait, on ne peut invoquer un réifieur au niveau objet (niveau 0) de la tour réflexive). Ces réifieurs n'ont une signification qu'au niveau du méta-interprète. Pour comprendre cette contrainte, il faut se rappeler le problème du recouvrement introspectif (chapitre 2 section 1).

Comment modifier une entité réifiée? Il existe deux prédicats de système en 3-Prolog* qui permettent d'enlever et d'ajouter une clause dans une base de règles virtuelle. Ces prédicats sont comparables aux prédicats méta-logiques Prolog `retract/1` et `assert/1`. Pour les différencier de ceux de Prolog, nous les avons nommés `forget/1` et `assume/1`. Si, par exemple, un programmeur veut modifier la résolvante en ajoutant un but, il utilise tout d'abord, `forget/1` pour retirer la résolvante de la base de règles virtuelles courante. Il modifie ensuite la structure de donnée en question. Après, il utilise `assume/1` pour réinsérer le nouvelle résolvante dans la base virtuelle. Voici l'exemple dans la notation de 3-Prolog*.

```
reflexif_explicite_E(ajoute_but_resolvante/1).

ajoute_but_resolvante(But) :-
  forget(goal_stack(Resolvante)),
  NResolvante = (But, Resolvante),
  assume(goal_stack(NResolvante)).
```

Encore une fois, il faut comprendre que ces transformations *doivent* à tout coup s'opérer au méta-niveau. Dans le petit programme précédent, remarquons que le prédicat `ajoute_but_resolvante/1` sera toujours résolu au méta-niveau parce qu'il est un prédicat réflexif. En fait, la base de règles virtuelles n'existe que pour les appels du méta-interprète 3-Prolog*. Conceptuellement, il n'existe pas de base de règles virtuelle pour le niveau 0 de la tour réflexive. S'il y a plus d'un méta-niveau dans la tour réflexive, chaque méta-niveau a sa propre base de règles virtuelle indépendante des autres et chaque base de règles virtuelle n'est accessible que par un appel au méta-niveau auquel elle appartient.

Voici un petit programme réflexif implicite en 3-Prolog*. Ce programme peut être vu comme une redéfinition grossière de la trace du déboggeur de Prolog.

```
reflexif_implicite_E(trace/1).

trace(0) :-
  nl,
  goal(B),          % Appel d'un reifieur
  fonctor(B, Pred),
  (Pred = true/0 ->
   true;
```

[21] Ou bien utiliser le prédicat de système 3-Prolog* `metacall/1` pour "monter" des buts au méta-niveau.

```
   write(B),
   write(' ?'),
   read(_)).
```

Voici un exemple de son interprétation en 2-Prolog. Nous avons interprété une requête sur le programme `grand_parent/2` que nous retrouvons en annexe B section 1.

```
| ?- prog_grand_parent(Prog), tpl(grand_parent(X,Y), Prog).
grand_parent(_40,_41) ?y.

grand_mere(_40,_41) ?y.

mere(_40,_109) ?y.

parent(jane,_41) ?y.

pere(jane,_41) ?y.

mere(jane,_41) ?y.

X = france,
Y = lucie ?
```

Pour comprendre cette trace, il faut examiner les clauses effectivement utilisées au cours de la résolution de la requête `grand_pere(X,Y)`.

(i) `grand_parent(X, Z) :- grand_mere(X, Z).`

(ii) `grand_mere(A, C) :- mere(A, B), parent(B, C).`

(iii) `mere(france, jane).`

(iv) `parent(A, B) :- pere(A, B).`

(v) `parent(A, B) :- mere(A, B).`

(vi) `mere(jane, lucie).`

Voici maintenant le programme réflexif `!/0` explicite qui a la même sémantique que la structure de contrôle de Prolog coupe-choix. Cet exemple est une démonstration de la puissance expressive du langage 3-Prolog*.

```
reflexif_explicite_E(getbacktrack/1).
reflexif_explicite_E(cutbacktrack/1).

reflexif_implicite_E(recherche_cc/0).

% getbacktrack(-NUMERO).
getbacktrack(No) :- num_stack(No), cut_floor.

% cutbacktrack(+NUMERO).
cutbacktrack(0).
cutbacktrack(No) :-
        0 < No,
        stack(Pile),
        functor(Pile, Pred),
```

```
        cutbacktrack(Pile, Pred),
        cut_floor.                  % Appel reflexif qui "detruit"
                                    % l'etage courant devenu
                                    % inutile dans la tour
                                    % reflexive.

% cutbacktrack(+PREDICAT, +PILE).
cutbacktrack([]/0, _).
cutbacktrack(Pred, Pile) :-
        not(Pred = []/0),
        forget(stack(Pile)),
        cutbacktrack(Pile, No, NPile),
        assume(stack(NPile)).

% cutbacktrack(+PILE, +NO_PILE, -NOUVELLE_PILE).
% clause_prog(+TETE, -CORPS, +PROGRAMME).

cutbacktrack([], _, []).
cutbacktrack([ Pt | RPile ], No, NPile) :-
        clause_prog(num_stack(Num_stack),_, Pt),
        (Num_stack = No ->
            NPile = RPile;
            cutbacktrack(RPile, No, NPile)).

% Programme reflexif implicite.
recherche_cc :-
        num_stack(No),
        forget(goal(G)),
        recherche_cc(G, NG, No),
        assume(goal(NG)),
        forget(goal_stack(Res)),
        recherche_cc(Res, NRes, No),
        assume(goal_stack(NRes)).

recherche_cc(Goal, NGoal, No) :- [...] % Remplace le but "!" par le but "cutbacktrack(No)".
```

Voyons comment ce programme fonctionne. Ce dernier comporte trois prédicats réflexifs. Un premier se nomme `getbacktrack/1`. Son appel a pour effet d'unifier une variable au nombre de points de choix créé jusqu'au moment de son appel (utilisation du réifieur `num_stack/1`). Le prédicat de la forme `cutbacktract(+Num)` sert à retirer de manière itérative les points de choix de la pile de points de choix jusqu'à ce que le nombre `Num` soit égal au numéro de points de choix du premier point dans la pile. Celui-ci est ensuite retiré de la pile. C'est le dernier point de choix retiré de la pile.

En bref, les prédicats `getbacktrack/1` et `cutbacktrack/1` servent à couper les points de choix localement dans une clause. Par exemple, dans une clause comme la suivante, les prédicats réflexifs retirent les points de choix (possiblement) créés au moment de la résolution des buts `c` et `d` mais ils n'ont aucune influence sur les points de choix venant des buts `b` et `e`.

```
a :- b, getbacktrack(Num), c, d, cutbacktrack(Num), e.
```

Enfin, le prédicat `recherche_cc/0` est réflexif implicite. Il vérifie avant chaque réduction s'il y a présence dans la résolvante courante, du but `!` (coupe-choix). S'il est présent, il est immédiatement remplacé par le but `cutbacktrack(Num)` où `Num` est le nombre de points de choix créé à l'instant de sa détection.

Ces trois prédicats réflexifs, ensemble, offrent une implémentation correcte du coupe-choix de Prolog.[22]

La plupart des implémentations Prolog supportent seulement les coupe-choix statiques c'est-à-dire des coupe-choix valables s'ils apparaissent explicitement dans une clause du programme. Contrairement à ceux-ci, l'implémentation du coupe-choix en 3-Prolog* offre l'utilisation de coupe-choix dynamiques au sens où ceux-ci peuvent être introduits dynamiquement dans la résolvante. Ce qui est une amélioration d'expression comparativement au langage Prolog. En fait, le coupe-choix dynamique n'est pas implémenté dans les systèmes Prolog actuels car il empêche les compilateurs traditionnels Prolog d'optimiser efficacement les programmes.

Une justification d'un tel programme 3-Prolog* se retrouve en rapport à une des raisons d'être de la réflexion. C'est la même que celle concernant la méta-interprétation: le prototypage rapide et simple de modification de la sémantique du langage. Par exemple, un programmeur pourrait être intéressé de modifier la sémantique du coupe-choix pour des raisons d'enseignement.

Néanmoins, il est bien évident que le coupe-choix qui a pour fonction originelle d'accélérer l'exécution de programmes déclaratifs ne semble plus répondre vraiment à cet avantage dans ce cas-ci. En effet, l'exécution de ce programme est lente (de notre implémentation de 3-Prolog* en Prolog) comparativement au coupe-choix Prolog. Et ceci s'explique par le fait qu'il y a méta-interprétation du coupe-choix. Nous verrons dans le chapitre 5 quelques perspectives de recherche pour rendre l'utilisation d'une telle implémentation du coupe-choix plus raisonnable.

6.1. Méta-3-Prolog*. Dans les sections qui vont suivre, nous allons discuter de l'implémentation de 3-Prolog* en 2-Prolog.

Nous avons mentionné que 3-Prolog* est un langage qui implémente une tour réflexive qui peut varier en hauteur dynamiquement et n'ayant aucune limite théorique sur le nombre d'étages. Pour ce faire, il a fallu d'abord développer un nouveau langage non-réflexif que nous avons nommé Méta-3-Prolog*. Son implémentation s'est fait en 3-Prolog$_U$ dans lequel nous avons défini le prédicat réflexif `var/1`. Seuls le programme interprété et la résolvante sont

[22] Ce programme réflexif a été testé sur plusieurs programmes 2-Prolog avec coupe-choix. Mais, son utilisation pratique exigerait une preuve d'équivalence sémantique avec le coupe-choix de Prolog.

explicites et correctement manipulés par l'interprète Méta-3-Prolog*. Celui-ci est capable d'interpréter tout le langage 2-Prolog et, en plus, deux nouveaux prédicats nommés `forget/1` et `assume/1`. Ceux-ci ont la sémantique dont nous avons discuté dans les quelques paragraphes précédents. Dans le langage Méta-3-Prolog*, ces derniers prédicats sont considérés des prédicats de système c'est-à-dire que leur sémantique ne peut être altérée par le programmeur.

3-Prolog* est un interprète pour le langage Méta-3-Prolog* et il est écrit en Méta-3-Prolog*. De plus, 3-Prolog* est méta-circulaire. Ce qui veut dire qu'il est capable d'interpréter, autant les programmes 3-Prolog* que les programmes Méta-3-Prolog*.

Méta-3-Prolog* a aussi la fonction de détecter les prédicats réflexifs[23] sans, par contre, être capable de les interpréter. Quand il détecte la présence d'un but réflexif dans la résolvante, il "démarre" automatiquement le méta-interprète 3-Prolog* *sur* la résolvante en question de manière à se "décharger" de la tâche d'interprétation du prédicat réflexif et du reste de la résolvante. Schématiquement, voici comment cela se fait. Soit `Res` la résolvante courante du méta-interprète Méta-3-Prolog*. Supposons que Méta-3-Prolog* détecte la présence d'un but réflexif. Voici la transformation produite sur la tour réflexive.

Méta-3-Prolog(`Res`) $\rightarrow$ Méta-3-Prolog(3-Prolog*(`Res`))

Chaque niveau d'interprétation 3-Prolog* correspond à un étage de la tour réflexive. Si Méta-3-Prolog* démarre l'interprète 3-Prolog* en cours d'exécution à cause de la présence d'un prédicat réflexif alors cela signifie qu'un nouvel étage supérieur a été ajouté à la tour réflexive. C'est l'implémentation de la création paresseuse des niveaux de la tour.

Comment peut-on "démarrer" en cours d'exécution un interprète 3-Prolog* si la pile de points de choix n'est pas explicitement disponible? En effet, Méta-3-Prolog* ne gère pas explicitement la pile de points de choix. Nous avons voulu gardé cette pile implicite au niveau du langage 2-Prolog et du langage Méta-3-Prolog*. La raison est simple. Tout d'abord, il faut mentionner que 3-Prolog* réifie la pile de points de choix. C'est lui qui peut offrir une représentation de la pile au programmeur dynamiquement. L'interprète 3-Prolog* a été écrit de telle façon qu'il ne génère aucun point de choix au niveau de l'interprète Méta-3-Prolog*. Alors, si Méta-3-Prolog* interprète le langage 3-Prolog*, nous sommes assurés que Méta-3-Prolog* n'a généré aucun point de choix au cours de son travail. Il est donc simple de "démarrer" un deuxième interprète 3-Prolog* "par dessus" un premier car il suffit d'invoquer celui-là sur la résolvante courante (explicite à Méta-3-Prolog*) avec une pile de points de choix *vide*.

[23] Prendre note qu'il détecte de la même manière le prédicat de système `metacall/1`. Pour alléger le texte, nous ne mentionnerons plus ce prédicat au cours des explications. Comprenons seulement que s'il est question de prédicats réflexifs, ce prédicat `metacall/1` est traité de la même manière que ceux-là.

Une question nous vient à l'esprit concernant l'évolution de la tour réflexive. Est-il plus avantageux de détruire les étages de la tour tout de suite au moment où ils deviennent inutiles pour le cours du calcul? En effectuant plusieurs expériences en 2-Prolog, nous nous sommes rendu compte que, oui, il est préférable d'enlever les étages d'interprétation inutiles. Un niveau d'interprétation 3-Prolog* supplémentaire est très coûteux en temps de calcul, tellement coûteux qu'il semble toujours préférable de s'en débarrasser le plus vite possible. La figure suivante montre un exemple d'évolution de la tour réflexive en cours d'interprétation. De par la conception de 3-Prolog*, il est très simple de pouvoir détruire un étage de la tour réflexive sans altérer le cours du calcul. Sans vouloir entrer dans les détails, comprenons seulement que, pour ce faire, nous avons défini un prédicat réflexif que nous avons nommé `cut_floor/0`. Remarquons que ce prédicat a été utilisé dans le programme réflexif coupe-choix présenté antérieurement. Sa définition est simple mais assez technique. Il n'est pas important d'en savoir plus là-dessus.

6.2. Méta-interprète 3-Prolog*. Pour bien programmer de façon réflexive en 3-Prolog*, il faut absolument savoir comment fonctionne le méta-interprète 3-Prolog*. Nous expliquons dans cette sous-section quelles sont les opérations principales de l'interprète 3-Prolog* sans, par contre, présenter le code de celui-ci.

Voici, dans l'ordre, les opérations en question se produisant à l'intérieur de l'appel récursif `troispl/2`, prédicat principal de l'interprète. Supposons qu'il y a eu auparavant n réductions depuis le début de l'interprétation de la requête. En d'autres mots, supposons qu'il y a eu n appels récursifs sur `troispl/2`. Supposons aussi que les réductions sont numéroté 1, 2, 3, ..., n, ... Les instructions suivantes représentent donc l'appel $n + 1$.

(i) Construction de la nouvelle résolvante avec le corps de la règle de la réduction précédente. S'il n'y a pas de réduction précédente ($n = 0$), la résolvante courante reste telle quelle.

(ii) Sélection d'un but B dans la résolvante courante R. Mise à jour du réifieur `goal/1` et `goal_stack/1`.

(iii) Sélection d'une règle Cl dans le programme qui s'unifie avec le but B. Construction de la structure contenue en paramètre du réifieur `clause/1`. Cette structure Cls est une liste des clauses s'unifiant au but B. Si le but B est un prédicat de système, il n'y a donc pas de règle choisie et la structure Cls est une liste vide.

(iv) Appel du prédicat réflexif explicite si possible. S'il existe, celui-ci a été détecté au moment de la réduction précédente et stocké temporairement dans la base de règles virtuelles courantes.

(v) Appel de tous les prédicats réflexifs implicites s'ils existent.

(vi) Création d'un point de choix s'il est nécessaire en utilisant l'état courant. Généralement, le point de choix créé sera l'état courant dans lequel on a modifié le réifieur `clauses/1`.

(vii) Réduction du but B choisi avec la règle Cl.

(viii) Fin de l'appel $n+1$. Appel récursif sur `troispl/2`.

À quel endroit, dans l'ordre des opérations précédentes, est-il préférable d'appeler les prédicats réflexifs implicites? Remarquons que cet appel se fait juste avant la création d'un point de choix. Cette position est discutable. Après réflexion, cela nous a semblé plus commode. En fait, le point de choix créé dépend de tous les réifieurs. Il est donc préférable d'effectuer les modifications sur les réifieurs avant la création du point de choix de telle manière que le point de choix dépende aussi des modifications sur l'état.

En conclusion, nous avons analysé plusieurs langages réflexifs tous implémentés directement ou indirectement en 2-Prolog, une rationalisation du langage Prolog. Nous avons vu qu'il est simple à l'aide de la réflexion de définir les principaux prédicats méta-logiques du langage Prolog (par exemple, `clause/2`, `var/1` et !/0) dans un langage déclaratif (sans utiliser ceux-ci bien entendu). Il est clair que plusieurs implémentations des langages réflexifs présentés sont tout à fait inefficaces en Prolog. Dans les chapitres suivants, nous présenterons quelques exemples d'optimisation de ces langages à l'aide de l'évaluation partielle.

CHAPITRE 4

Évaluation partielle en Prolog

1. Introduction

Depuis maintenant plus de dix ans, l'évaluation partielle est reconnue comme étant une technique d'optimisation en programmation logique. Bien qu'encore beaucoup d'obstacles pratiques et théoriques empêchent son utilisation de pair avec les techniques d'optimisation traditionnelle (compilation), plusieurs résultats encourageants sont apparus au cours des dernières années. Nous présenterons dans ce chapitre un aperçu des résultats importants dans le domaine. À titre d'introduction, voici une brève définition de la technique et un exemple simple.

L'évaluation partielle est une technique de transformation syntaxique de programmes. Dans le cadre de la programmation logique, pour certaines requêtes R et un programme logique P, un évaluateur partiel [1] retourne un nouveau programme P' (éventuellement) plus efficace (moins de réductions) pour toute requête s'instanciant à l'une des requêtes R. De plus, le programme P' retourne les mêmes réponses (correctes et bien calculées) que le programme P. Cette technique de "compilation" de programmes logiques utilise le mécanisme de résolution (renommé dépliage dans la littérature dans le domaine). Cette technique connaît particulièrement du succès dans le cas de méta-interprètes. On parle alors de spécialisation de méta-interprètes. Il a été démontré que, dans plusieurs cas, la spécialisation de méta-interprètes est capable d'éliminer complètement la couche d'interprétation. De plus, dans le cadre de la programmation logique pure[2], l'évaluation partielle est souvent appelée **déduction partielle**. Pour simplifier la compréhension, attardons-nous premièrement à cette dernière.

La déduction partielle peut être vue sous forme d'une sémantique théorique. De ce point de vue, on retrouve quelques concepts bien connus, entre autres, les règles de sélection, dérivations et d'arbres de dérivation. La tâche principale de la déduction partielle est de construire un

[1] À l'heure actuelle, toutes les implémentations d'algorithmes d'évaluation partielle de programmes logiques se sont faites en méta-programmation logique.

[2] Le terme pur est utilisé en rapport à la programmation logique du langage Prolog dans lequel il y a des prédicats non-logiques tels que le coupe-choix, `var/1` et d'autres.

arbre de dérivation partielle à partir d'une expression bien formée R (requête). Cette dernière est la racine de l'arbre. L'arbre de dérivation est partiel car certaines feuilles de l'arbre correspondent à des requêtes non complètement résolues. Une **résultante** est formée à partir de chaque dérivation finie qui ne termine pas par un échec. C'est une clause dans laquelle la tête est la requête R rendue plus spécifique par certaines substitutions et le corps de la clause est une conjonction de requêtes non-complètement résolues qu'on retrouve aux feuilles de l'arbre de dérivation. Une interprétation intuitive d'une résultante peut être vue en rapport à l'arbre de dérivation partielle. Une résultante peut être vue comme une preuve partielle de la requête. Si les requêtes non-complètement résolues sont éventuellement prouvées vraies alors la conclusion (la requête R) est donc vraie.

```
grand_parent(X, Z) :- grand_mere(X, Z).
grand_parent(X, Z) :- grand_pere(X, Z).

grand_pere(A, C) :- pere(A, B), parent(B, C).
grand_mere(A, C) :- mere(A, B), parent(B, C).

parent(A, B) :- pere(A, B).
parent(A, B) :- mere(A, B).

pere(daniel,lucie).
pere(andre, daniel).
pere(jean,denis).
```

Considérons le programme précédent et la requête Prolog `grand_parent(Qui, lucie)` où `Qui` est une variable libre. Cherchons à construire le programme résultant à partir de cette requête. Supposons aussi que la règle de sélection est de choisir l'atome le plus à gauche dans la résolvante sans jamais choisir un atome de prédicat `mere/2`. Cette dernière condition vient du fait que le programme de départ est partiel au sens où il n'y pas de définition pour le prédicat `mere/2`. À partir de cette règle, on peut construire l'arbre de dérivation qu'on retrouve en figure 4.2

Voici le programme qui peut être facilement inféré à partir de l'arbre de dérivation.

```
grand_parent(Qui, lucie)      :- mere(Qui, daniel).
grand_parent(Qui, lucie)      :- mere(Qui, B), mere(B, lucie).
grand_parent(daniel, lucie)   :- mere(lucie, lucie).
grand_parent(andre, lucie).
grand_parent(andre, lucie)    :- mere(daniel, lucie).
grand_parent(jean, lucie)     :- mere(denis, lucie).
```

Les prédicats `grand_pere/2`, `grand_mere/2`, `parent/2` et `pere/2` ont été éliminés par la transformation. Ce résultat est correct et bien calculé en rapport à la sémantique procédurale et déclarative, respectivement. De plus, ce programme est évidemment plus efficace que le premier pour toute requête de la forme `grand_parent(Qui, lucie)`.

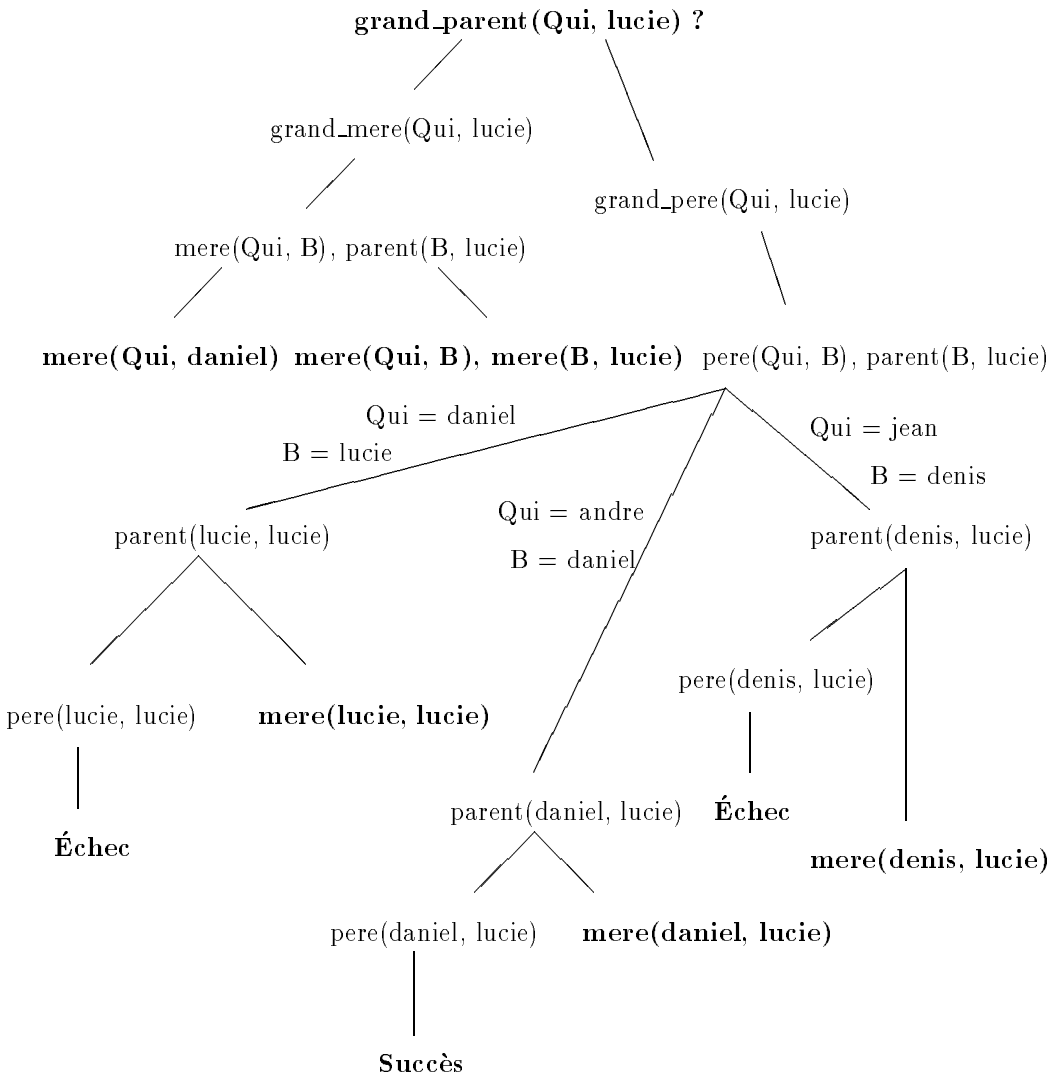

FIGURE 4.2. Arbre de dérivation pour `grand_parent(Qui, lucie)`

Plus formellement, la déduction partielle doit respecter la propriété d'intégrité et de complétude suivante.

- L'exécution de $P \cup \{R\}$ et de $P' \cup \{R\}$ retournent des réponses bien calculées *identiques*.[3]

De plus, dans la plupart des cas, elle doit aussi respecter cette autre propriété.

- $P \cup \{R\}$ échoue dans un temps fini si et seulement si $P' \cup \{R\}$ échoue aussi dans un temps fini.

Cette dernière est importante pour le respect de l'intégrité de la transformation dans le cas d'utilisation de règle d'inférences avec négation par échec fini. Néanmoins, il s'avère qu'il n'est pas possible de respecter cette propriété en sa totalité pour toutes les situations. En effet, dans certains cas, l'évaluation partielle transforme les boucles en un échec fini. Seulement un résultat plus limité a été démontré relatif à l'intégrité pour la règle d'inférence avec négation par échec fini [LS91b].

Dans la suite de ce chapitre, nous discuterons des différentes notions générales, théorèmes importants, problèmes encore à résoudre de l'évaluation partielle d'un point de vue théorique et opérationnelle en programmation logique.

2. Théorie et algorithme

Dans cette section, nous allons présenter un algorithme d'évaluation partielle extrait de [Gal93]. Les cinq prochaines définitions sont des traductions des définitions de [LS91b]. La certaine terminologie de base se retrouve dans [Llo87].

Définition 2.1. ***Résultante***

Une résultante est une expression du premier ordre $Q_1 \leftarrow Q_2$ où Q_i est ou bien absente ou bien une conjonction de littéraux. Toute variable dans Q_1 ou Q_2 est universellement quantifiée en avant de la résultante.

Définition 2.2. ***Résultante d'une dérivation***

Soit P un programme normal[4] *et $G_0 = \leftarrow G$, une requête normale. Soient $G_0, \ldots, G_n$ une dérivation SLDNF de $P \cup \{G_0\}$ où $G_n = \leftarrow Q$ et $\theta_0, \ldots, \theta_n$ est la séquence des substitutions associées à la dérivation. Soit $\theta = \theta_1\theta_2 \ldots \theta_n$. Alors, la dérivation a une longueur n, la réponse*

[3] L'intégrité ("soundness") d'un programme évalué partiellement en rapport à un programme P et une requête R dans le cadre de la sémantique déclarative (procédurale resp.) signifie que les réponses correctes (bien calculées resp.) de R et P' sont aussi des réponses correctes (bien calculées resp.) pour R et P. La complétude est la contraposée de l'affirmation précédente.

[4] Un ensemble fini d'expressions du premier ordre avec négation.

bien calculée est $G\,|_\theta$ et la résultante a la forme $G\theta \leftarrow Q$. Si G est un atome alors la résultante est une clause normale.

DÉFINITION 2.3. ***Évaluation partielle***

Soit P, un programme normal et A, un atome. Soit T un arbre de dérivation SLDNF pour $P \cup \{\leftarrow A\}$, et soient $\leftarrow G_1, \ldots, \leftarrow G_n$ des requêtes choisies à partir des noeuds non-racines de T telles qu'il existe exactement une requête à partir de chaque branche de T qui ne termine pas par un échec. Soient $\theta_1, \ldots, \theta_n$ des réponses bien calculées des dérivations à partir de $\leftarrow A$ jusqu'à $\leftarrow G_1, \ldots, \leftarrow G_n$ respectivement. Alors l'ensemble des résultantes $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$ est appelé une évaluation partielle de A dans P.

Si A est un ensemble fini d'atomes alors une évaluation partielle de A dans P est une union des évaluations partielles des éléments de A.

Nous passons maintenant aux conditions fondamentales de fermeture et d'indépendance qui ont une portée globale pour établir l'intégrité de l'évaluation partielle.

DÉFINITION 2.4. ***Fermeture***

*Soient S un ensemble d'expressions du premier ordre et A un ensemble fini d'atomes. S est A-**fermé** si chaque atome dans S contenant un symbole de prédicat qui a occurence dans A est une instance d'un atome dans A.*

DÉFINITION 2.5. ***Indépendance***

Soit A un ensemble d'atomes. A est indépendant si aucun des atomes de A pris deux à deux ont une instance commune.

Pour un atome A et un programme P, il existe une infinité d'évaluations partielles possibles de A dans P. Il nous faut donc une règle déterministe pour générer les résultantes. Celle-ci s'appelle la règle du dépliage.

DÉFINITION 2.6. ***Règle du dépliage***

Une règle du dépliage D est une fonction qui, pour un programme P et un atome A, retourne un ensemble fini de résultantes lesquelles forment une évaluation partielle de A dans P. Si A est un ensemble fini d'atomes et P est un programme alors l'ensemble des résultantes obtenues par la règle D en lui appliquant chaque atome de A est appelé une évaluation partielle de A dans P par D.

2.1. Théorème de l'évaluation partielle. Voici le théorème fondamental de l'évaluation partielle prouvé dans [LS91b].

THÉORÈME 2.1. *Soit P un programme normal et A un ensemble indépendant d'atomes. Soit P' une évaluation partielle de A dans P. Pour tous les buts R tels que $P' \cup R$ est A-fermée:*

- *$P \cup R$ a une SLDNF-réfutation avec réponse bien calculée θ si et seulement si $P' \cup R$ a une SLDNF-réfutation avec réponse θ.*
- *$P \cup R$ a un SLDNF-arbre avec échec fini si et seulement si $P' \cup R$ a un SLDBF-arbre avec échec fini.*

Ce théorème a des implications certaines dans la construction des algorithmes corrects d'évaluation partielle. La difficulté d'un algorithme basé sur ce théorème est de calculer un ensemble indépendant d'atomes A tel que P' est une évaluation partielle de A dans P et que $P' \cup R$ soit A-fermée.

2.2. Algorithme de base. Nous présentons maintenant un algorithme naïf d'évaluation partielle, naïf au sens où cet algorithme n'est pas efficace. Il n'est présenté que dans un but pédagogique.

```
ENTREE: Un programme P, une requete R et une regle de depliage D.
SORTIE: Un ensemble d'atomes.

DEBUT

  A[0] := l'ensemble des atomes de R;
  i := 0;

  REPETER
    P' := une evaluation partielle de A[i] dans P utilisant U.
    S := A[i] UNION { p(t) | (B <- Q, p(t), Q') dans P'
                          OU
                          (B <- Q, not(p(t)), Q') dans P' };
    A[i+1] := abstraction(S);
    i := i + 1;
  JUSQU'A A[i] = A[i-1] (modulo un renommage de variables)

FIN
```

FIGURE 4.3. Algorithme de l'évaluation partielle

DÉFINITION 2.7. *Soit S, un ensemble de règles.* `abstraction(`S`)` *est un ensemble indépendant d'atomes A ayant les mêmes prédicats que ceux de S tel que chaque atome de S est une instance d'un atome de A.*

Le rôle de l'opération `abstraction(`S`)` est d'assurer la terminaison de l'algorithme et, en même temps, de satisfaire la condition d'indépendance.

Si l'algorithme termine correctement ses opérations alors il laisse à son arrêt un ensemble indépendant d'atomes `A[i]`. Par construction `A[i]` satisfait la condition suivante. Si P' est une évaluation partielle de `A[i]` dans P utilisant D, $P' \cup R$ est `A[i]`-fermé. Donc, par le théorème d'évaluation partielle, P' est une évaluation partielle correcte de P par rapport à R.

2.3. Problèmes de terminaison. Généralement, on distingue deux types de terminaison de l'algorithme de l'évaluation partielle.

(i) Terminaison de la règle du dépliage D.

(ii) Terminaison globale de l'algorithme itératif (la boucle `REPETER-JUSQU'A`) dans l'algorithme de la figure 4.3.

Les chercheurs ont souvent mis l'emphase sur le premier type de terminaison. Néanmoins, il semble que le deuxième soit plus difficile à respecter car cela concerne la résolution de l'opération `abstraction/1`. Le problème de terminaison du deuxième type est de s'assurer que les ensembles d'atomes générés par la suite `A[1]`, `A[2]`, ... ne croissent pas trop. C'est la raison de la présence de l'opération `abstraction/1` qui a pour fonction de réduire l'ensemble `A[i]`.

EXEMPLE 2.1. *Soit P un programme qui renverse les listes en utilisant un accumulateur.*

```
reverse([], L, L).
reverse([X | Xs], Ys, Zs) :- reverse(Xs, Ys, [X | Zs]).
```

Soit R la requête `<- reverse(_,_,[])`. *Sans l'opération* `abstraction/1`, *l'algorithme d'évaluation partielle génère la suite infinie des atomes suivants.*

```
reverse(_, _, []).
reverse(_, _, [X1]).
reverse(_, _, [X1, X2]).
reverse(_, _, [X1, X2, X3]).
...
```

Proprement dit, l'opération d'abstraction généralise les atomes qui vont apparaître dans le corps des résultantes. Ces atomes peuvent être sélectionnés pour dépliage. En généralisant les atomes, nous sommes certains que le programme résultant de l'évaluation partielle supportera au moins toute requête, instance de requête du programme original. Pour en savoir plus long sur l'opération `abstraction/1`, consultez [Gal93].

2.4. Renommage. Bien plus souvent qu'autrement, la complexité d'exécution d'un programme logique se calcule par le nombre d'inférences logiques. Par contre, une utilisation efficace des structures apparaissant dans les prédicats a aussi son importance au niveau du temps d'exécution. Une optimisation significative peut être obtenue simplement par la spécialisation des

structures des prédicats [GB90]. Il s'agit de transformer les clauses en enlevant les paramètres des prédicats devenus inutiles. Plusieurs évaluateurs partiels pour Prolog ont déjà incorporés le concept de spécialisation de structures avec l'évaluation partielle. Il suffit de consulter [Fuj87], [SS86a], [Sah91] et [Gal91] pour s'en rendre compte. L'incorporation de cette spécialisation de structures est relativement simple. Pour en savoir plus sur ce sujet, consulter [Gal93].

3. Considérations pratiques

Des considérations pratiques sont requises pour l'implémentation d'un évaluateur partiel. Voici la première et la plus importante.

- L'exécution de $P' \cup \{R\}$ est plus efficace que celle de $P \cup \{R\}$.

Cette condition n'est assurée par aucun algorithme d'évaluation partielle. Il y a plusieurs aspects à son respect. Voici les principaux.

- **Contrôle**: La règle du dépliage ne doit pas encourager la création de points de choix aboutissant à l'échec. Ces points de choix inutiles augmentent le temps de recherche dans l'espace des solutions de façon significative.
- **Espace**: La règle du dépliage tend à propager les structures de données en effectuant certaines substitutions. Cette propagation peut avoir de désastreuses conséquences sur l'espace utilisé à l'exécution et a fortiori sur le temps de calcul du programme évalué partiellement. Il est donc préférable d'enlever les structures de données redondantes ou devenues inutiles du programme original en effectuant une spécialisation de structures telle que mentionnée dans la section 2.4.
- **Indexation**: La spécialisation par l'évaluation partielle ne devrait pas éliminer les informations servant à l'indexation du programme. Cette indexation sert à augmenter la vitesse de l'algorithme d'unification. Il a été suggéré d'ajouter certaines techniques bien précises telles que la construction de disjonction pour aider à garder les indexations du programme original (voir [Sah91]).

4. Programmation logique versus Prolog

Un nombre considérable de recherches [VD88] [Sah90, Sah91] ont été faites sur l'évaluation partielle en Prolog de manière à supporter les principaux outils non-logiques ou méta-logiques tels que le coupe-choix, `var/1`, `assert/1`, `retract/1`, les prédicats d'entrées/sorties et d'autres.

Et ce, en même temps que les recherches sur l'évaluation partielle en programmation logique pure[5].

Bien que ces recherches sont importantes car il s'est écrit à l'heure actuelle d'innombrables programmes en Prolog, les résultats restent mitigés. Plusieurs difficultés de l'évaluation partielle en Prolog en entier sont synthétisées dans [Owe89] dans lequel Owen déduit non moins de 16 règles supplémentaires (techniques de spécialisation) nécessaires en plus de la règle du dépliage pour gérer correctement les manipulations des prédicats méta-logiques de Prolog au cours de l'évaluation partielle. Bien que plusieurs de ces problèmes ont été réglés ([BR89], [CW89], [LS91a], [LS88] et [Sah90, Sah91]), encore trop de problèmes sont restés non-résolus. Les études sur le traitement du coupe-choix ont suggéré l'introduction de plusieurs nouvelles règles de transformation dont, entre autres, une règle nommée **pliage** qui a l'effet d'assembler une séquence de conjonctions et de créer une nouvelle clause pour leur traitement ([GS89], [KF86], [PP91], [TS83], [TS86] et [Sek93]). L'introduction de cette nouvelle procédure oblige l'ajout d'une stratégie de sélection pliage/dépliage efficace pour rendre le programme de l'évaluation partielle déterministe. Cette stratégie n'est pas évidente à concevoir. C'est principalement dans les années 90 qu'il s'est développé de telles stratégies de sélection (voir [Sat90], [Pre90] ou [Sah91]). Avant 1989, il n'existait pas de moyen déterministe sûr de choisir dynamiquement entre la règle du dépliage ou du pliage [vH89]. Ce n'est que tout récemment qu'une stratégie de sélection a été montrée très efficace (voir [Pre93]).

Dans [Lak90], on argumente que l'ajout de plusieurs nouvelles règles de transformation va à l'encontre des besoins significatifs de *simplicité* de l'algorithme de transformation, *d'extensibilité* (il n'est pas évident que l'ajout des règles déterminées ne formeront pas un système complet et suffisamment efficace comparable à d'autres transformations plus spécialisées) et de *méta-circularité* (application de l'évaluation partielle sur l'évaluateur partiel lui-même).

5. Au delà de l'évaluation partielle

L'évaluation partielle ou la déduction partielle ont plusieurs limitations. Des solutions sont envisageables pour étendre les forces de celles-ci. Comme nous l'avons vu dans la section précédente, ce sont l'utilisation de nouvelles règles de transformation ou d'analyse. Néanmoins, au lieu d'amalgamer plusieurs des règles de transformation dans un seul programme, Lakhotia ([Lak90]) propose plutôt de "découper" le procédé de transformation ou de spécialisation en plusieurs systèmes indépendants, chacun spécialisé dans une ou quelques types de transformation. L'un d'entre ces systèmes serait l'évaluation partielle ou peut-être plutôt, la déduction

[5]Rappelons que plusieurs chercheurs préfèrent nommer l'évaluation partielle en programmation logique pure, la déduction partielle pour la distinguer de l'évaluation partielle en Prolog avec tous ses outils.

partielle. Celle-ci n'utiliserait qu'une ou deux règles de transformation dont la principale serait le dépliage (probablement, la seconde serait la règle du pliage). Plus récemment, dans [Gal92], on retrouve une idée similaire. Gallagher propose que la spécialisation de programmes logiques devrait être regardée comme une application parmi d'autres applications d'analyses de programmes logiques, par exemple, l'inférence de types.

Suivant ces idées, dans [Gal93], l'auteur propose que les termes déduction partielle ou évaluation partielle soient remplacés par le terme plus général de **spécialisation** de programmes logiques. Ce dernier porte moins une connotation de mécanisme opérationnel et on accepte mieux que ce terme représente d'autres règles de transformation que le dépliage (ou le pliage) plus attribuables au terme évaluation partielle.

En tout cas, il est clair que l'algorithme d'évaluation partielle doit obligatoirement avoir une stratégie de contrôle du dépliage pour assurer la terminaison de l'évaluation partielle. Cette stratégie peut se faire de deux façons différentes [Jon88]: dynamiquement au cours de l'exécution à l'aide d'une stratégie de détection de boucles sans fin ou bien à l'aide d'analyse statique du programme [par exemple, une analyse des temps de liaison ("binding time analysis")] qui informe l'évaluateur partiel du contrôle du dépliage en lui fournissant des **annotations** (sortie de l'analyseur statique) pré-calculées concernant le programme. Dans le premier cas, on parle de **stratégies de contrôle en direct** ("online control strategies") et dans le deuxième, de **stratégies de contrôle en différé** ("offline control strategies").

Dans la plupart des implémentations actuelles, on incorpore plutôt une stratégie de contrôle en direct. Le principe du fonctionnement est simple: il s'agit de maintenir à jour une structure de tous les buts antérieurs gérés au cours du dépliage. Des exemples de stratégies de ce type se retrouvent dans [SB86], [FA88], [LS88], [BL89], [Sah90], [PP90] [BdSM91] et [Pre93]. Les stratégies en direct sont plus puissantes car elles ont accès à de l'information qui ne peut pas toujours être inférée par les analyses statiques. Par contre, la quantité d'information relative à la détection de boucles sans fin peut croître indéfiniment ce qui peut, pour de gros programmes, rendre l'évaluation partielle extrêmement lente [Pre93]. C'est pour cette raison que si une stratégie en direct est adoptée, il faut que l'analyse effectuée pour le contrôle soit *indépendante* de la quantité d'information accumulée au cours du calcul. Cette exigence n'est pas facile à respecter mais elle reste raisonnable. Cela a été réussi dans [Pre93]. Par contre, il est pratiquement impensable d'obtenir la méta-circularité de l'évaluateur partiel avec une stratégie en direct. C'est pour ces raisons qui certains chercheurs se sont intéressés aux stratégies en différé. Un cadre théorique très prometteur pour la formalisation des stratégies en différé est **l'interprétation abstraite** présentée pour la première fois dans [CC77] et reprise pour les langages déclaratifs

dans [Abr87]. La relation entre évaluation partielle et interprétation abstraite a été étudiée dans [Gal86], [Fuj88], [GCS88] et, plus récemment, dans [GB91] et dans [Bou92].

Notons aussi que l'interprétation abstraite a récemment été utilisée pour développer un outil d'analyse de programmes pour l'élimination des clauses inutiles. Cet outil a son intérêt certain en relation à l'évaluation partielle car cette dernière génère occasionnellement des résultats qui peuvent être significativement optimisés par l'élimination des clauses inutiles. On en discute dans [dWG92a] et dans [Gal93].

6. Méta-programmation et évaluation partielle

Le problème fondamental de la méta-programmation a toujours été en rapport à la représentation des programmes-objets, expressions et termes dans le langage. Deux principales approches de représentation ont été identifiées: **représentation close** ("ground représentation"), aussi appelé représentation de Gödel, et **représentation non-close**. La différence entre ces représentations se situe au niveau de la représentation des variables. Dans la représentation non-close, les variables sont représentées par des variables libres au méta-niveau tandis que, dans le représentation close, les variables sont représentées par des termes clos au méta-niveau.

La représentation de Gödel en méta-programmation est apparue pour la première fois en programmation logique dans [Bow82]. Elle a été occasionnellement mentionnée dans la littérature (par exemple dans [Esh86]). Les bases théoriques dans le cadre de la méta-programmation ont été développées dans [HL88b] et dans [HL88a]. Dans [HL88a], les différences entre les deux représentations sont discutées et il est montré que la représentation close est la plus puissante des deux. La représentation non-close souffre de plusieurs problèmes sémantiques lesquels disparaissent par l'utilisation de la représentation close. Justement, certains de ces problèmes sont aussi des obstacles au bon ménage entre méta-programmation et évaluation partielle. La représentation close est donc la bienvenue dans l'univers de la transformation de programmes telle que l'évaluation partielle [Gal93]. Néanmoins, bien des problèmes d'efficacité d'exécution sont en relation à l'utilisation de la représentation close. En conséquence, un nouveau langage de programmation logique nommé Gödel a été créé [HL92] pour offrir un support plus adéquat au développement d'outils et de méthodologies pour la méta-programmation utilisant la représentation de Gödel. Déjà, un évaluateur partiel nommé *SAGE* ([Gur94b] et [Gur94a]) a été conçu pour le langage Gödel. Sa réalisation démontre la puissance de l'utilisation de la représentation de Gödel en évaluation partielle. Il est montré que *SAGE* peut optimiser des méta-programmes de façon à ce que leurs temps d'exécution soient potentiellement équivalent aux temps d'exécution des programmes-objets qu'ils manipulent sans méta-interprétation. De plus,

SAGE est méta-circulaire c'est-à-dire qu'étant donné qu'il est lui-même un méta-programme écrit en Gödel, il est possible d'appliquer l'évaluation partielle sur lui-même. De cette manière, il est possible de spécialiser l'évaluateur partiel par rapport à un méta-interprète quelconque de façon à obtenir un compilateur. L'évaluation partielle de *SAGE* par rapport à lui-même retourne ce qu'on peut appeler un générateur de compilateurs.

7. Comparaisons

L'évaluation partielle en programmation logique est comparable aux techniques de même nom développées dans d'autres paradigmes de programmation dont la programmation fonctionnelle. Néanmoins, plusieurs différences significatives la séparent des autres.

- Le non-déterminisme inhérent des programmes logiques offre une grande flexibilité au mécanisme du dépliage. Ce mécanisme n'est pas généralement présent en programmation fonctionnelle ou impérative [Kom92] [Gal93].
- En programmation fonctionnelle, une distinction entre information statique et information dynamique s'est faite dans le but de créer des outils d'aide (analyse statique) pour alléger les évaluateurs partiels de leur travail. Néanmoins, cette distinction est plus difficile à faire en programmation logique en contraste aux langages fonctionnels. Au cours de l'évaluation partielle, l'information est propagée vers l'avant et vers l'arrière par l'application des substitutions. L'évaluation partielle peut donc modifier autant les structures de données des entrées que celles des sorties de la requête spécifiée. De ce fait, il est plus difficile de spécifier qu'un paramètre restera une information statique ou dynamique par une simple analyse statique du programme.
- Le mariage entre la méta-programmation et l'évaluation partielle est plus significatif en programmation logique que pour les autres paradigmes car il y a plusieurs manières non-traditionnelles d'interpréter un programme logique (exemples: recherche vers l'avant ou vers l'arrière, en profondeur ou en largeur, co-routine, etc.) qui peuvent s'exprimer sous forme de méta-programmes et être optimisées par l'évaluation partielle. Cela a de grands potentiels en intelligence artificielle.

Ceux qui aimeraient en connaître plus sur ce qui se fait en évaluation partielle en programmation fonctionnelle et impérative peuvent consulter le livre de Jones, Gomard et Sestoft [dJGS93]. Dans ce dernier article, tout un chapitre traite de l'évaluation partielle en Prolog pur. Un aperçu plus technique se trouve dans [CD93].

8. Notes historiques

L'évaluation partielle a son origine la plus lointaine en rapport au théorème de Kleene S_n^m ([Kle52, Théorème XXIII, p. 342]) dans le cadre de la théorie des fonctions récursives. Elle a été introduite en informatique par Lombardi et Raphael en 1964 [LR64][Lom67] et, plus officiellement, par Futamura en 1971. Dans son article [Fut71], Futamura met en évidence la relation entre interprétation et compilation. Il propose l'utilisation de l'évaluation partielle en tant que compilation, génération de compilateurs et compilation de compilateurs[6]. Les deux dernières projections sont beaucoup plus difficiles à réaliser étant donné que l'évaluateur partiel (le programme qui évalue partiellement) doit pouvoir s'appliquer sur lui-même. L'évaluation partielle fut étudiée intensément par la suite en Suisse ([B+76]) et en Russie ([Ers80]) dans les années 70. En 1977, l'expression anglaise "mixed computation" pour désigner l'évaluation partielle est introduite pour la première fois [Ers77]. En 1976, l'un des premiers évaluateurs partiels nommé REDFUN a vu le jour pour un sous-ensemble suffisamment large de LISP ([B+76][Har77, Har78]). Ces recherches dans les années 70 ont attiré l'attention de grands nombres de chercheurs dans les années 80. L'activité de recherche en évaluation partielle s'est alors répandue très rapidement après 1976. La plupart des premiers résultats sérieux ont été en programmation fonctionnelle et en programmation impérative.

En programmation logique, l'évaluation partielle fut introduite en 1981 par Komorowski ([Kom81][Kom82]). C'est ce chercheur qui a introduit le terme plus approprié de déduction partielle[7]. Venken ([Ven84]) semble être le premier à avoir écrit un évaluateur partiel pour Prolog et il l'a appliqué à l'optimisation de requêtes de bases de données déductives. L'importance de l'évaluation partielle en tant que technique d'optimisation en programmation logique n'a été reconnu qu'en 1986 avec [Gal86] et plusieurs chercheurs se sont mis à s'y intéresser dont, entre autres, [BL89], [CW89], [CvS89], [FA88] et [Kur88], et plus spécialement en méta-programmation dans [SS86a], [TF86], [Tak86] [SB86, SB89], [FF88], [LS88] et d'autres. Mentionnons que dans [KC84], il est montré qu'un compilateur de Prolog vers Lisp peut être produit à l'aide de l'évaluation partielle d'un interprète Prolog écrit en Lisp (l'évaluation partielle se fait là en Lisp et non en Prolog). [8]

---

[6] Dans la littérature, on fait référence à ces projections sous le nom de **projections de Futamura**. La première projection est la compilation, la deuxième, la génération de compilateurs et la troisième, la compilation de compilateurs.

[7] Une bonne introduction à la déduction partielle se retrouve dans [Kom92].

[8] Plus récemment, dans [CK91], un interprète de Prolog écrit en Scheme a été conçu. Quelques essais de compilation à l'aide de l'évaluation partielle ont été tentés. Celle-ci diminue seulement d'un facteur six le temps d'interprétation de programmes Prolog. Ce n'est pas beaucoup car, par exemple, dans [CD91], pour des programmes comparables au langage Algol écrit en Scheme, l'évaluation partielle diminuait d'un facteur de plus de 100 le temps d'interprétation. Les auteurs, Consel et Khoo, de [CK91] expliquent ces pauvres résultats dans le cas de programmes Prolog par le fait que le processus d'unification ne peut pas être grandement optimisé

Les deux principales directions de recherche concernant la combinaison de la méta-programmation et de l'évaluation partielle sont la *compilation* d'interprètes non-standards tels que les interprètes "co-routines" et les *extensions* tels que les coquilles de systèmes experts et les outils de déverminage.

Le développement de méthodologie de programmation et le rôle de l'évaluation partielle ont été étudiés dans [Lak89] et [Kom89].

Les bases théoriques de l'évaluation partielle en programmation logique sont présentées dans [LS91b]. Les auteurs, Shepherdson et Lloyd ont clairement identifié deux conditions nécessaires (condition d'indépendance et de fermeture) pour le respect de l'intégrité ("correctness") de l'évaluation partielle. Un algorithme basé sur ces conditions a été conçu dans [BL90].

Enfin, la réalisation de la deuxième et la troisième projection de Futamura (méta-circularité de l'évaluation partielle) a pour la première fois été obtenue en 1985 en Lisp par Jones et d'autres ([JSS85]). Plusieurs des notions et des techniques utilisées par ce système se retrouvent encore aujourd'hui dans les évaluateurs partiels méta-circulaires actuels. En Prolog, les résultats de méta-circularité sont beaucoup moins encourageants. Les principaux problèmes viennent de l'utilisation des prédicats non-logiques rendant la sémantique de Prolog impure ([Owe89]). L'algorithme de l'évaluation partielle est rendu trop sophistiqué par le traitement de ces prédicats pour que la méta-circularité soit envisageable ([Sah91] [VD88]). Les premières tentatives d'implémentation d'évaluateurs partiels méta-circulaires en Prolog se retrouvent dans [FF88] et [FA88]. Pour éviter d'alourdir trop l'algorithme de contrôle de l'évaluateur partiel, les prédicats non-logiques de Prolog ne sont pas traités. Plus récemment, Logimix ([MB93]) a été conçu spécifiquement pour l'étude de l'optimisation par la méta-circularité. Contrairement aux autres, ce système utilise partiellement la représentation de Gödel telle qu'expliqué dans la section précédente. L'utilisation de cette représentation permet de distinguer plus clairement les différents niveaux d'interprétation (par exemple, l'évaluateur partiel et le méta-interprète) facilitant la méta-circularité. Les résultats d'efficacité de ce dernier système sont remarquables à comparer aux deux précédents. Dernièrement, un système appelé SAGE a été conçu dans le langage Gödel [Gur94a]. Ce système a la particularité d'utiliser la représentation de Gödel

---

par l'évaluation partielle car il se fait trop souvent d'appels d'unification d'informations dynamiques (qui ne peuvent donc pas être éliminées). Ils ajoutent que les informations statiques de programmes Prolog ne sont pas aussi considérables que pour d'autres langages. Pour remédier au problème, les auteurs de [CK91] disent qu'il faudrait introduire d'autres transformations de programmes. Leurs recherches démontrent que l'optimisation du processus d'unification est au centre du problème d'optimisation de programmes Prolog. Nous soupçonnons qu'ils n'ont pas suffisamment analysé le problème de l'optimisation du mécanisme d'unification. Donc, il ne semble pas faux de croire que les recherches de l'évaluation partielle en représentation de Gödel peuvent possiblement résoudre certains problèmes en rapport à l'optimisation de l'unification similaire à ceux rapportés dans l'article de Consel et Khoo car, là aussi, en représentation de Gödel, le problème d'optimisation du processus d'unification est central.

complète. De cette manière, bien des problèmes rapportés dans [Owe89] sont éliminés. Contrairement à Logimix, ce système est complètement automatique. Les résultats d'optimisation sont remarquables autant au niveau de la première projection de Futamura que pour les deux autres projections[Gur94b].

Pour de plus amples références, voir [SS88] et [SZ88] qui contiennent plus de 250 références sur l'évaluation partielle.[9]

9. Évaluateurs partiels opérationnels et disponibles

Plusieurs évaluateurs partiels ont été développés pour l'expérimentation [LS88] [Fuj87] [Kur88] [TF86] [Gal91] [Pre92] [LS91a] [Sah91] [MB93]. L'évaluateur partiel le plus distribué et peut-être le plus connu depuis le début des années 90 se nomme Mixtus [Sah90], supporte tout le langage Prolog et donne encore aujourd'hui de très bons résultats [Pre93] comparativement aux autres systèmes malgré son âge avancé (4 ans). Il a été écrit par Sahlin dans le cadre de sa thèse de doctorat [Sah91]. Mixtus a l'avantage d'être complètement automatique et d'utiliser une stratégie de contrôle en direct. Par contre, son implémentation n'est que très peu documentée et il est pratiquement impossible de le modifier sans l'aide du concepteur. Il a aussi le grand désavantage de fonctionner dans un temps pseudo-quadratique[10] en fonction de la taille du programme [Pre93].

Nous avons eu l'occasion d'expérimenter le système ProMiX qui a été écrit par Lakhotia en 1989 [LS91a]. Ce système traite aussi tout le langage Prolog. Dans la version d'origine, il utilisait une stratégie de contrôle en direct complètement déterminée par l'ajout de l'usager d'annotations (difficiles à spécifier [LS90]). Dans une version qui a suivi (1991), une stratégie de contrôle en direct automatique a été incorporée à la précédente. Bien qu'il ne donne pas toujours de bons résultats comparativement à Mixtus et d'autres [Lam89], son implémentation est bien documentée et facile à modifier. Nous l'avons expérimenté longuement ([MD93]) et nous avons décelé quelques erreurs de traitement de certaines structures de contrôle. Nous ne nous sommes attardés à la correction et la modification de l'implémentation que quelques mois avant d'abandonner définitivement son utilisation pour mieux se concentrer sur nos expérimentations en Mixtus.

---

[9] Il est aussi possible d'accéder par Internet à un fichier dans le format BibTeX qui est une bibliographie mise à jour régulièrement par Peter Sestoft. Ce fichier se situe à l'adresse électronique **ftp.diku.dk** sous le nom **/pub/doc/partial-eval.bib.Z**.

[10] Ce terme est utilisé dans [Pre93] pour signifier que la complexité n'est pas théoriquement prouvée quadratique mais qu'elle se comporte comme telle en pratique.

Gallagher a conçu un évaluateur partiel nommé SP [Gal91] en 1991 pour un sous-ensemble pur de Prolog. Ce système, après examen, semblait être un bon candidat pour nos expérimentations. Néanmoins, il ne traite aucune structure de contrôle de Prolog même pas la structure IF-THEN-ELSE. Nous ne pouvions donc pas nous servir de celui-ci sans modifier la majeure partie des implémentations de nos langages réflexifs. Nous l'avons donc abandonné.

Seul le système Logimix de Mogensen et Bondorf [MB93] utilise une stratégie de contrôle du dépliage en différé. Il est donc très efficace en temps de calcul et aussi du point de vue de l'optimisation. Sa particularité première est d'être méta-circulaire. Il est donc possible de générer des compilateurs et des générateurs de compilateurs avec des facteurs d'optimisation approximatifs de 56% et 26% respectivement. Nous avons pris le temps d'étudier rapidement son fonctionnement et nous nous sommes rendu compte que puisque les concepteurs n'ont pas automatisé la génération des annotations nécessaires pour le contrôle du dépliage (par exemple en développant un programme d'analyse des temps de liaison), il n'était pas raisonnable de faire des expérimentations sérieuses avec ce système. En effet, l'écriture des annotations doit se faire à la main pour tous les programmes à évaluer partiellement. L'écriture de celles-ci exige beaucoup de temps et beaucoup trop d'attention.

En fait, il nous aurait fallu le système PADDY de Prestwich [Pre92] mais nous n'avions pas connaissance de son existence au moment de nos recherches et de nos expérimentations. Ce système évalue partiellement efficacement tout Prolog et utilise une stratégie de contrôle en direct. Il a la particularité de fonctionner dans un temps pseudo-linéaire en fonction de la taille du programme en paramètre [Pre93]. Comme nous le verrons dans le chapitre qui va suivre, nous avons eu plusieurs problèmes avec Mixtus causés par son temps d'exécution pseudo-quadratique. Dans certains cas, ceci a eu l'effet de faire exploser le temps de calcul de l'évaluation partielle. Il aurait été intéressant de réaliser les mêmes tests avec PADDY.

Nous nous sommes donc contenté d'utiliser seulement Mixtus[11] pour tenter de démontrer que l'évaluation partielle peut optimiser les programmes réflexifs. Bien que le choix de n'utiliser qu'un seul système peut sembler contraignant vu la multitude des systèmes expérimentaux disponibles, ce choix reste acceptable considérant le fait que nous n'étions pas intéressés principalement à l'obtention d'une évaluation partielle rapide mais plutôt à l'obtention d'un bonne optimisation.

[11] Nous avons utilisé Mixtus avec le système Sicstus Prolog version 0.7 #4 qui date de 1989 et non pas une version moderne de Quintus Prolog. La raison est simple: nous avons décelé des erreurs d'exécution au cours de nos expérimentations de Mixtus sous Quintus Prolog.

10. Évaluation partielle et réflexion

La relation entre la réflexion et l'évaluation partielle a été pour la première fois établie dans [CFL+88] en 1986. Par la suite, dans [TW88], les problèmes entre l'évaluation partielle et les principes de réflexion de Weyhrauch ont été analysés.

Ce n'est qu'en 1988 que le lien entre la réflexion par interprétation *à la 3-Lisp* et l'évaluation partielle a été profondément analysé. Les résultats ont été publiés sous la référence [Dan88]. Dans cet article, l'auteur, Danvy, montre que l'évaluation partielle peut éliminer tous les niveaux d'interprétation dans le cas de méta-interprètes non-réflexifs. Néanmoins, il est moins évident qu'elle peut faire de même dans le cas de méta-interprètes réflexifs. En effet, étant donné que les programmes réflexifs ont le pouvoir de briser la hiérarchie d'interprétations de la tour réflexive, l'évaluation partielle est incapable d'éliminer les niveaux d'interprétation dans la plupart des cas. Certaines solutions sont envisagées:

- Évaluer partiellement la tour réflexive itérativement sur chacun des niveaux d'interprétation en fonction d'un programme réflexif.
- Évaluer partiellement la tour réflexif d'une seule passe.
- Établir une communication entre l'évaluateur partiel et la tour réflexive de manière à ce qu'à chaque appel réflexif, l'évaluateur partiel soit informé de la manière dont il doit évaluer partiellement l'appel en question. On en discute aussi dans [dJGS93].

Danvy reconnaît qu'il reste encore beaucoup à découvrir de la relation théorique et pratique entre la réflexion et l'évaluation partielle.

En programmation logique, une telle relation a été étudié dans [BCL+88]. Malheureusement, dans cette recherche, c'est plutôt une étude comparative entre l'introspection directe (réflexion très limitée) et la méta-interprétation (optimisée par l'évaluation partielle) qui est tentée. Les auteurs ont essayé de montrer que l'introspection directe est plus avantageuse que la méta-interprétation aidée de l'évaluation partielle. Ces recherches sont relativement éloignées des nôtres.

Enfin récemment, dans [MD93], nous avons communiqué nos premiers résultats de recherche de l'application de l'évaluation partielle en réflexion de comportement *à la 3-Lisp*. Nous avons montré que, pour des exemples simples, il est possible d'éliminer tous les niveaux d'interprétation de la tour réflexive par l'utilisation de l'évaluation partielle. Pour démontrer cela, nous avons défini un méta-interprète réflexif simple pour un sous-ensemble de Prolog. Nous avons présenté deux exemples de programmes réflexifs simples et, nous les avons optimisé par l'application de l'évaluateur partiel ProMiX [LS91a] en utilisant des annotations pour le

contrôle du dépliage [LS90]. Quelques mois plus tard, nous obtenions des résultats similaires avec l'utilisation de l'évaluateur partiel automatique Mixtus de Sahlin [Sah90] sans aucune annotation.

11. Conclusion

Nous avons pris la peine de présenter toutes les directions de recherche en rapport direct ou indirect à l'évaluation partielle parce qu'il n'existe pas à l'heure actuelle de système démontré efficace, complet et opérationnel dont nous aurions pu nous servir sans devoir comprendre son fonctionnement. La communauté de chercheurs en évaluation partielle en programmation logique n'a pas encore réussi à automatiser suffisamment l'évaluation partielle pour le développement d'un tel système. Au moment où nous avons commencé nos recherches, il n'existait qu'un seul système implémenté, complet et opérationnel (mais très inefficace) pour le langage Prolog [Sah90]. D'après nos recherches, il n'existe qu'un seul article ([Pre93]) dans toute l'histoire de l'évaluation partielle en programmation logique qui discute spécifiquement du problème de la complexité des algorithmes d'évaluation partielle des implémentations actuelles. De plus, aucune méthodologie reconnue de méta-programmation en vue de l'optimisation par l'évaluation partielle ne s'est développée.

Nous avons donc cru bon de présenter et d'expliquer toutes les directions de recherche de façon à bien montrer au lecteur qu'il est encore raisonnable de penser réaliser concrètement un évaluateur partiel efficace pour des langages déclaratifs dans un avenir rapproché de pair avec le développement d'une méthodologie de méta-programmation spécifiquement conçue pour l'optimisation efficace par l'évaluation partielle.

Dans le chapitre suivant, nous présentons nos résultats de l'application de l'évaluateur partiel Mixtus sur des exemples de programmes réflexifs écrits dans nos différents langages réflexifs.

CHAPITRE 5

Spécialisation et réflexion

Dans le chapitre précédent, nous avons présenté un compte-rendu des recherches et des résultats relatifs à l'évaluation partielle en programmation logique. Nous avons fait une recherche bibliographique approfondie pour déterminer les tendances antérieures et actuelles. Cela nous a permis de développer une façon particulière d'analyser nos résultats d'expérimentation. Nous tenterons au cours de ce chapitre d'appliquer et d'expliquer cette attitude intellectuelle.

Pour les expérimentations présentées dans ce chapitre, nous avons utilisé exclusivement l'évaluateur partiel Mixtus (qui a adopté une stratégie en direct) car il n'a pas été possible de mettre la main sur d'autres systèmes suffisamment complets, automatiques et performants que le système Mixtus. Nous avions commençé nos expérimentations avec ProMiX dans [MD93] mais nous nous sommes rendu compte de ses lacunes importantes. Nous l'avons donc abandonné.

Nous ne nous sommes pas intéressé directement aux problèmes pratiques du contrôle du dépliage, du contrôle de la taille du programme généré, de la préservation de l'indexation et du temps de calcul de l'évaluation partielle. Indirectement, bien entendu, ces problèmes ont une certaine influence sur notre considération principale: l'optimisation. L'utilisation d'un évaluateur partiel automatique nous a permis d'éviter de nous attarder aux problèmes précédents délibérément mis de côté.

Dans le chapitre précédent, nous avons mis en évidence les obstacles encore très contraignants de l'évaluation partielle dans le langage Prolog en sa totalité. Nous avons pu montrer les tendances actuelles de rationalisation du langage Prolog et nous tentons de les suivre le plus justement possible. Nous ne nous sommes pas intéressé à l'évaluation partielle des programmes avec coupe-choix ou avec la plupart des prédicats méta-logiques n'ayant pas de sémantique déclarative dans le langage.

L'intérêt principal des expérimentations est de mesurer le gain d'efficacité obtenu par l'évaluation partielle. En conséquence, pour la plupart des tests d'évaluation partielle, nous

présenterons le code du résultat de l'évaluation partielle et, à l'aide de celui-ci, nous mesurerons *qualitativement* les gains obtenus. Une mesure qualitative est une analyse du contenu du code du résultat dans le but de déceler les changements qui ont l'effet d'augmenter l'efficacité de l'exécution. Pour quelques exemples seulement, nous mesurerons quantitativement le gain d'optimisation en calculant le nombre de réductions (le nombre de déplacements vers l'avant dans l'arbre de recherche) pour certaines requêtes sur les programmes, avant et après évaluation partielle.

Comprenons bien que nous estimons qu'il est préférable d'utiliser une mesure qualitative plutôt que quantitative. En effet, la mesure quantitative pourrait laisser croire qu'il y a une faible dépendance entre l'évaluation partielle et les algorithmes des programmes évalués partiellement. Au contraire, il est important de comprendre qu'une dépendance forte existe et qu'une analyse qualitative d'optimisation est indispensable pour déterminer les forces et les limites de l'évaluation partielle.

Relativement peu de programmes-objets font partie de nos expérimentations. En fait, il n'est pas nécessaire d'appliquer l'évaluation partielle sur de nombreux programmes-objets différents pour un même méta-programme quand une mesure qualitative est utilisée. Les résultats qualitatifs ont l'effet de se répéter pour chaque programme-objet pour un même méta-programme. Cette idée d'indépendance entre le programme interprété et le méta-programme est mentionnée aussi dans [LS91a].

Pour chaque méta-programme mis à l'étude, nous nous sommes demandé ce que nous espérions obtenir comme résultat d'optimisation *avant* d'appliquer l'évaluation partielle. Cette démarche est importante. En effet, si l'estimation du gain espéré n'est aucunement similaire au gain réel, il est très probable que nous nous trouvions devant une faiblesse de l'évaluation partielle sur ce méta-programme.

Enfin, précisons que nous nous sommes intéressé à l'optimisation de tours réflexives d'une seule couche d'interprétation. Nous avons constaté que les difficultés de l'évaluation partielle se situaient d'abord au niveau de la tentative d'élimination de la première couche d'interprétation de la tour réflexive. Les couches supplémentaires ne nous ont apparemment pas causé de difficultés autres que le ralentissement considérable de l'évaluateur partiel. Nous avons fait quelques expérimentations d'évaluation partielle de plusieurs couches d'interprétation sans problème. Nous croyons que les problèmes principaux de l'optimisation par l'évaluation partielle se manifestent premièrement dans le cas d'une seule couche d'interprétation. Mentionnons que dans [MD93] et dans [Dan88], l'évaluation partielle de plusieurs couches d'interprétation est étudiée dans le cadre de la réflexion.

1. Méta-interprétation et évaluation partielle

Nous allons examiner le comportement de l'évaluation partielle sur nos langages réflexifs développés au chapitre 3.

Nous commençons le coeur de ce chapitre par une analyse de l'évaluation partielle de l'algorithme de résolution utilisée dans nos méta-interprètes réflexifs. Ensuite, nous illustrerons des résultats importants concernant la compilation et la spécialisation par l'évaluation partielle de nos langages sur différents programmes réflexifs. Pour les besoins de l'exposé, nous nous sommes donné des définitions bien spécifiques des termes compilation et spécialisation de la méta-programmation. La **compilation** se fait par l'évaluation partielle du méta-interprète quand *seul* (et en sa totalité) le programme-objet est connu du méta-interprète; la ou les requêtes (du programme-objet) sont inconnues.[1] La **spécialisation** est aussi une transformation par l'évaluation partielle mais, cette fois-là, avec le programme *et* la ou les requêtes connues. Nous considérons aussi que les définitions des prédicats réflexifs seront toujours connues à l'évaluation partielle. Il est relativement peu intéressant de compiler ou de spécialiser un méta-interprète quand les définitions réflexives sont inconnues.

Dans [LS90] et repris dans [Gur94a], il est montré que la méta-interprétation se divise en deux parties distinctes: l'**analyse** ("parsing") et l'**exécution**. Dans le cas où le programme-objet est connu à l'évaluation partielle, la première partie, grandement dépendante du programme, a suffisamment d'informations statiques pour être optimisée significativement. La deuxième partie, plutôt dépendante de la représentation du but et de la résolvante, ne peut être optimisée que si le but (-objet) est connu.

La compilation et la spécialisation sont les deux grands modes d'utilisation de l'évaluation partielle en méta-programmation. Par la compilation, on cherche à éliminer la plus grande partie de l'analyse du méta-interprète et par la spécialisation, on cherche à éliminer, en plus de l'analyse, une partie de l'exécution.

2. Optimisation et spécialisation de la résolution

Rappelons que nous nous sommes attardé à cinq opérations comportementales de l'algorithme de résolution: la stratégie de sélection d'un but dans la résolvante, la sélection d'une clause dans le programme, l'exemplarisation de la clause, l'algorithme d'unification et la création

[1] Prendre note que la notion de compilation telle que nous la concevons dans ce chapitre n'existe pas dans la littérature sur l'évaluation partielle de méta-interprètes traditionnels en Prolog. En effet, à notre connaissance, il n'existe pas d'évaluateur partiel qui traite le prédicat méta-logique `clause/2` de façon à intégrer toutes les clauses de la base de règles Prolog à l'évaluation partielle lors d'une telle compilation (le but interprété n'ayant pas été spécifié). Il y a seulement le système PADDY [Pre93] qui propose de remplacer le prédicat `clause/2` par un prédicat spécifique aux programmes-objets interprétés (qui serait défini par l'ensemble des faits dont chacun contiendrait une clause du programme).

des points de choix. Selon le degré de réflexion du langage réflexif, ces opérations comportementales deviennent plus ou moins complexes à implémenter en programmation logique. Il est important de comprendre comment elles doivent être écrites de façon à être éventuellement optimisées par l'évaluation partielle. En effet, ce n'est pas toutes les implémentations correctes qui peuvent donner de bons résultats à l'évaluation partielle.

2.1. Stratégie de sélection la plus à gauche. Dans [Gur94a], Gurr a montré qu'il existe plusieurs algorithmes de sélection du but la plus à gauche dans la résolvante. Il distingue deux types d'algorithme: l'algorithme orienté-succès et l'algorithme orienté-échec. L'algorithme orienté-succès se différencie de l'autre par le fait qu'il n'échouera jamais quelle que soit la structure de la résolvante (syntaxiquement correcte bien entendu). Il retournera plutôt un prédicat pré-défini `vide/0` indiquant qu'aucun but n'a pu être choisi. L'algorithme orienté-échec échouera quand il est impossible de choisir un but dans la résolvante. Il s'avère que l'algorithme orienté-succès est plus efficace et plus clair [Gur94a, Chapitre 3, section 3.4.4]. Nous l'avons donc utilisé pour nos méta-interprètes.

Dans [MD93], l'algorithme orienté-succès a été le choix immédiat pour l'écriture du méta-interprète réflexif. Pour des raisons de simplicité, pour nos méta-interprètes présentés dans ce mémoire, nous avons remplacé toute occurence du prédicat `vide/0` par le prédicat déjà pré-défini `true/0` en 2-Prolog. De plus, nous avons choisi d'incorporer au code de sélection l'interprétation partielle de la structure de contrôle IF-THEN-ELSE. Il est question ici d'une interprétation partielle car seul le test peut être interprété. Le corps (THEN et ELSE) ne sera jamais interprété à cet endroit. Si la sélection du but le plus à gauche est bloqué par la présence d'une structure IF-THEN-ELSE alors le test (premier paramètre) de la structure est immédiatement interprété pour sélectionner le choix "THEN" ou le choix "ELSE". De cette manière, nous sommes assurés du succès de la sélection. Pour bien comprendre de quoi il s'agit, nous avons inclus ici le code de `selectionne_but/4`.

Du point de vue de la réflexion, cela a des conséquences. En effet, dans les langages réflexifs tels que 3-Prolog$_R$ et 3-Prolog*, il est impossible d'appeler un but réflexif au niveau du test d'une structure IF-THEN-ELSE. En effet, dans le cas des appels réflexifs qui accèdent à la résolvante, il serait impossible de rendre celle-ci disponible en sa totalité au bon moment. Nous croyons que cette contrainte n'enlève pas de la puissance d'expression aux programmeurs. Il sera que très rarement impossible d'écrire les programmes réflexifs pour que les appels réflexifs à proprement dit se fassent à l'extérieur des tests des structures IF-THEN-ELSE.

```
% selectionne_but(+RESOLVANTE, -BUT_CHOISI, -RESTE_BUTS, +PROGRAMME).
% Selection d'un but dans la resolvante.
```

```
selectionne_but(Buts, But, Reste, Prog) :-
        functor(Buts, Pred),
        (Pred = ','/2 ->
            Buts = (A, B),
            selectionne_but(A, But, ResteA, Prog),
            nresolvante(ResteA, B, Reste);
            selectionne_but(Pred, Buts, But, Reste, Prog)).

selectionne_but(Pred, Buts, But, Reste, Prog) :-
        (Pred = ifcut/3 ->
            Buts = (C -> D;E),
            (solve_refl(C, Prog) ->                    % Interpretation du test.
                selectionne_but(D, But, Reste, Prog);
                selectionne_but(E, But, Reste, Prog));
                But = Buts,
                Reste = true).

% nresolvante(+CORPS, +RESOLVANTE, -NOUVELLE_RESOLVANTE).
% Construction de la nouvelle resolvante.

nresolvante(Corps, Res, NRes) :-
    functor(Corps, Pred1),
    (Pred1 = true/0 ->
                NRes = Res;
     functor(Res,  Pred2),
                (Pred2 = true/0 ->
                        NRes = Corps;
                        NRes = (Corps, Res))).
```

Par exemple, si la résolvante est la structure `(C->D;E)`, le but `C` sera premièrement interprété (`solve_refl(C, Prog)`) pour qu'ensuite, le choix de sélection soit fait sur `D` ou bien sur `E` dépendant de la valeur de vérité de l'interprétation de `C`.

L'évaluation partielle élimine complètement tout le code de sélection du but quand il est possible de choisir de manière statique (sans interprétation partielle nécessaire) le premier but dans la résolvante. Par exemple, il est possible sans effectuer aucune interprétation de choisir le premier but dans la résolvante suivante `(a(A), b, c(B), d)`. Dans ce cas, le but `a(A)` est le premier but de la résolvante. Pour cet exemple, l'évaluateur partiel Mixtus élimine tout le traitement de la sélection du but. On retrouve ce qu'on appelle le programme-solution qui suit. Un **programme-solution** est un résultat de l'évaluation partielle pour une requête R qui contient, sous forme de faits, l'ensemble fini de toutes les solutions possibles pour la requête R. L'évaluateur partiel Mixtus a la particularité d'ajouter une clause de façon à ce que la requête R (ou une instance de celle-ci) puisse aussi être appelée sur le programme évalué partiellement. C'est la raison de la présence de la première clause du résultat suivant.

```
selectionne_but((a(A),b,c(B),d), C, D, E) :-
        'selectionne_but,a1'(A, B, C, D, E).
'selectionne_but,a1'(A, B, a(A), (b,c(B),d), _).
```

Il se peut que l'évaluation partielle ne puisse pas éliminer complètement le code de la sélection de but si certaines informations dynamiques font obstacle à une sélection statique telle qu'une interprétation d'un test d'une structure IF-THEN-ELSE lequel test dépend d'information dynamique.

2.2. Sélection d'une clause. Généralement en méta-programmation en Prolog, il n'est pas nécessaire de sélectionner une clause par rapport au but courant avant la réduction. Il est préférable de faire la sélection et la réduction du but courant avec la clause *au même moment*. Néanmoins, du point de vue de la méta-programmation réflexive, il est quelques fois utile de faire une pré-sélection de la clause qui participera à la prochaine réduction, de la rendre disponible à l'usager de façon réflexive pour modification et de poursuivre la réduction ensuite. Cela se retrouve dans l'implémentation du langage 3-Prolog*. Ce découpage plus fin du processus de résolution a l'effet d'ajouter une unification supplémentaire (ou une tentative d'unification) du but courant. Une première unification se fait au cours de la sélection de la clause et une deuxième unification se fait au cours de la réduction en tant que telle.

À l'évaluation partielle, toute manipulation d'information dynamique a des conséquences importantes sur le code généré en sortie. Le but courant est une information hautement dynamique. En conséquence, plus il y a de manipulations (tentatives d'unification) du but courant, plus l'évaluateur partiel a "de la difficulté" à éliminer les opérations de résolution. C'est aussi le cas pour cet exemple. Si le processus de sélection des clauses dépend du but courant, l'évaluateur partiel Mixtus a peu de chance d'optimiser efficacement celui-ci. Au mieux, il génère un programme de grande taille.

Pour éviter ce problème, il est possible de faire un compromis au niveau de la puissance d'accès à la méta-information. Au lieu de faire une sélection de *la* clause qui participera à la réduction ultérieure, il est possible de sélectionner un *ensemble* de clauses qui ont de bonne chance de participer à la réduction ultérieure. Cette sélection des clauses pourrait être fonction du *prédicat* du but courant au lieu du but en sa totalité (prédicat avec tous ses paramètres réels). De cette façon, le processus de sélection dépend de peu relativement peu d'informations dynamiques. Voici un exemple.

Soit le but `pere(X, lucie)` et le programme suivant. Nous l'appellerons le programme **père-mère**.

```
pere(daniel,lucie).   % clause no.1
pere(andre, daniel).  % clause no.2

mere(jeanne,denis).   % clause no.3
```

Si une sélection d'une clause par rapport au but courant devait se faire, la clause numéro 1 devrait être choisie. Si une sélection d'un ensemble des clauses par rapport au prédicat du but courant devait se faire, l'ensemble $\{1, 2\}$ serait construit.

Voici le méta-programme qui permet de construire l'ensemble de clauses en fonction du prédicat du but courant.[2]

```
% selection_clauses(+BUT, -ENSEMBLE_CLAUSES, +PROGRAMME).
selection_clauses(But, Ensemble_clauses, Programme) :-
     selection_clauses(But, Ensemble_clauses, [], Programme).

selection_clauses(But, EnsCls, Liste_partielle, Programme) :-
     functor(But, Predicat, Arite),
     ((member(cl(Tete, _, NoCl), Programme),
       functor(Tete, Predicat, Arite),
       not(member(NoCl, Liste_partielle))) ->
         NListe_partielle = [ NoCl | Liste_partielle ],
         EnsCls = [ NoCl | NEnsCls ],
         selection_clauses(But, NEnsCls, NListe_partielle, Programme);
         EnsCls = []).
```

À titre de démonstration de l'évaluation partielle sur ce méta-programme, voici le résultat quand le programme-objet en paramètre est **père-mère** et que la sélection des clauses doit se faire en fonction du but `pere(_, _)`. On obtient le programme-solution suivant.

```
selection_clauses(pere(A,B), C, [cl(pere(daniel,lucie),true,1), ...]) :-
        selection_clausespere1(A, B, C).

selection_clausespere1(_, _, [1,2]).
```

Il faut dire que cette optimisation est bonne seulement dans le cas où on connaît le but en fonction duquel la sélection doit se faire. Elle sera donc bien meilleure au moment d'une spécialisation plutôt qu'une compilation.

Dans [Gur94a], Gurr conjecture qu'il est possible d'écrire un méta-interprète qui simulerait l'indexation du premier argument de la tête des clauses et que celui-ci puisse être évalué partiellement efficacement. À la lumière des résultats d'évaluation partielle du prédicat `selection_clauses/3`, nous croyons que cette conjecture est raisonnable. En effet, le travail du prédicat `selection_clauses/3` peut être considéré comme l'extraction d'un ensemble de clauses en fonction d'un indexe: le prédicat de la tête d'une clause. Il serait raisonnable de chercher à extraire un ensemble de clauses en fonction d'un nouvel indexe composé du prédicat et du premier argument de la tête d'une clause.

---

[2] Il aurait été beaucoup plus simple d'utiliser ici le prédicat méta-logique `findall/3` de Prolog pour générer cet ensemble. Nous ne l'avons pas utilisé parce qu'il est difficile à évaluer partiellement en Prolog. Il était préférable d'écrire un méta-programme plus spécifique au problème. Celui-ci s'est avéré beaucoup plus facile à évaluer partiellement efficacement.

2.3. Exemplarisation d'une clause. Nous passons maintenant à une analyse des résultats de l'évaluation partielle du processus d'exemplarisation d'une clause (non déterministe) dans un programme.

Dans les chapitres précédents, nous avons vu que la représentation du programme est cruciale pour une bonne évaluation partielle. Nous nous sommes aperçu que l'exemplarisation des clauses devait se faire à l'aide de termes représentant des variables (méta-variables locales) et non pas en utilisant le prédicat méta-logique `copy_term/2` pour exemplariser les variables libres. Voici la façon la plus naturelle d'écrire cette opération. L'idée consiste en l'exemplarisation de tout le programme et la sélection sur le programme exemplarisé. Il est clair que cet algorithme est naïf et peu efficace en lui-même car il exemplarise tout le programme à chaque sélection. Il s'avère que l'évaluation partielle optimise mieux un tel algorithme naïf plutôt qu'un algorithme plus performant qui n'exemplariserait que la clause voulue. On en déduit que les algorithmes performants ne sont pas nécessairement les algorithmes les plus faciles à optimiser par l'évaluation partielle.

Rappelons que l'exemplarisation d'une clause doit principalement résulter en l'obtention d'une clause ne contenant que des variables fraîches. Dans ce cas-ci, le programme dans lequel se trouve la clause à exemplariser ne contient pas de variables libres mais plutôt des méta-variables locales (termes de la forme `v(N)` où `N` est un nombre). L'exemplarisation consiste en le remplacement de ces méta-variables par des variables libres.

```
% clause_prog(+TETE, ?CORPS, +PROGRAMME).
% Semantique equivalente au predicat meta-logique Prolog "clause/2".
clause_prog(Tete, Corps, Prog) :-
        trans_prog(Prog, NProg),
        member(cl(Tete, Corps, _), NProg).

% trans_prog(+PROGRAMME, -NOUV_PROGRAMME).
trans_prog([], []).
trans_prog([ ref(No) | Reste ], [ ref(No) | NReste ]) :-
        trans_prog(Reste, NReste).
trans_prog([ cl(T, C, Ref) | Reste ], [ cl(NT, NC, Ref) | NReste ]) :-
    new_vars((T:-C), (NT:-NC)),
    trans_prog(Reste, NReste).

% new_vars(+TERME, -NOUVEAU_TERME).
new_vars(Terme, NTerme) :- [...]                % Definition omise.
% Ce predicat remplace toutes les occurrences du terme v(N)
% (meta-variable locale) ou N est un nombre par une variable libre.
```

Voici ce qu'on obtient par l'évaluation partielle dans le cas du programme `grand_parent/2`.

```
% Evaluation partielle: 42 secondes.

clause_prog1(A, B) :-
        membercl2(A, B).
```

```
membercl2(grand_parent(A,B), grand_mere(A,B)).
membercl2(grand_parent(A,B), grand_pere(A,B)).
membercl2(grand_pere(A,B), (pere(A,C),parent(C,B))).
membercl2(grand_mere(A,B), (mere(A,C),parent(C,B))).
membercl2(parent(A,B), pere(A,B)).
membercl2(parent(A,B), mere(A,B)).
membercl2(mere(france,jane), true).
membercl2(mere(jane,lucie), true).
membercl2(pere(daniel,lucie), true).
membercl2(pere(andre,daniel), true).
membercl2(pere(jean,denis), true).
```

Le calcul d'exemplarisation a été complètement éliminé. C'est le meilleur résultat que nous pouvions obtenir. Le programme résultant est aussi (sinon plus, si l'on ajoute les facteurs d'optimisation de la compilation traditionnelle) efficace que l'utilisation du prédicat méta-logique `clause/2`.

2.4. Réduction, unification et réification. La réduction d'un but réflexif ou non est un procédé important et délicat à évaluer partiellement. Examinons ce qui se produit en représentation non-close. Dans ce cas, nous espérons que l'évaluation partielle spécialisera l'opération de réduction à certains prédicats réflexifs bien définis. Cette spécialisation éliminera les calculs tels que `append/3` qu'on retrouve dans la définition du prédicat `reduit_refl/4`.

```
reduit_refl(But,Corps,Prog, NProg) :-
   functor(But, Pred, Arite),
        (Pred/Arite = call/1 ->
            call(Corps) = But,
            NProg = Prog;
            (reflexif_explicite(Pred/Arite) ->
                functor(But, Pred, Arite),                % Important.
                But =.. ListeBut,
                append(ListeBut,[Prog,NProg],NListeBut),  % Construction du but reflexif defini.
                NBut =.. NListeBut,
                metavar(NBut),
                Corps = true;
                NProg = Prog,
                (system(Pred/Arite) ->
                    call(But),
                    Corps = true;
                    clause_prog(But, Corps, Prog)))).
```

Supposons, par exemple, que l'on veuille la spécification des prédicats réflexifs suivants. Nous omettons les définitions de ces prédicats. Nous espérons alors retrouver explicitement les appels aux définitions réflexives (inconnues à l'évaluation partielle).

```
reflexif_explicite(predicat1/3).
reflexif_explicite(predicat2/2).
reflexif_explicite(predicat3/1).
```

Voici le résultat de la spécialisation du programme `reduit_refl/4` en fonction de la définition `reflexif_explicite/1` précédente.

```
reduit_refl(A, B, C, D) :-
        reduit_refl1(A, B, C, D).
reduit_refl1(A, B, C, D) :-
        functor(A, E, F),
        (E=call, F=1 ->
            A=call(B), C=D;
            (E=predicat1, F=3 ->
                A=predicat1(G,H,I),
                predicat1(G, H, I, C, D),    % Appel def. reflexive.
                B=true;
                (E=predicat2, F=2 ->
                    A=predicat2(J,K),
                    predicat2(J, K, C, D),   % Appel def. reflexive.
                    B=true;
                    (E=predicat3, F=1 ->
                        A=predicat3(L),
                        predicat3(L, C, D),  % Appel def. reflexive.
                        B=true;
                        D=C,
                        (system(E/F) ->
                            call(A),
                            B=true;
                            clause_prog(A, B, C))).
```

Ce n'est pas une optimisation très importante comparée aux résultats d'optimisation du prédicat `clause_prog/3`. Par contre, les appels aux prédicats `append/3` et `=../2` ont été enlevés. Cela a été rendu possible par l'ajout d'un appel du prédicat `functor/3` dans le code original de `reduit_refl/3` (Dans le code, il y a un commentaire "Important" pour indiquer la position de ce prédicat). Sans ce prédicat (inutile du point de vue de l'algorithme), Mixtus n'aurait pas été capable d'exécuter statiquement les appels aux prédicats `append/3` et `=../2`. Nous avons ici un exemple simple montrant bien qu'il y a une forte dépendance entre l'évaluation partielle et l'algorithme du programme. Brièvement, cela est causé par la présence de la structure IF-THEN-ELSE qui pour effet d'empêcher Mixtus à poursuivre la propagation de l'unification.

Il faut dire aussi que cette optimisation qui, à première vue, ne semble pas être significative, peut l'être si l'on pense que les compilateurs traditionnels modernes de Prolog peuvent plus facilement optimiser ce programme transformé qui ne contient plus qu'un seul appel au prédicat `call/1` (au lieu de 3). Notons aussi que s'il avait fallu optimiser manuellement le méta-interprète en fonction des prédicats réflexifs connus, nous aurions eu un résultat similaire à celui de Mixtus.

Passons maintenant à un programme de réduction qui réifie complètement l'algorithme d'unification. Si l'on effectue des tests similaires sur ce prédicat de réduction, nous remarquerons qu'à peu près les mêmes optimisations relatives sont obtenues par évaluation partielle.

L'ajout d'un algorithme d'unification explicite ne permet pas de faire une meilleure optimisation à ce stade des tests. Cela est normal car il n'est pas possible d'optimiser significativement l'algorithme d'unification sans connaître une bonne partie des termes à unifier. L'algorithme d'unification travaille surtout sur de l'information hautement dynamique. Voyons maintenant comment fonctionne cet algorithme d'unification explicite. Nous comprendrons mieux par la suite les résultats de l'évaluation partielle sur celui-ci.

Comme nous l'avions expliqué dans le chapitre 3, la réification des unifications demande d'avoir une représentation des variables sous forme de termes qui permettent une pleine manipulation et dénotation. Pour être conforme à la représentation utilisée dans [Gur92], les variables sont représentées par un terme (qu'on nomme une méta-variable) de la forme `v(Niv, No)` où `Niv` est un type pour permettre de distinguer différents ensembles de variables et `No` est un nombre qui identifie de façon unique chacune des variables.

Le programme logique qui permet de faire l'unification explicite de deux termes s'appelle par une requête de la forme `unify(+TERME1, +TERME2, +NIVEAU, +UNIFICATIONS, -NOUV_UNIFICATIONS)`. Ce programme vérifie si les deux termes peuvent être unifiés en fonction des unifications actuelles (`UNIFICATIONS`) et du type `NIVEAU` des variables et applique les unifications si cela est possible.

L'algorithme d'unification est particulier. Il est tout aussi en mesure d'unifier des termes dans leur représentation close que des termes dans leur représentation non-close. Les termes `TERME1` et `TERME2` peuvent contenir des variables libres sans problème. Pour ce faire, il utilise en priorité l'unificateur de base du langage par l'appel au prédicat `=/2` défini en 2-Prolog. Cela permet d'accélérer significativement la tâche de la réduction d'un terme clos avec un terme non-clos. En effet, la réduction est d'abord une exemplarisation des clauses du programme, c'est-à-dire une transformation des clauses dans lesquelles les termes clos `v(No)` sont remplacés par des variables libres fraîches et locales pour chaque clause. Ensuite, on tente d'unifier le but courant (toujours en représentation close) avec la tête d'une des clauses transformées. Si l'unificateur de base réussit l'unification du but et de la tête de la clause, la tâche d'unification est complète. Il ne reste plus qu'à unifier les variables libres restantes du corps de la règle choisie à des méta-variables fraîches (terme `v(Niv, No)`) pour terminer la réduction.

Cet algorithme d'unification est simple à implémenter et relativement rapide à l'exécution. Par contre, il serait difficile d'incorporer un test d'occurence au cours de l'unification sans devoir réécrire autrement une majeure partie de l'algorithme. Il n'est donc pas le mieux adapté pour nos besoins de réflexion. Il est un première implémentation complète pour les expérimentations.

La spécialisation d'un algorithme d'unification explicite a été étudiées dans [dWG92b] et dans [Kur87]. Le langage Gödel [HL92] a été conçu principalement pour l'implémentation efficace d'un tel algorithme.

Dans ce langage, une implémentation déclarative de l'algorithme d'unification explicite totalement différente a été créée. Nous conjecturons que notre algorithme est plus efficace. Par contre, celui du langage Gödel est mieux adapté pour les applications complexe de réflexion. Éventuellement, nous devrons passer à l'utilisation d'un tel algorithme dans le domaine de la réflexion déclarative *à la 3-Lisp*. Sa conception se base sur la définition de six prédicats nommés **WAM-comparables**. Ces prédicats sont analogues aux instructions WAM [War83] [AK91] suivantes: `GetValue`, `GetConstant`, `GetFunction`, `UnifyValue`, `UnifyVariable` et `UnifyConstant`. Dans Gödel, certains de ces prédicats sont implémentés efficacement dans un langage de bas niveau.[3] Aidé de l'évaluation partielle, les concepteurs de Gödel ont montré qu'il est possible d'exécuter un méta-interprète sur un programme-objet *aussi efficacement* qu'interpréter directement les programmes-objets par l'interprète de base [Gur94a].

Ce procédé d'implémentation *à la WAM* de l'algorithme d'unification fait partie d'une *méthodologie* plus générale pour le développement d'un compilateur Prolog par spécialisation de méta-programmes. Les développements et résultats récents concernant cette méthodologie laissent penser que, dans un avenir rapproché, la résolution en méta-programmation sera implémentée aussi efficacement en représentation close qu'en représentation non-close.

2.5. Création de points de choix. L'implémentation de la création de points de choix n'a été pensée que dans l'optique de la conception du langage 3-Prolog*. À l'origine, l'implémentation du langage 3-Prolog* avait pour but principal d'expérimenter une méthodologie de réflexion *à la 3-Lisp*. Au cours du développement, il s'est avéré clair que l'évaluation partielle aurait peu de chances de pouvoir servir de façon sérieuse à la compilation de ce langage sous sa forme actuelle. L'implémentation est totalement désavantagée par le fait qu'elle utilise la représentation non-close. L'intention d'utiliser l'évaluation partielle pour compiler les programmes 3-Prolog* a donc été temporairement suspendue jusqu'au moment où une implémentation efficace de celui-ci en représentation close soit réalisée. Les recherches futures pourraient être constituées de la réalisation d'une telle implémentation.

D'une part, nous conjecturons qu'à la lumière des travaux de Consel et Khoo sur la génération d'un compilateur Prolog vers Scheme [CK91], une partie du processus de création de points de choix pourrait être spécialisée de façon significative en connaissance du programme-objet seulement (compilation). En effet, Consel et Khoo montrent que l'évaluation partielle

[3] Malheureusement, l'implémentation de bas niveau de Gödel n'est encore tout à fait terminée.

peut éliminer une partie du code relatif au retour-arrière. Sans vouloir entrer dans les détails, mentionnons seulement que la compilation de l'interprète Prolog écrit en Scheme élimine complètement ce que les auteurs appellent la continuation avec échec ("the failure continuation") qui constitue une part importante de la procédure du retour-arrière.

D'autre part, la combinaison des travaux de Gurr sur l'évaluation partielle méta-circulaire en Gödel et les travaux de Nilsson [Nil93] sur la simulation complète du langage WAM en programmation logique laissent entrevoir la possibilité de concevoir un méta-interprète déterministe (qui simulerait explicitement la création de points de choix) évaluable partiellement. Les travaux de Gurr démontrent tout d'abord que l'utilisation en méta-programmation de la représentation close est raisonnable et suffisamment efficace et ceux de Nilsson démontrent qu'il est possible de simuler les instructions WAM (de manipulation de points de choix) `Try-Me-Else`, `Retry-Me-Else` et `Trust-Me` convenablement en programmation logique. Il est même prévu qu'une implémentation déterministe de l'algorithme de résolution puisse être plus efficace que son homologue non-déterministe (actuellement implémenté en Gödel).

3. Compilation de langages réflexifs

Dans cette section, nous allons compiler les langages réflexifs sur différents programmes réflexifs et non-réflexifs. Rappelons qu'on entend par compilation l'évaluation partielle dont seul le programme-objet est complètement spécifié (la requête à interpréter n'est pas du tout connue).

Bien sûr, étant donné que l'on ne spécifie pas la requête, nous ne pouvons pas nous attendre à éliminer complètement la méta-interprétation car, à partir du programme, l'évaluation partielle ne peut en aucune façon inférer l'ensemble des requêtes possibles pour ce méta-interprète. En d'autres mots, l'ensemble des requêtes possibles pour un certain langage et pour un certain programme de ce langage (réflexif ou non) n'est pas fini. Par contre, nous savons qu'une première optimisation peut être obtenue en connaissance du programme seulement. C'est cette optimisation que nous allons chercher à obtenir.

Dans la première section, nous nous attarderons au langage réflexif utilisant un algorithme d'unification implicite. Nous présenterons alors quelques exemples de compilation des langages 3-Prolog$_P$ et 3-Prolog$_R$.

Dans la deuxième section, il sera question de la compilation de langages réflexifs ayant adoptés plutôt un algorithme d'unification explicite: les langages 3-Prolog$_U$ et 3-Prolog*.

3.1. Compilation et exemplarisation. Comme nous l'avons vu, il y a deux parties à la tâche de la méta-interprétation: l'analyse et l'exécution. Par la compilation, nous cherchons à éliminer la plus grande partie de l'analyse. L'exemplarisation des clauses fait partie de l'analyse. Nous avons vu que l'évaluation partielle améliore remarquablement l'efficacité de l'exemplarisation des clauses du programme. Voyons maintenant si l'évaluation partielle est encore capable d'éliminer le calcul de l'exemplarisation d'un point de vue plus global au processus de résolution, c'est-à-dire du point de vue d'un méta-interprète.

Premièrement, nous avons appliqué l'évaluation partielle sur le méta-interprète 3-Prolog$_P$ avec pour programme-objet non-réflexif `transpose/2`. Nous nous attendions évidemment, comme pour les tests précédents, que tout le calcul d'exemplarisation soit éliminé par l'évaluation partielle.

Malheureusement, l'évaluateur partiel Mixtus n'a pas réussi à éliminer l'exemplarisation tel qu'on s'y attendait. Après analyse du résultat, nous avons compris qu'il est difficile pour Mixtus d'atteindre ce but même s'il est clair qu'aucune modification (par des prédicats réflexifs) ne se produit sur le programme en cours d'exécution (le programme `transpose/2` n'est pas réflexif). On explique ce phénomène par la présence du mécanisme de réflexion pour la représentation du programme-objet qui crée un lien de dépendance entre la représentation du programme à un moment donné et l'état d'exécution. Ce lien de dépendance, pour les méta-interprètes non-réflexifs, n'existe pas. Il ne se rend pas compte du cas exceptionnel d'indépendance entre le programme et la requête.

D'un point de vue plus technique, cela s'explique par l'impossibilité de Mixtus d'unifier "statiquement" (à la compilation) deux variables `Prog` et `NProg`, paramètres du méta-interprète `solve_refl(Buts, Prog, NProg)` réflexif, dont la première représente le programme-objet et la deuxième représente le même programme mais après une réduction.

Le seul gain d'efficacité de cette compilation est l'élimination du code relatif aux appels réflexifs. Dans la cas du programme `transpose/2`, il n'y a aucun prédicat réflexif. Les tests relatifs aux appels réflexifs ont donc été complètement enlevés.

Bien que nous ayons obtenu un succès d'optimisation du méta-programme d'exemplarisation *pris seul*, la *combinaison* de ce même méta-programme à l'intérieur d'un méta-programme plus complexe (méta-interprète), ne nous assure pas nécessairement du même succès d'optimisation. La somme des gains d'optimisation de méta-programmes pris individuellement n'est pas nécessairement égale au gain d'optimisation de la combinaison des méta-programmes.

Dans le cas de la compilation d'un programme réflexif, nous obtenons un résultat similaire au cas précédent. Le calcul de l'exemplarisation n'a pas été éliminé. Nous omettons la présentation d'un exemple.

Nous présentons maintenant la compilation du même programme `transpose/2` dans le langage 3-Prolog$_R$ sans aucun prédicat réflexif. Cette fois-ci, nous obtenons un bien meilleur résultat car étant donné que le programme est une entité qui ne peut être modifiée en cours de l'interprétation, l'évaluation partielle peut effacer tout le code relatif à l'exemplarisation.

```
sr1(A) :-
        solve_refl1(A).
solve_refl1(A) :-
        functor(A, B),
        (   B=true/0 ->
            true
        ;   selectionne_but1(A, C, D),
            functor(C, E, F),
            (   E=call,
                F=1 ->
                C=call(G),
                H=D
            ;   H=D,
                (   system(E/F) ->
                    G=true, call(C)
                ;   C=transpose(I,[]), G=nullrows(I)
                ;   C=transpose(J,[K|L]), G=(colMat(K,M,J),transpose(M,L))
                ;   C=colMat([],[],[]), G=true
                ;   C=colMat([N|O],[P|Q],[[N|P]|R]), G=colMat(O,Q,R)
                ;   C=nullrows([]), G=true
                ;   C=nullrows([[]|S]), G=nullrows(S)
                )
            ),
            functor(G, T),
            (   T=true/0 ->
                U=H
            ;   functor(H, V),
                (   V=true/0 -> U=G
                                ;   U=(G,H))),
            solve_refl1(U)).

% Le predicat "selection_but1/3" est similaire au predicat
% "selectionne_but/3" du programme original.
```

Remarquons que le paramètre du programme-objet a été enlevé, l'exemplarisation a été statiquement exécutée et les différentes clauses du programme ont été incorporées au niveau de la boucle principale du méta-interprète (`solve_refl1/1`).[4]

Voici maintenant un exemple de programme réflexif pour le langage 3-Prolog$_R$.

```
a(A, B, C) :- a(A), b(B), c(C).
```

[4]Cette compilation permet d'obtenir un méta-interprète *plus* efficace qu'un interprète traditionnel utilisant le prédicat méta-logique `clause/2` car les clauses interprétées ne sont plus stockées dynamiquement dans la base de règles. Le compilateur connaît cette fois-ci très précisément les clauses interprétées. De cette façon, il peut améliorer significativement l'efficacité. Cette affirmation a été vérifiée expérimentalement.

```
a(1) :- cc_gf.                  b(3).                   c(5).
a(2).                           b(4).                   c(6).

reflexif_implicite_R(_) :- fail.
reflexif_explicite_R(cc_gf/0).

cc_gf(Res, NRes) :-
  NRes = (call(Res) -> true; fail).
```

Voici le résultat de sa compilation. Le programme et le code du prédicat réflexif ont été intégrés à la boucle principale. C'est le résultat que nous nous attendions.

```
% Evaluation partielle: 19 secondes.
sr1(A) :-
        solve_refl1(A).
solve_refl1(A) :-
        functor(A, B),
        (   B=true/0 ->
            true
        ;   selectionne_but1(A, C, D),
            functor(C, E, F),
            (   E=call, F=1 ->
                C=call(G), H=D
            ;   E=cc_gf, F=0 ->
                C=cc_gf, G=true,
                H=(call(D)->true;fail)              % Code reflexif.
            ;   H=D,
                (   system(E/F) ->
                    G=true, call(C)
                ;   C=a(I,J,K), G=(a(I),b(J),c(K))
                ;   C=a(1), G=cc_gf
                ;   C=a(2), G=true
                ;   C=b(3), G=true
                ;   C=b(4), G=true
                ;   C=c(5), G=true
                ;   C=c(6), G=true)),
            functor(G, L),
            (   L=true/0 ->
                M=H
            ;   functor(H, N),
                (   N=true/0 -> M=G; M=(G,H))),
            solve_refl1(M)).

% Le predicat "selection_but1/3" est similaire au predicat
% "selectionne_but/3" du programme original.
```

3.2. Compilation et unification. Dans cette section, nous nous intéresserons à la compilation de méta-interprètes en représentation close. Cette fois-ci au lieu de présenter le code du résultat pour montrer que l'évaluation partielle apporte une amélioration d'efficacité, nous allons plutôt calculer, pour certaines requêtes, le nombre de réductions avant et après évaluation partielle. Le nombre de réductions du parcours dans l'arbre de recherche est une mesure couramment utilisée dans le domaine.

Par exemple, le nombre de réductions avant évaluation partielle de l'interprétation du programme `transpose/2` pour des matrices 5 par 5 et 10 par 10 sont de 53252 et de 224142.

Sans interprétation, nous obtenons respectivement 47, 142 et 1312 réductions. On déduit que l'interprétation avec unification explicite demande respectivement environ 1000 et 1500 fois plus de réductions que l'exécution sans méta-interprétation.

Après 10 minutes de compilation, nous obtenons respectivement les nombres de réductions de 18663 (1.8 fois moins de réductions), 118203 (0.9 fois moins de réductions). L'analyse du programme compilé révèle que le gain d'optimisation se retrouve encore une fois au niveau de l'exemplarisation du programme et de l'appel de prédicat réflexif.

Il y a peu d'intérêt à étudier la compilation de programmes réflexifs. Il suffit de mentionner que nous avons fait quelques tests et nous avons obtenu des gains d'efficacité comparables au cas du programme non-réflexif précédent.

4. Spécialisation de programmes réflexifs

Rappelons que ce qu'on entend par spécialisation est l'évaluation partielle d'un méta-interprète dont on connaît le programme-objet et le "squelette" de la ou les requêtes sur lesquelles on aimerait améliorer l'efficacité.

4.1. 3-Prolog$_P$. Voici la spécialisation du langage 3-Prolog$_P$ à la requête de la forme `transpose(_,_)`. Le niveau d'interprétation a été complètement éliminé. C'est ce que nous espérions obtenir.

Notons qu'ici, il n'y a eu aucun problème relatif à l'élimination du calcul de l'exemplarisation du programme similaire à celui que nous avions eu à la compilation. Étant donné que, cette fois-ci, la requête était connu à l'évaluation partielle, celle-ci a pu déterminer précisément à quel moment dans l'interprétation, les manipulations du programme devaient se dérouler. Il a donc pu les exécuter de façon statique.

```
% Evaluation partielle: 15 secondes.

srptranspose1(A, B) :-
        solve_refltranspose1(A, B).
solve_refltranspose1(A, []) :-
        solve_reflnullrows1(A).
solve_refltranspose1(A, [B|C]) :-
        'solve_refl,colMat1'(B, D, A, D, C).
solve_reflnullrows1([]).
solve_reflnullrows1([[]|A]) :-
        solve_reflnullrows1(A).
'solve_refl,colMat1'([], [], [], A, B) :-
        solve_refltranspose1(A, B).
'solve_refl,colMat1'([C|D], [E|F], [[C|E]|G], A, B) :-
        'solve_refl,colMat1'(D, F, G, A, B).
```

Voici maintenant un programme réflexif pour le langage 3-Prolog$_P$. Nous allons tenter de le spécialiser à la requête `a(_,_,_)`. Le prédicat réflexif `reverse_prog/0` a pour fonction d'inverser l'ordre des clauses dans le programme et ce, dynamiquement. Après son appel, la sélection des clauses dans le programme se fait dans l'ordre inverse. L'appel de la requête `a(N,M,O)` retourne, dans l'ordre, les réponses suivantes $(1,2,1), (1,2,2), (1,1,1), \ldots$ S'il n'y avait pas eu présence des appels réflexifs, l'ordre aurait été $(1,1,1), (1,1,2), (1,2,1), \ldots$ Rappelons que la réification du programme offre principalement ce pouvoir de modification indirect du mécanisme de sélection des clauses.

```
reflexif_implicite_P(_) :- fail.
reflexif_explicite_P(reverse_prog/0).

reverse_prog(Prog, NProg) :-
    reverse(Prog, NProg).

reverse(List, NList) :-
    reverse(List, NList, []).

reverse([], L, L).
reverse([E | RL1], L2, L3) :-
    reverse(RL1, L2, [E | L3]).

%%%%%%%%%%

a(N, M, P) :- a(N), reverse_prog, a(M), reverse_prog, a(P).

a(1).
a(2).
```

Voici le programme identité retourné par l'évaluation partielle pour la requête `a(_,_,_)` dans le langage 3-Prolog$_P$.

```
% Evaluation partielle: 23 secondes.

srp(a(A,B,C), [ref(53),cl(a(v(2),v(1),v(0)), ...]) :-
        srpa1(A, B, C).
srpa1(1, 2, 1).
srpa1(1, 2, 2).
srpa1(1, 1, 1).
srpa1(1, 1, 2).
srpa1(2, 2, 1).
srpa1(2, 2, 2).
srpa1(2, 1, 1).
srpa1(2, 1, 2).
```

Ensuite, nous avons tenté de spécialiser le programme `solve/1` (défini après ce paragraphe) à la requête `solve(A)` où `A` est libre. L'évaluateur partiel Mixtus ne termine pas son exécution. L'algorithme de terminaison ne réussit pas à détecter certaines boucles infinies au cours du dépliage. En fait, ce problème n'est pas causé par une faiblesse de l'algorithme de terminaison

de Mixtus. Au contraire, cette non-terminaison est tout à fait normale. Cela s'explique plutôt comme suit. À un moment donné au cours du dépliage, Mixtus instancie `A` à un terme de la forme `solve(B)`. En effet, rien n'empêche Mixtus de traiter la requête de la forme `solve(solve(B))`. L'algorithme de terminaison ne signale pas l'existence d'un début de boucle sans fin; le dépliage se poursuit donc. Par la suite, la variable `B` s'instancie à un terme `solve(C)`. On se retrouve avec la requête `solve(solve(solve(C)))`. Les instanciations se poursuivent sans arrêt, sans plainte de la part de l'algorithme de terminaison.

```
solve(true).
solve((A,B)) :-
        solve(A),
        solve(B).
solve(A) :-
        functor(A, Pred),
        not(Pred = ','/2),
        clause(A, B),
        solve(B).

% PLUS Programme "transpose/2".

%%%%%%%%%%%%

reflexif_explicite_P(clause/2).

clause(Tete, Corps, Prog, Prog) :-
        clause_prog(Tete, Corps, Prog).
```

De ce petit exemple, on déduit que certains types de réflexion peuvent causer la non-terminaison de l'évaluation partielle. Dans ce cas-ci, c'est une forme d'auto-référence (par le prédicat réflexif `clause/2`) à la base de règles du programme-objet qui produit la non-terminaison.

Pour s'assurer que notre explication est raisonnable, examinons le cas de la spécialisation du même programme à la requête `solve(transpose(_, _))`. Notons que cette fois-ci, il ne peut pas se produire d'appel auto-référenciel comme pour le cas précédent. En effet, Mixtus retourne le résultat attendu suivant.

```
% Evaluation partielle: 8 minutes.

srp(solve(transpose(A,B)), [ref(60),cl(solve(true),true,50),...]) :-
        srpsolvetranspose1(A, B).
srpsolvetranspose1(A, B) :-
        solve_reflsolvetranspose1(A, B).
solve_reflsolvetranspose1(A, []) :-
        solve_reflsolvenullrows1(A).
solve_reflsolvetranspose1(A, [B|C]) :-
        'solve_refl,solvecolMat1'(B, D, A, D, C).
solve_reflsolvenullrows1([]).
solve_reflsolvenullrows1([[]|A]) :-
        solve_reflsolvenullrows1(A).
'solve_refl,solvecolMat1'([], [], [], A, B) :-
        solve_reflsolvetranspose1(A, B).
```

```
'solve_refl,solvecolMat1'([C|D], [E|F], [[C|E]|G], A, B) :-
        'solve_refl,solvecolMat1'(D, F, G, A, B).
```

4.2. 3-Prolog$_R$. Dans le langage 3-Prolog$_R$, nous avons spécialisé le programme réflexif présenté en section 3.1 à la requête-objet `a(_, _, _)`. Nous nous attendions à obtenir le programme identité en sortie. Malheureusement, Mixtus n'a pas réussi à générer ce programme identité. Voici ce qui nous été retourné.

```
% Evaluation partielle: 40 secondes.

sr(a(A,B,C), [ref(57),cl(a(v(2),v(1),v(0)), ...)]) :-
        sra1(A, B, C).
sra1(1, A, B) :-
        (   A=3, B=5 -> C=true, D=true
        ;   A=3, B=6 -> C=true, D=true
        ;   A=4, B=5 -> C=true, D=true
        ;   A=4, B=6 -> C=true, D=true
        ;   C=fail, D=true),
        functor(C, E, F),
        (   E=call, F=1 -> C=call(G), H=D
        ;   E=cc_gf, F=0 -> C=cc_gf, G=true,
            H=(call(D)->true;fail)
        ;   H=D,
            (   system(E/F) ->
                G=true, call(C)
            ;   C=a(I,J,K), G=(a(I),b(J),c(K))
            ;   C=a(1), G=cc_gf
            ;   C=a(2), G=true
            ;   C=b(3), G=true
            ;   C=b(4), G=true
            ;   C=c(5), G=true
            ;   C=c(6), G=true)),
        functor(G, L),
        (   L=true/0 ->
            M=H
        ;   functor(H, N),
            (   N=true/0 ->
                M=G
            ;   M=(G,H))),
        solve_refl2(M).

sra1(2, 3, 5).
sra1(2, 3, 6).
sra1(2, 4, 5).
sra1(2, 4, 6).

% Le predicat "solve_refl2/1" est le resultat de la compilation
% du programme-objet au langage 3-Prolog_R.
```

Après réflexion sur le code généré, nous avons compris la raison de cet échec. Il s'agit d'un problème de représentation des variables présentes dans la requête spécifiée à Mixtus. Nous avions fourni une requête telle que la suivante: `a(A, B, C)` où `A`, `B` et `C` sont des variables libres. Mixtus, ayant pris connaissance de la requête, considère les variables libres comme étant de

l'information dynamique. Ces informations dynamiques sont propagées. Au cours de l'évaluation partielle, il se produit un blocage de la propagation au moment de l'exécution du code réflexif. En effet, l'appel réflexif a pour effet de transformer la résolvante R en une résolvante de la forme (`R -> true; fail`). Par la suite, l'évaluateur partiel tente de simplifier l'interprétation de cette nouvelle résolvante. Il ne peut y arriver si R contient des variables dynamiques car R se trouve au niveau du test de la structure de contrôle IF-THEN-ELSE et Mixtus ne peut déterminer quelle est la valeur de vérité de façon statique. C'est pour cette raison qu'il est incapable de produire le programme identité attendu.

Pour remédier à ce problème, nous avons modifié quelque peu le code du méta-interprète de façon à pouvoir informer Mixtus que les variables `A`, `B` et `C` sont *statiques* et non dynamiques. les variables sont remplacées par les termes (méta-variables locales) `v(0)`, `v(1)` et `v(2)`. Le changement de statut de ces variables a des conséquences importantes. Cela signifie que la requête autorisée sur le programme en sortie de Mixtus sera toujours la suivante: `a(v(0), v(1), v(2))`. Par exemple, les requêtes de la forme `a(1, 3, C)` ne seront plus autorisées.

Voici la modification fait au méta-interprète. Le nouveau prédicat du méta-interprète devient `solve_resolvante_clos/3`.

```
% solve_resolvante_clos(+REQUETE_CLOSE, -REPONSE, +PROGRAMME)
solve_resolvante_clos(Buts, Rep, Prog) :-
        new_vars(Buts, NButs),
        sr(NButs, Prog),      % Appel de l'interprete 3-Prolog_R.
        Rep = NButs.

% Le predicat "new_vars/2" transforme les meta-variables locales "v/1"
% en variables libres.
```

En utilisant ce nouveau méta-interprète pour la requête `a(v(1), v(2), v(3))`, cette fois-ci, nous obtenons le programme identité.

```
% Evaluation partielle: 29 secondes.

srg(a(v(1),v(2),v(3)), A, [ref(57),cl(a(v(2),v(1),v(0)), ...]) :-
        srgav11(A).
srgav11(a(1,3,5)).
srgav11(a(2,3,5)).
srgav11(a(2,3,6)).
srgav11(a(2,4,5)).
srgav11(a(2,4,6)).
```

On en déduit que la réflexion de la résolvante n'est pas la cause de ce problème de spécialisation. Il est plutôt question ici d'une faiblesse de l'évaluateur partiel Mixtus. En effet, Mixtus utilise une représentation non-close de la requête. En passant à une représentation close de la

requête, le problème disparaît. Il y a donc ici l'identification d'un besoin d'utilisation de la représentation close en évaluation partielle.

4.3. 3-Prolog$_U$. Nous passons maintenant à notre dernière série de tests. Nous spécialiserons quelques programmes réflexifs et non-réflexifs sur le langage 3-Prolog$_U$.

La spécialisation du programme `transpose/2` a été d'une durée de 11 minutes. Pour des matrices 5 par 5 et 10 par 10, nous obtenons les nombres de réductions suivants respectivement: 16534 (0.12 fois moins de réductions que le résultat de la compilation) et 111629 (0.05 fois moins de réductions). Le gain d'efficacité est mystérieusement négligeable! Notre algorithme d'unification explicite n'est pas adapté pour l'évaluation partielle. Il est probable que nous ayons utilisé beaucoup trop la structure IF-THEN-ELSE. Une trop grande utilisation de cette structure peut empêcher l'évaluation partielle de propager suffisamment certaines informations statiques. Il est préférable d'utiliser l'indexation plutôt que la structure IF-THEN-ELSE pour réduire la création de points de choix.

Considérons le programme réflexif suivant.

```
grand_pere(X,Y) :-
        var(X),
        pere(Z, Y),
        pere(X, Z).
grand_pere(X,Y) :-
        nonvar(X),
        pere(X, Z),
        pere(Z, Y).

% le predicat "pere/2" n'est pas defini.

reflexif_explicite_U(var/1).
reflexif_implicite_U(_) :- fail.

var(v(Niv,No), Niv, u(N, Unif), u(N, Unif)) :-
        (member((v(Niv, No)=Term), Unif) ->
            functor(Term, Pred),
            not(Pred = mv/1);
            No =< N).
```

Nous allons spécialiser ce programme à la requête `grand_pere(_, _)`. Nous comparerons les résultats à ceux de la méta-interprétation.

Avant évaluation partielle, pour une définition courte du prédicat `pere/2`, nous obtenons le nombre de réductions de 1038 pour la première réponse. Après évaluation partielle, pour la même définition, nous obtenons 220. Nous avons donc obtenu un programme résultant qui prend 3.7 fois moins de réductions pour la première réponse.

Examinons le cas de l'évaluation partielle d'une requête très simple: `var(A)`, l'appel du prédicat réflexif qui vérifie si la variable `A` est libre ou non. Nous nous attendons évidemment

que le programme résultant soit de très petite taille. En effet, voici le résultat retourné par Mixtus.

```
% Evaluation partielle: 2 secondes.

sgr(var(A), var(B), C, [ref(0)]) :-
        sgrvar1(A, B, C).
sgrvar1(v(1,B), v(1,B), A) :-
        B=<A.
```

L'évaluation partielle a éliminé toute la méta-interprétation. Le résultat est bien une spécialisation de la définition réflexive de `var/1` présenté plus haut.

4.4. 3-Prolog*. Comme nous l'avons dit antérieurement, nous n'avons pas réussi à compiler des exemples à l'aide de Mixtus dans un temps et un espace raisonnables. Les quelques tentatives de compilation de 3-Prolog* ont trop souvent échoué pour raison d'un manque d'espace mémoire! Le peu d'exemples qui a abouti à un succès avait consommé un temps (plusieurs jours) et un espace (plusieurs centaines de méga-octets).

Deux raisons peuvent expliquer cela: Mixtus n'est pas suffisamment efficace et 3-Prolog* est incorrectement implémenté. Mixtus fonctionne dans un temps pseudo-quadratique [Pre93]. 3-Prolog* a été implémenté en 3-Prolog$_U$ plutôt que directement en 2-Prolog. Cela le rend impossible à évaluer partiellement (et même à exécuter). Les quelques essais d'évaluation partielle ont été tentés en considérant 3-Prolog* implémenté *en Prolog* plutôt qu'en 3-Prolog$_U$. Mais même en enlevant la couche 3-Prolog$_U$, il était pratiquement impossible de penser obtenir des résultats satisfaisants à cause de la présence des prédicats méta-logiques tels que `copy_term/1` difficile à évaluer partiellement.

Nous croyons, par contre, qu'une implémentation en représentation close efficace et évaluable partiellement dans un temps acceptable puisse être réalisée. Les recherches sur la représentation close par l'entremise de la conception du langage Gödel ont montré qu'une implémentation bien pensée d'un méta-interprète utilisant la représentation close et aidé de l'évaluation partielle puisse donner une efficacité comparable à une exécution d'une implémentation de bas niveau de l'algorithme du même méta-interprète.

5. Conclusion

Étant donné le nombre de tests présentés dans ce chapitre, il est nécessaire de revoir brièvement les résultats obtenus. D'abord, mentionnons quelques faits importants. Tous les tests se sont faits à l'aide de l'évaluateur partiel Mixtus. Nous nous sommes intéressé surtout à la qualité du résultat en sortie en mesurant qualitativement les gains d'optimisation. Sachant

que Mixtus, bien implémenté, ne génère que des résultats corrects et bien calculés, nous n'avons pas vérifié si les programmes sont sémantiquement équivalents aux programmes de départ. Nous n'avons décelé donc aucune erreur d'équivalence sémantique.

Nous nous sommes intéressé à l'évaluation partielle de tours réflexives d'un seul niveau d'interprétation. Nous avons d'abord spécialisé chaque opération de base du méta-interprète réflexif: sélection du but, sélection d'une clause, exemplarisation d'une clause, réduction, unification et réification. Nous cherchions à déceler les opérations qui pouvaient être effectivement optimisées. L'exemplarisation d'une clause et la réification sont les deux opérations qui obtiennent la meilleure optimisation.

D'un point vue plus global, nous nous sommes intéressé à deux types d'optimisation de la méta-interprétation: la compilation et la spécialisation. Bien entendu, la compilation ne nous fournit pas le degré d'optimisation que la spécialisation peut nous fournir. Néanmoins, comme nous l'avons vu, la compilation n'est pas une optimisation négligeable. De son côté, la spécialisation a la particularité d'éliminer, dans bien des cas, toute la couche d'interprétation.

La compilation des langages réflexifs s'est passée sans embûche majeure sauf pour le langage 3-Prolog$_P$ dont l'évaluation partielle n'a pas éliminé le calcul de l'exemplarisation comme pour la compilation des autres langages réflexifs.

La spécialisation élimine toute opération de méta-interprétation dans le cas des langages 3-Prolog$_P$ et 3-Prolog$_R$. Par contre, pour le langage 3-Prolog$_U$, la spécialisation n'améliore que relativement peu l'exécution. Plusieurs opérations d'unification explicite demeurent dans le programme résultant. Ce n'est que sur des requêtes relativement simples telles que `var/1` qu'elle enlève entièrement les opérations d'unification explicite. Les résultats d'optimisation de la spécialisation avec unification explicite sont faibles comparés aux résultats obtenus dans [Gur94b, Gur94a]. Dans notre cas, nous obtenons une optimisation d'un facteur 4. Cela s'explique par le fait que les travaux de Gurr publiés dans les articles précédents et aussi dans [Gur92] ont permis de développer une méthodologie de méta-programmation fortement déclarative en représentation de Gödel de façon à tirer profit de plusieurs techniques traditionnelles d'optimisation de l'exécution telles que l'indexation, la compilation traditionnelle, et l'implémentation de bas niveau. Par exemple, il a utilisé, pour l'écriture de ses méta-programmes, les prédicats WAM-comparables (éventuellement implémentés très efficacement en Gödel). De plus, il a évité le plus possible d'utiliser la structure IF-THEN-ELSE en s'appuyant fortement sur l'indexation des clauses. Dans notre cas, nous n'avons pas fait attention à cela.

Enfin, très peu d'expérimentations ont abouti à un succès dans le cas de la compilation et la spécialisation du langage 3-Prolog*. La lenteur du système Mixtus, la complexité et l'inefficacité de notre implémentation du langage 3-Prolog* ont fortement suggéré l'abandon de tentatives d'optimisation par l'évaluation partielle. Il faudra penser implémenter 3-Prolog* en représentation close et reprendre les tests d'évaluation partielle. Nous conjecturons qu'une implémentation bien pensée de ce langage rende son utilisation sérieuse possible dans un proche avenir.

CHAPITRE 6

Perspectives et conclusion

Dans ce mémoire, nous avons étudié deux domaines de recherches: la réflexion *à la 3-Lisp* et l'optimisation par l'évaluation partielle en programmation logique.

Nous nous sommes attardé d'abord à la rationalisation du langage Prolog de façon à réduire la complexité de la tâche d'intégration des notions de réflexion. Nous avons abouti à la spécification du langage appelé 2-Prolog. Dans celui-ci, nous avons écrit quatres langages réflexifs: 3-Prolog$_P$ qui réifie le programme, 3-Prolog$_U$ qui réifie la liste des unifications, 3-Prolog$_R$ qui réifie la résolvante et 3-Prolog* qui réifie principalement la pile de points de choix. À l'aide de ces langages expérimentaux, nous avons donné quelques démonstrations de la puissance d'expression de la réflexion en programmation logique en redéfinissant simplement les prédicats méta-logiques qui avait déjà été exclus par rationalisation.

Ensuite, nous avons appliqué l'évaluation partielle sur nos quatres langages réflexifs. Nous nous sommes intéressé plus particulièrement à deux types d'application de l'évaluation partielle en méta-programmation: la compilation et la spécialisation. Il a aussi été question de représentation close et de représentation non-close, les deux grandes philosophies de méta-programmation logique.

En représentation non-close, nous avons obtenu de bons résultats par la compilation. Dans ce cas, l'optimisation s'est faite surtout au niveau de l'exemplarisation des clauses. De bien meilleurs résultats ont été rendus par la spécialisation. En effet, dans bien des cas, elle a permis l'élimination complète de la couche d'interprétation.

Beaucoup moins d'études ont été faites en représentation close. La complexité de l'algorithme de résolution a empêché la compréhension profonde des problèmes. Les quelques essais sur 3-Prolog$_U$ de compilation dans cette représentation ont résulté en une optimisation presqu'exclusive du calcul de l'exemplarisation des clauses. La spécialisation, quand à elle, n'a résulté qu'en une optimisation faible comparée à celle de la compilation. Nous croyons qu'il est possible

d'obtenir mieux en réécrivant plus adéquatement l'algorithme de résolution en représentation close. Aucune évaluation partielle a été tentée sur 3-Prolog*. L'implémentation actuelle n'est pas suffisamment efficace.

Les résultats de nos recherches nous ont permis de tirer quelques conclusions plus générales. À l'aide de celles-ci, nous énumérons quelques perspectives de recherche future. La figure 6.4 illustre les relations de dépendance de plusieurs notions centrales discutées dans ce mémoire. Nous commentons les différents liens de dépendance qui ont été numérotés de 1 à 6 dans la figure.

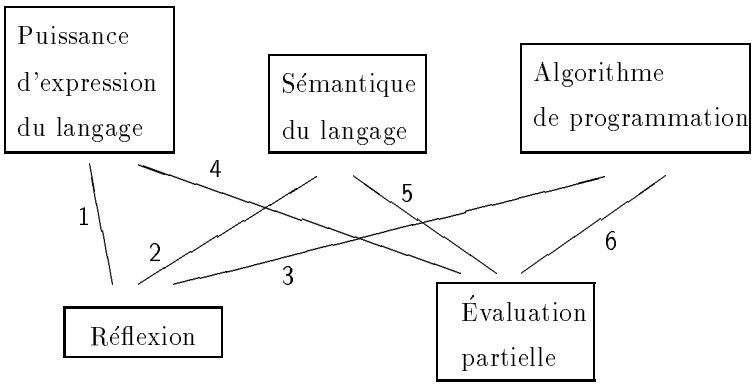

FIGURE 6.4. Schéma de relations de dépendance

1. **Réflexion et puissance d'expression:** L'objectif principal de l'intégration des notions de réflexion dans un langage est l'augmentation naturelle de la puissance d'expression du langage. Pour atteindre cet objectif, il faut d'abord que le langage soit suffisamment puissant avant même l'ajout de la réflexion. Le langage doit être capable d'exprimer et de dénoter facilement toutes les entités relatives à la syntaxe ou à la sémantique du langage.

   Le langage Prolog a déjà une grande puissance d'expression mais elle est encore insuffisante: il est impossible de dénoter de façon unique et naturelle toutes les entités du langage, en particulier, les variables. Pour remédier à ce problème, nous avons dû concevoir une représentation des variables pour les rendre dénotables. Cela a eu l'effet de complexifier considérablement l'implémentation de la réflexion dans le langage. De plus, cette représentation a énormément diminué l'efficacité d'exécution du langage de base. Par contre, cela a permis le développement d'une réflexion très puissante implémentée dans un langage déclaratif ne contenant que très peu d'outils dont leur sémantique est figée dans le langage.

Nos recherches nous ont convaincu que la représentation close est plus puissante et mieux adaptée à l'implémentation d'une réflexion *à la 3-Lisp* de haut niveau. Pour obtenir de bons résultats à l'évaluation partielle, les recherches devront cependant être orientées vers le développement d'outils de compilation efficace de bas niveau des algorithmes de résolution en représentation close de façon à pallier à sa piètre efficacité actuelle.

2. **Réflexion et sémantique:** La réflexion ajoute de la puissance au langage tout en permettant de simplifier et d'uniformiser sa sémantique.
   Nous avons rationalisé le langage Prolog de façon à éliminer une grande part des prédicats méta-logiques n'ayant aucune sémantique déclarative tels que `clause/2`, `var/1`, `==/2`, le coupe-choix et d'autres. Nous avons obtenu un langage presque complètement déclaratif. Nous avons pu par la suite réintroduire les prédicats méta-logiques de façon réflexive seulement. Cela a pour avantage majeur que les définitions de ceux-ci deviennent accessibles pour modification.
   Nos recherches démontrent que la réflexion aide à la formalisation et la simplification de la sémantique fixe (non-modifiable par le programmeur) des langages. Il faudra orienter les recherches futures pour déterminer plus précisément jusqu'où peut aboutir cette entraide entre la sémantique des langages et la réflexion.

3. **Réflexion et algorithme:** Pour développer la réflexion en programmation logique, nous nous sommes grandement inspiré de l'algorithme de contrôle traditionnel. Cet algorithme est composé des structures de requête, résolvante, liste d'unifications et pile de points de choix.
   L'expérience a montré qu'il n'est pas simple d'écrire par la méta-programmation un algorithme de contrôle pour un langage ayant un haut niveau de réflexion. La plupart des algorithmes d'implémentation efficaces ont plutôt été développés en programmation fonctionnelle alors que peu d'algorithmes complexes avaient vu le jour en programmation logique. Nous avons repris l'un d'entre eux (dans 3-Prolog$_P$, 3-Prolog$_R$ et 3-Prolog$_U$) et nous en avons développé un nouveau (dans 3-Prolog*). Néanmoins, ce dernier est désavantagé par le fait qu'il est inefficace et difficile à optimiser par l'évaluation partielle.
   Les recherches futures serviront à simplifier les algorithmes d'implémentation en programmation logique et d'en développer d'autres plus efficaces. De plus, il est probable que d'autres algorithmes déjà développés en programmation fonctionnelle puissent

servir de sources d'inspiration pour le développement de nouvelles implémentations de la réflexion en programmation logique.

4. **Évaluation partielle et puissance d'expression:** L'historique des recherches en évaluation partielle en programmation logique a montré que la puissance d'expression du langage a une grande importance pour l'obtention d'un algorithme d'évaluation partielle simple et efficace. En effet, nous avons vu que certaines recherches tentent de démontrer qu'en programmation logique, la représentation close, est beaucoup mieux adaptée pour l'évaluation partielle de méta-programmes.
   Bien que l'optimisation de méta-interprètes en représentation non-close ait donné des résultats spectaculaires pour des exemples de réflexion simple, les résultats d'optimisation dans le cas de réflexion plus complexe en utilisant les prédicats méta-logiques de Prolog sont désastreux. La représentation close doit donc être envisagée.
   Bien que nos résultats ne puissent pas nous convaincre que l'évaluation partielle optimise aussi bien les méta-programmes en représentation close que ceux en représentation non-close, nous croyons qu'elle est beaucoup mieux adaptée pour l'optimisation de méta-interprètes très complexes tels que 3-Prolog$_U$ et 3-Prolog*. Étant donné la complexité de tels méta-interprètes, il faudra développer des algorithmes d'évaluation partielle simples et très performants de façon à s'assurer de temps d'optimisation raisonnables.

5. **Évaluation partielle et sémantique:** Auparavant, il était convenu qu'un évaluateur partiel devait contenir plusieurs règles de transformation pour pouvoir optimiser efficacement un très large éventail de programmes. Par la suite, les recherches ont montré que plus un évaluateur partiel contenait de règles de transformation, moins il s'avérait efficace. Récemment, les intérêts de recherche se sont dirigés vers la conception d'évaluateurs partiels très performants et ne contenant que peu de règles de transformation.
   Pour pouvoir ne s'en tenir qu'à quelques règles de transformation sans devoir réduire trop les capacités de l'évaluation partielle face à la puissance du langage, il a fallu entreprendre une forme de "nettoyage" de la sémantique du langage Prolog. Par exemple, la notion de déduction partielle est apparue pour faire opposition à l'évaluation partielle. La déduction partielle est l'étude de l'évaluation partielle restreinte exclusivement aux langages complètement déclaratifs. En méta-programmation, la notion de représentation close a servi à éviter l'utilisation des prédicats méta-logiques tels que

`clause/2` et `copy_term/2` très utiles au moment de l'écriture de méta-programmes mais exigeant de nouvelles règles de transformation coûteuses.

Par contre, cette simplification des mécanismes de base a affaibli significativement la puissance d'optimisation de l'évaluation partielle. Les recherches futures devront s'orienter, d'abord, vers le développement d'outils d'analyse de programmes qui serviront (aux programmeurs) à estimer les gains d'optimisation pour en mesurer la pertinence de l'application de certaines transformations de programmes et, ensuite, vers le développement de nouvelles techniques efficaces de transformation de programmes basée sur la sémantique qui, ou bien aideront indirectement à une meilleure évaluation partielle, ou bien serviront à l'obtention d'une meilleure optimisation des programmes.

6. **Évaluation partielle et algorithme:** Il existe une dépendance forte entre les algorithmes et l'évaluation partielle. Il n'est pas simple d'écrire un bon algorithme de façon à ce qu'il soit évaluable partiellement. À titre d'exemple, nous avons développé un langage réflexif très complexe nommé 3-Prolog$_U$. Bien qu'il soit relativement efficace à l'exécution, il est difficile à bien évaluer partiellement. En fait, l'implémentation n'est pas suffisamment déclarative au sens où il y a une trop grande utilisation de la structure IF-THEN-ELSE. Pour comprendre cette raison, il est nécessaire de bien saisir les mécanismes internes des évaluateurs partiels utilisés.

   De plus, les concepteurs du langage Gödel ont conçu une méthodologie d'implémentation par la méta-programmation dans le but d'une meilleure optimisation par évaluation partielle. Pour ce faire, ils se sont basés sur les algorithmes écrits dans le langage WAM [War83]. Néanmoins, il reste encore bien des problèmes à régler relatifs à de meilleures évaluations partielles de méta-programmes complexes.

   Des recherches devront donc se faire pour le création de nouvelles méthodologies de méta-programmation (fortement déclarative). De plus, il faudra développer des outils de vulgarisation des mécanismes internes des évaluateurs partiels pour aider les programmeurs dans leurs choix d'algorithmes pour l'obtention de meilleures évaluations partielles.

Beaucoup de recherches et de développement restent à faire en réflexion en programmation logique et en évaluation partielle. Par contre, notre travail suggère de nouvelles pistes encourageantes vers lesquelles orienter les recherches. Nous croyons que malgré les résultats mitigés

des recherches actuelles, il est possible de penser réaliser dans un avenir rapproché un langage déclaratif hautement réflexif et suffisamment efficace après compilation.

APPENDICE A

Méta-interprètes réflexifs

1. Méta-programmes utiles

```
% clause_prog(+TETE, -CORPS, +PROGRAMME)
clause_prog(Tete, Corps, Prog) :-
    trans_prog(Prog, NProg),
    member(cl(Tete, Corps, _), NProg).

trans_prog([], []).
trans_prog([ ref(No) | Reste ], [ ref(No) | NReste ]) :-
    trans_prog(Reste, NReste).
trans_prog([ cl(T, C, Ref) | Reste ], [ cl(NT, NC, Ref) | NReste ]) :-
    new_vars((T:-C), (NT:-NC)),
    trans_prog(Reste, NReste).

new_vars(Terme, NTerme) :-
    new_vars(Terme, NTerme, [], _).

new_vars(Terme, NTerme, LVars, NLVars) :-
    Terme = v(No) ->
    (member((v(No)=Var), LVars) ->
        NTerme = Var,
        NLVars = LVars;
        NLVars = [(v(No)=X) | LVars],
        NTerme = X);
        ( atomic(Terme) ->
            NTerme = Terme,
            NLVars = LVars;
            functor(Terme, P, A),
            functor(NTerme, P, A),
            new_vars(A, Terme, NTerme, LVars, NLVars)).


new_vars(Arite, T1, T2, LVars, NLVars) :-
    (Arite = 0 ->
        NLVars = LVars;
        arg(Arite, T1, Arg1),
        arg(Arite, T2, Arg2),
        NArite is Arite - 1,
        new_vars(Arg1, Arg2, LVars, LVars_x),
        new_vars(NArite, T1, T2, LVars_x, NLVars)).

metavar(But) :- call(But).
```

```
selectionne_but(Buts, But, Reste, Prog) :-
    functor(Buts, Pred),
    (Pred = ','/2 ->
        Buts = (A, B),
        selectionne_but(A, But, ResteA, Prog),
        nresolvante(ResteA, B, Reste);
        selectionne_but(Pred, Buts, But, Reste, Prog)).

selectionne_but(Pred, Buts, But, Reste, Prog) :-
    (Pred = ifcut/3 ->
        Buts = (C -> D;E),
        (solve_refl(C, Prog) ->
            selectionne_but(D, But, Reste, Prog);
            selectionne_but(E, But, Reste, Prog));
            But = Buts,
            Reste = true).

nresolvante(Corps, Res, NRes) :-
    functor(Corps, Pred1),
    (Pred1 = true/0 ->
        NRes = Res;
        functor(Res,  Pred2),
        (Pred2 = true/0 ->
            NRes = Corps;
            NRes = (Corps, Res))).

not(Term) :- (call(Term) -> fail; true).

notmember(Pred, Liste) :-
    notmember_x(Liste, Pred).

notmember_x([], _).
notmember_x([ Pred | RPreds ], APred) :-
    not(Pred = APred),
    notmember_x(RPreds, APred).

';'(A,_) :- A.
';'(_,B) :- B.
```

2. 3-PROLOG$_P$

```
% srp(+BUT, +PROGRAMME)
srp(But, P) :-
    solve_refl(But, P).

solve_refl(Resolvante, Prog) :-
    solve_refl(Resolvante, Prog, Prog).

solve_refl(Resolvante, Prog, NProg) :-
    functor(Resolvante, Pred),
    (Pred = true/0 ->
        NProg= Prog;
        selectionne_but(Resolvante,But,ResteResolvante, Prog),
        appel_reflexion_implicite(Prog, Prog_x),
        reduit_refl(But, Corps, Prog_x, Prog_xx),
        nresolvante(Corps, ResteResolvante, NResolvante),
        solve_refl(NResolvante, Prog_xx, NProg)).

reduit_refl(But,Corps,Prog, NProg) :-
    functor(But, Pred, Arite),
    (Pred/Arite = call/1 ->
        call(Corps) = But,
        NProg = Prog;
        (reflexif_explicite_P(Pred/Arite) ->
            functor(But, Pred, Arite),
            But =.. ListeBut,
            append(ListeBut,[Prog,NProg],NListeBut),
            NBut =.. NListeBut,
            metavar(NBut),
            Corps = true;
            NProg = Prog,
            (system_prog(Pred/Arite) ->
                call(But),
                Corps = true;
                clause_prog(But, Corps, Prog)))).

system_prog(Pred) :- system(Pred).

metavar(But) :- call(But).

selectionne_but(Buts, But, Reste, Prog) :-
    functor(Buts, Pred),
    (Pred = ','/2 ->
        Buts = (A, B),
        selectionne_but(A, But, ResteA, Prog),
        nresolvante(ResteA, B, Reste);
        selectionne_but(Pred, Buts, But, Reste, Prog)).

selectionne_but(Pred, Buts, But, Reste, Prog) :-
    (Pred = ifcut/3 ->
        Buts = (C -> D;E),
        (solve_refl(C, Prog) ->
            selectionne_but(D, But, Reste, Prog);
            selectionne_but(E, But, Reste, Prog));
            But = Buts,
            Reste = true).

nresolvante(Corps, Res, NRes) :-
    functor(Corps, Pred1),
```

```
    (Pred1 = true/0 ->
        NRes = Res;
        functor(Res,  Pred2),
        (Pred2 = true/0 ->
            NRes = Corps;
            NRes = (Corps, Res))).


appel_reflexion_implicite(Prog, NProg) :-
    appel_refl_imp([], Prog, NProg).

appel_refl_imp(Liste, Prog, NProg) :-
    ((reflexif_implicite_P(P/A),
    notmember(P/A, Liste)) ->
    functor(But, P, A),
    But =.. ListeBut,
    append(ListeBut, [Prog, NProg], NListeBut),
    NBut =.. NListeBut,
    metavar(NBut),
    appel_refl_imp([P/A|Liste], Prog, NProg);
    NProg = Prog).

notmember(Pred, Liste) :-
    notmember_x(Liste, Pred).

notmember_x([], _).
notmember_x([ Pred | RPreds ], APred) :-
    not(Pred = APred),
    notmember_x(RPreds, APred).
```

3. 3-PROLOG$_R$

```
% srg(+BUTS, -REPONSE, +PROGRAMME).
srg(Buts, Rep, Prog) :-
    new_vars(Buts, NButs),       % defini dans le programme "clause_prog/3".
    sr(NButs, Prog),
    Rep = NButs.

% solve_refl(+BUTS, +PROGRAMME).
solve_refl(Buts, Prog) :-
    functor(Buts, Pred),
    (Pred = true/0 ->
        true;
        appel_reflexion_implicite(Buts, NButs),
        selectionne_but(NButs, But, RRes, Prog),
        reduit_refl(But, Corps, RRes, TRes, Prog),
        nresolvante(Corps, TRes, NRes),
        solve_refl(NRes, Prog)).


reduit_refl(But, Corps, RRes, TRes, Prog) :-
    functor(But, Pred, Arite),
    (Pred/Arite = call/1 ->
        But = call(ABut),
        Corps = ABut,
        TRes = RRes;
        (reflexif_explicite_R(Pred/Arite) ->
            functor(But, Pred, Arite),
            Corps = true,
            But =.. ListeBut,
            append(ListeBut, [ RRes, TRes ], NListeBut),
            NBut =.. NListeBut,
            metavar(NBut);
            TRes = RRes,
            (system_res(Pred/Arite) ->
                Corps = true,
                call(But);
                clause_prog(But, Corps, Prog)))).

system_res(Pred) :- system(Pred).

appel_reflexion_implicite(Res, NRes) :-
    appel_refl_imp([], Res, NRes).

appel_refl_imp(Liste, Res, NRes) :-
    ((reflexif_implicite_R(P/A),
        notmember(P/A, Liste)) ->
        functor(But, P, A),
        But =.. ListeBut,
        append(ListeBut, [Res, Res_x], NListeBut),
        NBut =.. NListeBut,
        metavar(NBut),
        appel_refl_imp([P/A|Liste], Res_x, NRes);
        NRes = Res).
```

4. UNIFY/5 ET UNIFY/4

```
% unify(+TERME1, +TERME2, +UNIFICATIONS, -NOUVELLE_UNIFICATIONS).
% TERME1 doit etre un terme clos.
% TERME2 doit etre un terme non-clos.

% UNIFICATIONS contient des unifications de la forme (v(?, ?)=Terme).
% Terme est soit un terme clos soit une meta-variable libre.
% Terme ne peut pas etre en aucun cas une meta-variable v(?, ?).

% Si la meta-variable v(?, No) n'est pas presente dans UNIFICATIONS,
% cela veut dire que la variable du niveau objet est libre.

% Si Terme est une meta-variable libre, cela veut dire qu'il existe
% un lien d'unification entre cette variable et une autre dans
% UNIFICATIONS.

% unify(+TERME1, ?TERME2, +UNIFICATION, -UNIFICATION).
unify(T1, T2, Niv, Unif, NUnif) :-
    ((T1 = T2) ->
        NUnif = Unif;
        (variable(T1, Niv) ->
            unify_var1(T1, T2, Niv, Unif, NUnif);
            unify_atom1(T1, T2, Niv, Unif, NUnif))).

% T1 n'est pas une variable.
unify_atom1(T1, T2, Niv, Unif, NUnif) :-
    (atomic(T1) ->
        variable(T2, Niv),                % T2 est une variable.
        (member((T2=T6), Unif) ->
            unify_term(T1, T6, Niv, Unif, NUnif);
            NUnif = [(T2=T1)|Unif]);
            (variable(T2, Niv) ->
                unify(T2, T1, Niv, Unif, NUnif);
                functor(T1, F, Arite),
                functor(T2, F, Arite),
                unify(Arite, T1, T2, Niv, Unif, NUnif))).

% T1 est une variable.
unify_var1(T1, T2, Niv, Unif, NUnif) :-
    (member((T1=T3), Unif) ->
        unify_term(T2, T3, Niv, Unif, NUnif);
        unify_var1_free(T1, T2, Niv, Unif, NUnif)).

% T2 est un terme ou une meta-variable.
% dans la liste d'unifications.
unify_term(T1, T2, Niv, Unif, NUnif) :-
    (free_metavar(T2) ->
        unify_free(T1, T2, Niv, Unif, NUnif);
        unify(T1, T2, Niv, Unif, NUnif)).

% T2 est une meta-variable.
unify_free(T1, T2, Niv, Unif, NUnif) :-
    (variable(T1, Niv) ->
        unify_free2_var1(T1, T2, Niv, Unif, NUnif);
        mv(Liste) = T2,
        unify_metavars_atom(Liste, T1, Niv, Unif, NUnif)).

% T1 est une variable.
unify_free2_var1(T1, T2, Niv, Unif, NUnif) :-
```

```
    mv(List2) = T2,
    (member((T1=T3), Unif) ->
        (free_metavar(T3) ->
            mv(List1) = T3,
            append(List1, List2, List3),
            unify_metavars_free(List3, List3, Niv, Unif, NUnif);
            unify_metavars_atom(List2, T3, Niv, Unif, NUnif));
            v(Niv, No) = T1,
            List4 = [ No | List2 ],
            unify_metavars_free(List4, List4, Niv, Unif, NUnif)).

% LISTE1 est la liste des variables sur laquelle le changement doit se faire.
% LISTE2 est la liste qui doit apparaitre dans les nouvelles meta-variables.
% unify_metavars_free(+LISTE1, +LISTE2, +NIV, +UNIF, -NUNIF).

unify_metavars_free([], _, _, Unif, Unif).
unify_metavars_free([ No | RNos ], List, Niv, Unif,
    [ (v(Niv, No)=mv(List)) | RUnif ]) :-
    unify_metavars_free(RNos, List, Niv, Unif, RUnif).

% LISTE1 est la liste des variables sur laquelle le changement doit se faire.
% Les variables de LISTE doivent etre instanciees a TERME.
% unify_metavars_atom(+LISTE, +TERME, +NIV, +UNIF, -NUNIF).

unify_metavars_atom([], _, _, Unif, Unif).
unify_metavars_atom([ No | RNos ], Term, Niv, Unif,
    [ (v(Niv, No) = Term) | RUnif ]) :-
    unify_metavars_atom(RNos, Term, Niv, Unif, RUnif).

% T1 est une variable libre.
unify_var1_free(T1, T2, Niv, Unif, NUnif) :-
    (variable(T2, Niv) ->
        unify_var1_free_var2(T1, T2, Niv, Unif, NUnif);
        NUnif = [ (T1=T2) | Unif ]).

% T1 est une variable libre.
% T2 est une variable.
unify_var1_free_var2(T1, T2, Niv, Unif, NUnif) :-
    (member((T2=T4), Unif) ->
        (free_metavar(T4) ->
            unify_free(T1, T4, Niv, Unif, NUnif);
            NUnif = [(T1=T4)|Unif]);

            v(Niv, No1) = T1,
            v(Niv, No2) = T2,
            Val = mv([ No1, No2 ]),
            NUnif = [(T1=Val), (T2=Val) | Unif]).


unify(Arite, T1, T2, Niv, Unif, NUnif) :-
    (Arite = 0 ->
        NUnif=Unif;
        arg(Arite, T1, Arg1),
        arg(Arite, T2, Arg2),
        NArite is Arite - 1,
        unify(Arg1, Arg2, Niv, Unif, Unif_x),
        unify(NArite, T1, T2, Niv, Unif_x, NUnif)).

% unify(+TERME, ?TERME, +NIVEAU, +UNIFICATIONS).

unify(But, NBut, Niv, Unif) :-
```

```
    (variable(But, Niv) ->
        (member((But=Terme), Unif) ->
            unify(Terme, NBut, Niv, Unif);
            NBut = But );
            (atomic(But) ->
                But = NBut;
                functor(But, F, Arite),
                functor(NBut, F, Arite),
                unify_x(Arite, But, NBut, Niv, Unif))).

unify_x(Arite, T1, T2, Niv, Unif) :-
    (Arite = 0 ->
        true;
        arg(Arite, T1, Arg1),
        arg(Arite, T2, Arg2),
        NArite is Arite - 1,
        unify(Arg1, Arg2, Niv, Unif),
        unify_x(NArite, T1, T2, Niv, Unif)).

free_metavar(mv([_|_])).

variable(v(Niv, _), Niv).
```

5. 3-PROLOG$_U$

```
% solveg_refl(+BUT, -NBUT, +NO, +NIV).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
solveg_refl(But, NBut, No, Niv, Prog) :-
    solve_refl(Niv, But, No, _, [], Unif, Prog),
    unify(But, NBut, Niv, Unif).

solve_ground_refl(But, NBut, No, Niv, Unif, Prog) :-
    solve_refl(Niv, But, No, _, [], Unif, Prog),
    unify(But, NBut, Niv, Unif).

nv(But, NBut, Niv) :-
    numbervars(But, Niv, 0, _),
    NBut = But.

solve_refl(Niv, Buts, No, NNo, Unif, NUnif, Prog) :-
    functor(Buts, Pred),
    (Pred = true/0 ->
        NNo = No,
        NUnif = Unif;
        selectionne_but(Niv, Buts, But, RRes, No, No_x, Unif, Unif_x, Prog),
        appel_reflexion_implicite(No_x, No_xx, Unif_x, Unif_xx),
        reduit_refl(Niv, But, Corps, No_xx, No_xxx, Unif_xx, Unif_xxx, Prog),
        nresolvente(Corps, RRes, NRes),
        solve_refl(Niv, NRes, No_xxx, NNo, Unif_xxx, NUnif, Prog)).


appel_reflexion_implicite(No, NNo, Unif, NUnif) :-
  appel_refl_imp([], No, NNo, Unif, NUnif).

appel_refl_imp(Liste, No, NNo, Unif, NUnif) :-
    ((reflexif_implicite_U(P/A),
      notmember(P/A, Liste)) ->
          functor(But, P, A),
          But =.. ListeBut,
          append(ListeBut, [u(No, Unif), u(No_x, Unif_x)], NListeBut),
          NBut =.. NListeBut,
          metavar(NBut),
          appel_refl_imp([P/A|Liste], No_x, NNo, Unif_x, NUnif);
          NUnif = Unif,
          NNo = No).

reduit_refl(Niv, But, Corps, No, NNo, Unif, NUnif, Prog) :-
    functor(But, Pred, Arite),
    (Pred/Arite = call/1 ->
        But = call(ABut),
        unify(ABut, NBut, Niv, Unif),
        Corps = NBut,
        NNo = No,
        NUnif = Unif;
        (reflexif_explicite_U(Pred/Arite) ->
            functor(But, Pred, Arite),
            Corps = true,
            unify(But, But_x, Niv, Unif),
            But_x =.. ListeBut,
            append(ListeBut, [ Niv, u(No, Unif), u(NNo, NUnif) ], NListeBut),
            NBut =.. NListeBut,
            metacall(NBut);
            (system_ground(Pred/Arite) ->
```

```
                Corps = true,
                exemplarisation(But, NBut, Unif, Niv),
                call(NBut),
                unify(But, NBut, Niv, Unif, NUnif),
                numbervars(NBut, Niv, No, NNo);
                functor(Tete, Pred, Arite),
                clause_prog(Tete, Corps, Prog),
                unify(But, Tete, Niv, Unif, NUnif),
                numbervars((Tete:-Corps), Niv, No, NNo)))).

system_ground(Pred) :- system(Pred).

selectionne_but(Niv, Buts, But, Reste, No, NNo, Unif, NUnif, Prog) :-
    functor(Buts, Pred),
    (Pred = ','/2 ->
        Buts = (A, B),
        selectionne_but(Niv, A, But, ResteA, No, NNo, Unif, NUnif, Prog),
        nresolvente(ResteA, B, Reste);
        (Pred = ifcut/3 ->
            Buts = (C -> D;E),
            (solve_refl(Niv, C, No, No_x, Unif, Unif_x, Prog) ->
                selectionne_but(Niv, D, But, Reste, No_x, NNo, Unif_x, NUnif, Prog);
                selectionne_but(Niv, E, But, Reste, No, NNo, Unif, NUnif, Prog));
                But = Buts,
                Reste = true,
                NNo = No,
                NUnif = Unif)).

numbervars(Terme, Niv, No, NNo) :-
    (Terme = v(Niv, No) ->
        NNo is No + 1;
        (atomic(Terme)  ->
            NNo = No;
            functor(Terme, _, Arite),
            numbervars(Arite, Terme, Niv, No, NNo))).

numbervars(Arite, Terme, Niv, No, NNo) :-
    (Arite = 0 ->
        NNo = No;
        arg(Arite, Terme, Arg),
        NArite is Arite - 1,
        numbervars(Arg, Niv, No, No_x),
        numbervars(NArite, Terme, Niv, No_x, NNo)).

% exemplarisation(+Terme, -NTerme, +Unif, +Niv, [], _).

exemplarisation(Terme, NTerme, Unif, Niv, LVars, NLVars) :-
    Terme = v(Niv, No) ->
    (member((v(Niv, No)=Var), Unif) ->
        (free_metavar(Var) ->
            (member((v(Niv, No)=Var1), LVars) ->
                NTerme = Var1,
            NLVars = LVars;
            NLVars = [(v(Niv, No)=X) | LVars],
            NTerme = X);
            exemplarisation(Var, NTerme, Unif, Niv, LVars, NLVars));
            (member((v(Niv, No)=Var2), LVars) ->
                NTerme = Var2,
                NLVars = LVars;
                NLVars = [(v(Niv, No)=Y) | LVars],
                NTerme = Y));
```

```
                ( atomic(Terme) ->
                    NTerme = Terme,
                    NLVars = LVars;
                    functor(Terme, F, Arite),
                    functor(NTerme, F, Arite),
                    exemplarisation(Arite, Terme, NTerme, Unif, Niv, LVars, NLVars)).


exemplarisation(Arite, T1, T2, Unif, Niv, L, NL) :-
    (Arite = 0 ->
        NL = L;
        arg(Arite, T1, Arg1),
        arg(Arite, T2, Arg2),
        NArite is Arite - 1,
        exemplarisation(Arg1, Arg2, Unif, Niv, L, L_x),
        exemplarisation(NArite, T1, T2, Unif, Niv, L_x, NL)).

free_metavar(mv([_|_])).

variable(v(Niv, _), Niv).
```

6. Méta-3-Prolog*

```
tpl_ground(But, NBut, No) :-
    exemplarisation(But, But_x, []),
    tpl(But_x, No),
    NBut = But_x.

tpl_ground(But, NBut) :-
    tpl_ground(But, NBut, 1).

% Pour le traitement de la pile a tous les etages.
always_treatment_stack :- fail.
% always_treatment_stack.

% Traitement de la pile de points de choix au 2ieme etage.
%treatment_stack :- fail.
treatment_stack.

prog_start([ref(0)]).

test(Exp) :- Exp, treatment_stack.
test(_)   :- always_treatment_stack.

% setup_prog(-b(BUTS_SET), +PROG, +EXPLICITE_STACK, -b(BUTS_D), +b(BUTS_F)).
setup_prog(Buts_set, Exp, N, Buts_d, Buts_f, Prog) :-
    (test(Exp) ->
        E = assume(explicite_stack);
        E = assume((explicite_stack:-fail))),
        Buts_set = (
        assume(front_goal(Buts_d)),
        assume(back_goal(Buts_f)),
        assume(step(N)),
        E,
        assume(reflex_goals([])),
        assume(recursive_goals(Liste_refl)),
        assume(num_stack(1)),
        assume(stack([])),
        assume(program(Prog)),
        assume(goal_stack(Buts_f)),
        troispl(true, true)),
        liste_refl_implicite(Liste_refl).


liste_refl_implicite(Liste) :-
    liste_refl_implicite([], Liste).

liste_refl_implicite(Liste, NListe) :-
    ((meta3pl(reflexif_implicite(P/A)),
        notmember(P/A, Liste)) ->
        functor(But, P, A),
        NListe = [ But | NListe_x],
        liste_refl_implicite([ P/A | Liste ], NListe_x);
        NListe = []).

%%%%%%%%%%%%%%%%%%%%%%%%%

tpl(Buts, Metaprog) :-
    tpl(Buts, true, 0, Metaprog).

tpl(Buts, No, Metaprog) :- tpl(No, Buts, true, 0, Metaprog).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%

tpl(Buts, Explicite_stack, N, Metaprog) :-
    copy(Buts, Buts_fin),
    prog_start(Prog),
    setup_prog(Liste, Explicite_stack, N, Buts, Buts_fin, Prog),
    meta3pl(Liste, Prog, Metaprog).

%%%%%%%%%%%%%%%%%%%%%%%%%

tpl(1, Buts, Explicite_stack, N, Metaprog) :-
    tpl(Buts, Explicite_stack, N, Metaprog).

tpl(No, Buts, Explicite_stack, N, Metaprog) :-
    No > 1,
    NNo is No - 1,
    prog_start(Prog),
    setup_prog(Buts_set, Explicite_stack, N, Buts, Buts, Prog),
    NN is N+1,
    tpl(NNo, Buts_set, fail, NN, Metaprog).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Autre Simulateur de 3-Prolog qui permet la montee dynamique
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

meta3pl(Buts) :- meta3pl(Buts, [ref(500)]).

meta3pl(Buts, Metaprog) :-
    meta3pl(Buts, [ref(-1), cl(step(-1), true, -1)], _, Metaprog).

meta3pl(Buts, Prog, Metaprog) :-
    meta3pl(Buts, Prog, _, Metaprog).

meta3pl(Buts, Prog, NProg, Metaprog) :-
    functor(Buts, Pred),
    (Pred = true/0 ->
        NProg = Prog;
        selectionne_but(Buts, But, RRes, Prog, Metaprog),
        reduit_refl(But, Corps, Prog, NProg1, Reflection, Metaprog),
        nresolvante(Corps, RRes, NRes1),
        reflection(Reflection, NRes1, NRes, NProg1, NProg2),
        meta3pl(NRes, NProg2, NProg, Metaprog)).

%%%%%%%%%%%%%%%%%%%%%%%%%

% reflection(+REFLECTION, +RESOLVENTE, -NOUV_RESOLVENTE, +PROG, -PROG).
reflection(nonreflect, Res, Res, Prog, Prog).
reflection(reflect, Res, NRes, Prog, NProg) :-
    clause_x(step(N), _, Prog),
    prog_start(NProg),
    NN is N + 1,
    setup_prog(NRes, fail, NN, Res, Res, Prog).
reflection(detruct(Res, Prog), _, Res, _, Prog).
%%%%%%%%%%%%%%%%%%%%%%%%%

% selectionne_but(+BUTS, -BUT, -RESTE, +PROG).
% Il y a execution du test de A dans (A->B;C).

selectionne_but(Buts, But, Reste, Prog, Metaprog) :-
    functor(Buts, Pred),
    (Pred = ','/2 ->
```

```
        Buts = (A, B),
        selectionne_but(A, But, ResteA, Prog, Metaprog),
        nresolvante(ResteA, B, Reste);
        (Pred = ifcut/3 ->
            Buts = (C -> D;E),
            (meta3pl(C, Prog, Metaprog) ->
                selectionne_but(D, But, Reste, Prog, Metaprog);
                selectionne_but(E, But, Reste, Prog, Metaprog));
                But = Buts,
                Reste = true)).

%%%%%%%%%%%%%%%%%%%%%%%%%

% nresolvante(+CORPS, +RESOLVENTE, -NOUV_RESOLVENTE).
nresolvante(Corps, Res, NRes) :-
    functor(Corps, Pred1),
    (Pred1 = true/0 ->
        NRes = Res;
        functor(Res,  Pred2),
        (Pred2 = true/0 ->
            NRes = Corps;
            NRes = (Corps, Res))).

%%%%%%%%%%%%%%%%%%%%%%%%%
test_refl(reflect_goal/1).
test_refl(metacall/1).
test_refl(callgoals/1).

reduit_refl(But, Corps, Prog, NProg, Reflection, Metaprog) :-
    functor(But, Pred),
    (test_refl(Pred) ->
        reduit(But, Corps, Reflection),
        NProg = Prog;
        (Pred = new_metagoals/2 ->
            Corps = true,
            NProg = Prog,
            But = new_metagoals(NButs, P),
            Reflection = detruct(NButs, P);
            Reflection = nonreflect,
            (pred_property(Pred, system) ->
                Corps = true,
                call(But, Prog, NProg, Metaprog);
                clause(But, Corps, _, Prog, Metaprog),
                NProg = Prog))).


%reduit(But, NBut, Prog, Reflection) :-
reduit(But, NBut, Reflection) :-
    functor(But, Pred),
    (Pred = reflect_goal/1 ->
        NBut = But,
        Reflection = reflect;
        ((Pred = callgoals/1; Pred = metacall/1) ->
            NBut = But,
            Reflection = reflect;
            write('Erreur!'), nl,
            break)).

%%%%%%%%%%%%%%%%%%%%%%%%%

%filter_base(+PROGRAMME, +PREDICAT, -PROG1).
```

```
filter_base([], _, []).
filter_base([ ref(R) | RProg ], Pred, [ref(R) |Prog_pred]) :-
    filter_base(RProg, Pred, Prog_pred).
filter_base([cl(But, Corps, Ref) | RProg], Pred, Prog_pred) :-
    functor(But, Pred_but),
    (Pred = Pred_but->
        Prog_pred = [cl(But, Corps, Ref) | NProg_pred],
        filter_base(RProg, Pred, NProg_pred);
        filter_base(RProg, Pred, Prog_pred)).

exemplarisation(Terme, NTerme, Unif) :-
    exemplarisation(Terme, NTerme, Unif, [], _).

exemplarisation(Terme, NTerme, Unif, LVars, NLVars) :-
    Terme = v(No) ->
    (member((v(No)=Var), Unif) ->
        (free_metavar(Var) ->
            (member((v(No)=Var1), LVars) ->
                NTerme = Var1,
                NLVars = LVars;
                NLVars = [(v(No)=X) | LVars],
                NTerme = X);
                exemplarisation(Var, NTerme, Unif, LVars, NLVars));
                (member((v(No)=Var2), LVars) ->
                    NTerme = Var2,
                    NLVars = LVars;
                    NLVars = [(v(No)=X) | LVars],
                    NTerme = X));
                    ( atomic(Terme) ->
                        NTerme = Terme,
                        NLVars = LVars;
                        functor(Terme, F, Arite),
                        functor(NTerme, F, Arite),
                        exemplarisation(Arite, Terme, NTerme, Unif, LVars, NLVars)).

exemplarisation(Arite, T1, T2, Unif, L, NL) :-
    (Arite = 0 ->
        NL = L;
        arg(Arite, T1, Arg1),
        arg(Arite, T2, Arg2),
        NArite is Arite - 1,
        exemplarisation(Arg1, Arg2, Unif, L, L_x),
        exemplarisation(NArite, T1, T2, Unif, L_x, NL)).

variable(v(_)).

metanumbervars(Terme, No, NNo) :-
    (Terme = v(No) ->
        NNo is No + 1;
        (atomic(Terme)  ->
            NNo = No;
            functor(Terme, _, Arite),
            metanumbervars(Arite, Terme, No, NNo))).

metanumbervars(NthArg, Terme, No, NNo) :-
    (NthArg = 0 ->
        NNo = No;
        arg(NthArg, Terme, A1),
        metanumbervars(A1, No, No1),
        N1 is NthArg - 1,
        metanumbervars(N1, Terme, No1, NNo)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%

findall_refs(But, Prog, MProg, Refs) :-
    findall_refs(But, Prog, MProg, [], Refs).

findall_refs(But, Prog, MProg, AncRefs, NRefs) :-
    copy(But, NBut),
    ((clause_x(NBut, Ref, Prog, MProg),
    notmember(Ref, AncRefs)) ->
    NAncRefs = [ Ref | AncRefs ],
    NRefs = [ Ref | NRefs_x ],
    findall_refs(But, Prog, MProg, NAncRefs, NRefs_x);
    NRefs = []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clause/5 de 3-Prolog.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clause_x(But, Ref, Prog) :-
    clause(But, _, Ref, Prog, [ref(0)]).

clause_x(But, Ref, Prog, Metaprog) :-
    clause(But, _, Ref, Prog, Metaprog).

clause(But, Corps, Ref, Prog, Metaprog) :-
    clause3pl(But, Corps, Ref, Prog, Metaprog);
    clausesys3pl(But, Corps, Ref, Prog);
    metaclause(But, Corps, Ref, Metaprog).

metaclause(But, Corps, Ref, Metaprog) :-
    exemplarisation(Metaprog, NMetaprog, []),
    member(cl(But, Corps, Ref), NMetaprog).

% Empeche l'exemplarisation des clauses correspondantes.
pred_property(program/1, reified).
pred_property(goal/1, reified).
pred_property(clauses/1, reified).
pred_property(goal_stack/1, reified).
pred_property(stack/1, reified).
pred_property(front_goal/1, reified).
pred_property(back_goal/1, reified).
pred_property(no_stack/1, reified).
pred_property(reflex_goals/1, reified).
pred_property(recursive_goals/1, reified).

% Predicats de systeme 3-Prolog.
pred_property(Pred, system) :-
    (pred_system(Pred) ->
        true;
        system_tpl(Pred)).

system_tpl(Term) :- system(Term).
system_tpl(pred_property/2).
system_tpl(clause_x/3).
system_tpl(clause/5).
system_tpl(clause_x/4).
system_tpl(tpl/2).
system_tpl(meta3pl/2).
system_tpl(meta3pl/1).
system_tpl(meta3pl/4).
```

```
% Predicats de systeme reflexif.
pred_property(Pred, defined) :- pred_property(Pred, defined(_)).

pred_property(clause/2, defined([ 10 ])).
pred_property(clause/3, defined([ 20 ])).

clause3pl(clause(Tete, Corps),
          clause(Tete, Corps, _, P, MP), 10, P, MP).

clause3pl(clause(Tete, Corps, Ref),
          clause(Tete, Corps, Ref, P, MP), 20, P, MP).

clausesys3pl(But, Corps, Ref, Prog) :-
    member_unifiable(cl(But, Corps, Ref),  cl(ButU, CorpsU, Ref), Prog),
    functor(ButU, PredU),
    (pred_property(PredU, reified) ->
        But = ButU,
        Corps = CorpsU;
        copy(ButU, NButU),
        copy(CorpsU, NCorpsU),
        But = NButU,
        Corps = NCorpsU);
        cl(But, Corps, Ref).         %  Predicat qui permet d'acceder
                                     %  a la base de regles 3-Prolog*.

member_unifiable(Elem, PElem, [PElem | _]) :-
    not(not(Elem=PElem)).
member_unifiable(Elem, NElem, [_ | Reste]) :-
    member_unifiable(Elem, NElem, Reste).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicat 3-Prolog qui execute un predicat 3-Prolog systeme.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

call3pl(But, Prog) :- call3pl(But, Prog, _).

call(But, Prog, NProg, Metaprog) :-
    functor(But, Pred),
    (pred_system(Pred) ->
        call3pl(But, Prog, NProg, Metaprog);
        NProg = Prog,
        call(But)).

call3pl(call3pl_true(But, Prog), P, P, MP) :-
    copy(But, NBut),
    meta3pl(NBut, Prog, MP).

call3pl(meta3pl(Buts, Prog, NProg), P, P, MP) :-
    meta3pl(Buts, Prog, NProg, MP).

call3pl(meta3pl(Buts, Prog), P, P, MP) :-
    meta3pl(Buts, Prog, _, MP).

call3pl(call3pl(But, Prog), P, P, MP) :-
    meta3pl(But, Prog, MP).

call3pl(call(But, Prog, NProg), P, P, MP) :-
    call(But, Prog, NProg, MP).

call3pl(clause(But, Corps, Ref, Prog), P, P, MP) :-
```

```
    clause(But, Corps, Ref, Prog, MP).

call3pl(clauses_list(But, Refs, Prog), P, P, MProg) :-
    findall_refs(But, Prog, MProg, Refs).

call3pl(assume(Cl), Prog, [cl(T, C, NRef) | NProg], _) :-
    forget_ref(Ref, Prog, ProgP),
    NRef is Ref - 1,
    NProg = [ref(NRef) | ProgP],
    functor(Cl, Pred),
    (Pred = ':-'/2 ->
        Cl = (T:-C);
        T = Cl,
        C = true).

call3pl(forget(Cl), Prog, NProg, _) :-
    forget(Cl, Prog, NProg).


call3pl(assumed_clauses(Cls), Cls, Cls, _).

call3pl(new_assumed_clauses(Cls), _, Cls, _).

call3pl(create_new_stack_pt(Base, NBase), Cls, Cls, _) :-
    separate_base(Base, B1, B2),
    copy(B2, NB2),
    concat(B1, NB2, NBase).

forget(But, [ Cl | RCls ], NCls) :-
    functor(Cl, Pred_cl),
    (Pred_cl = ref/1 ->
        NCls = [ Cl | NRCls ],
        forget(But, RCls, NRCls);
        Cl = cl(T, C, R),
        functor(But, Pred1),
        (Pred1 = ':-'/2 ->
            But = (T:-C),
            NCls = RCls;
            functor(T, Pred2),
            (Pred1 = Pred2 ->
                But = T,
                NCls = RCls;
                NCls = [ cl(T, C, R) | NRCls ],
                forget(But, RCls, NRCls)))).

forget_ref(Ref, [ Cl | RCls ], NCls) :-
    functor(Cl, Pred),
    (Pred = ref/1  ->
        Cl = ref(Ref),
        NCls = RCls;
        NCls = [ Cl | NRCls ],
        forget_ref(Ref, RCls, NRCls)).

%separate_base(+BASE, -BASE1, -BASE2).
% BASE2 contient les buts a proteger.
separate_base(Base, Base1, Base2) :-
    clauses_protected(Liste_predicats),
    separate_base(Liste_predicats, Base, Base1, Base2).

%separate_base(LISTE_PREDICATS, +BASE, -BASE_PREDICATS, -RESTE_BASE).
separate_base([], Base, [], Base).
```

```
separate_base([Pred | RPreds], Base, Base_preds, RBase) :-
    filter_base(Base, Pred, Base_pred_x, RBase_x),
    separate_base(RPreds, RBase_x, Base_RPreds, RBase),
    append(Base_pred_x, Base_RPreds, Base_preds).

append([], L, L).
append([E|R], L, [E|NR]) :-
    append(R, L, NR).

%filter_base(+PROGRAMME, +PREDICAT, -PROG_PRED, -RESTE_PROG).
filter_base([], _, [], []).
filter_base([ ref(R) | RProg ], Pred, [ref(R) |Prog_pred], RProg_pred) :-
    filter_base(RProg, Pred, Prog_pred, RProg_pred).
filter_base([cl(But, Corps, Ref) | RProg], Pred, Prog_pred, RProg_pred) :-
    functor(But, Pred_but),
    (Pred = Pred_but->
        Prog_pred = [cl(But, Corps, Ref) | NProg_pred],
        filter_base(RProg, Pred, NProg_pred, RProg_pred);
        RProg_pred = [cl(But, Corps, Ref) | NRProg_pred],
        filter_base(RProg, Pred, Prog_pred, NRProg_pred)).

clauses_protected([ front_goal/1 ]).

pred_side_effect(write/1).
pred_side_effect(nl/0).
pred_side_effect(asserta/1).
pred_side_effect(assertz/1).

pred_system(clauses_list/3).
pred_system(calltpl/2).
pred_system(call3pl_true/2).
pred_system(meta3pl/3).
pred_system(meta3pl/2).
pred_system(call/3).
pred_system(clause/4).
pred_system(assume/1).
pred_system(forget/1).
pred_system(assumed_clauses/1).
pred_system(new_assumed_clauses/1).
pred_system(create_new_stack_pt/2).

% copy(+TERM1, -TERM2)

copy(Term1, Term2) :-
    check_vars_not_in_term(Term1, Term2),
    copy(Term1, Term2, [], _OutDict).

copy(Term1, Term2, InDict, OutDict) :-
    var(Term1) ->
    put_in_dict(InDict, OutDict, Term1, Term2);
    functor(Term1, Functor, Arity),
    functor(Term2, Functor, Arity),
    copy_x(Arity, Term1, Term2, InDict, OutDict).

copy_x(NthArg, Term1, Term2, InDict, OutDict) :-
    NthArg = 0 ->
    OutDict = InDict;
    arg(NthArg, Term1, A1),
    arg(NthArg, Term2, A2),
    copy(A1, A2, InDict, MidDict),
    N1 is NthArg -1,
```

```
    copy_x(N1, Term1, Term2, MidDict, OutDict).

% put_in_dict(+INDICTHEAD, +INDICTTAIL, +VARIABLE, -RENAMEDVARIABLE)

check_in_dict([OldVar-MatchVar|_Dict], Var1, MatchVar) :-
    OldVar == Var1.
check_in_dict([_X|Dict], Var1, Var2) :-
    check_in_dict(Dict, Var1, Var2).

put_in_dict(InDict, OutDict, Var1, Var2) :-
    (check_in_dict(InDict, Var1, Var2) ->
        OutDict = InDict;
        OutDict = [Var1-Var2|InDict]).

check_var_not_in_term(Var, Term) :-
    var(Term) ->
    not(Var == Term);
    functor(Term, _, Arity),
    check_var_not_in_term(Arity, Var, Term).

check_var_not_in_term(NthArg, Var, Term) :-
    (NthArg = 0 ->
        true;
        arg(NthArg, Term, A),
        check_var_not_in_term(Var, A),
        N1 is NthArg -1,
        check_var_not_in_term(N1, Var, Term)).

check_vars_not_in_term(Term1, Term2) :-
    var(Term1) ->
    check_var_not_in_term(Term1, Term2);
    functor(Term1, _, Arity),
    check_vars_not_in_term(Arity, Term1, Term2).

check_vars_not_in_term(NthArg, Term1, Term2) :-
    (NthArg = 0 ->
        true;
        arg(NthArg, Term1, A),
        check_vars_not_in_term(A, Term2),
        N1 is NthArg -1,
        check_vars_not_in_term(N1, Term1, Term2)).
```

7. Coeur de 3-Prolog*

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Meta-interprete 3-Prolog* simulant Prolog.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% troisProlog(+BUT, +PILE, +REFLEXIF).
% Selectionne la reduction dependant de But et de Pile.

troispl(Reflex) :-
    goal(But),
    stack(Pile),
    functor(But, Pred),
    (Pred = true/0 ->

     (goal_stack(true) ->
      (front_goal(UBut),
       back_goal(UBut);

       traite_pile(Pile),
       troispl(true));
      forget(goal(true)),
      troispl(true, true));

      (Pred = fail/0 ->
       traite_pile(Pile),
       troispl(true);

       (Pred = reflect_goal/1 ->
        forget(goal(But)),
        forget(clauses([])),
        But = reflect_goal(NBut),
        troispl(NBut, fail);

        (always_true(Pred, Reflex) ->
         reduction(But),
         troispl(true, true);

         (pred_property(Pred, system) ->
          (reduction(But),
           troispl(true, true);
           nonreduction(But),
           traite_pile(Pile),
           troispl(true));

          reduction(But, Corps),
          troispl(Corps, true)))))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% troispl(+CORPS, +REFLEX).
troispl(Corps, Reflex) :-
    nouv_resolvante(Corps),
    selectionne_but(But),
    program(Prog),
    selection_regle(But, Prog, Reflex),
    call_reflex_goals,
    call_recursive_goals,
    creation_stack_pt,
    troispl(Reflex).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clauses de reduction du Meta-interprete.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Permet d'eviter de rechercher la clause pour un but.
% always_true(+PREDICAT, +REFLEXIF)

always_true(Pred, Reflex) :-
    (always_true(Pred) ->
        true;
        reflexif_explicite(Pred),
        Reflex).

always_true(nl/0).
always_true(write/1).
always_true(assume/1).
always_true(metacall/1).
always_true(metacallsys/1).
%%%%%
always_true(clauses_list/3).
always_true(assumed_clauses/1).
always_true(new_assumed_clauses/1).
always_true(create_new_stack_pt/2).
%always_true(reflect_goal/1).

%%%%%%%%%%%%%%%%%%%%%%%%%

reduction(But, Corps) :-
    program(Prog),
    (explicite_stack ->
        forget(goal(But)),
        forget(clauses([Ref | _])),
        clause(But, Corps, Ref, Prog);
        forget(goal(But)),
        forget(clauses([])),
        clause(But, Corps, _, Prog)).

%%%%%%%%%%%%%%%%%%%%%%%%%

%reduction(+BUT) pour la reduction de faits.
reduction(But) :-
    functor(But, Pred),
    forget(goal(But)),
    forget(clauses([])),
    (reflexif_explicite(Pred) ->
        insert_reflex(reflect_goal(But));
        (Pred = metacall/1 ->
            metacall(NBut) = But,
            insert_reflex(callgoals(NBut));
            (Pred = metacallsys/1 ->
                But = metacallsys(NBut),
                insert_reflex(NBut);
                forget(program(P)),
                call(But, P, NP),
                assume(program(NP))))).


nonreduction(But) :-
    program(Prog),
    functor(But, Pred),
```

```
    (system(Pred) ->
        not(But);
        not(meta3pl(But, Prog))).

not(But) :- But -> fail; true.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clauses de construction de la nouvelle resolvante.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nouv_resolvante(+CORPS).

nouv_resolvante(Corps) :-
    forget(goal_stack(Res)),
    nresolvante(Corps, Res, NRes),
    assume(goal_stack(NRes)).

% nresolvante(+CORPS, +RESOLVENTE, -NOUV_RESOLVENTE).
nresolvante(Corps, Res, NRes) :-
    functor(Corps, Pred1),
    (Pred1 = true/0 ->
        NRes = Res;
        functor(Res,  Pred2),
        (Pred2 = true/0 ->
            NRes = Corps;
            NRes = (Corps, Res))).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clauses de selection de but du Meta-interprete.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

selectionne_but(But) :-
    forget(goal_stack(Buts)),
    program(Prog),
    selectionne_but(Buts, But, Reste, Prog),
    assume(goal_stack(Reste)),
    assume(goal(But)).

% selectionne_but(+BUTS, -BUT, -RESTE, +PROG).
% Il y a execution du test de A dans (A->B;C).

selectionne_but(Buts, But, Reste, Prog) :-
    functor(Buts, Pred),
    (Pred = ','/2 ->
        Buts = (A, B),
        selectionne_but(A, But, ResteA, Prog),
        nresolvante(ResteA, B, Reste);
        (Pred = ifcut/3 ->
            Buts = (C -> D;E),
            (meta3pl(C, Prog) ->
                selectionne_but(D, But, Reste, Prog);
                selectionne_but(E, But, Reste, Prog));
                But = Buts,
                Reste = true)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clauses de selection de regles reifiee du Meta-interprete.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%selection_regle(+NOUVEAU_BUT, +PROGRAMME).
% Vrai dans tous les cas.
```

```
% Il selectionne une regle seulement quand c'est possible.

selection_regle(But, Prog, Reflex) :-
    (explicite_stack ->
        functor(But, Pred),
        ((pred_property(Pred, system); always_true(Pred, Reflex)) ->
            assume(clauses([]));
            clauses_list(But, Refs, Prog),
            assume(clauses(Refs)));
            assume(clauses([]))).

%%%%%%%%%%%%%%%%%%%%%%%%%

creation_stack_pt :-
    explicite_stack ->
    forget(stack(Pile)),
    next_stack_pt(Pile, NPile),
    assume(stack(NPile));
    true.

next_stack_pt(Pile, NPile) :-
    clauses(Cls),
    functor(Cls, PCls),
    (PCls = []/0 ->
        NPile = Pile;
        Cls = [ _ | RCls ],
        functor(RCls, PRCls),
        (PRCls = []/0 ->
            NPile = Pile;
            forget(clauses([ _ | RCls ])),
            assume(clauses(RCls)),
            assumed_clauses(Base),
            incremente_no_pc,
            create_new_stack_pt(Base, NBase),
            NPile = [ NBase | Pile ],
            forget(clauses(RCls)),
            assume(clauses(Cls)))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clauses utiles a 3-Prolog.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

traite_pile([PC | RPC]) :-
    new_assumed_clauses(PC),
    clauses(Cls),
    functor(Cls, PCls),
    ( PCls = []/0 ->
        assume(stack(RPC));
        Cls = [ _ | RCls ],
        functor(RCls, PRCls),
        ( PRCls = []/0 ->
            assume(stack(RPC));
            next_stack_pt(RPC, NPC),
            assume(stack(NPC)))).

incremente_no_pc :-
    forget(num_stack(No)),
    Nouv_no is No + 1,
    assume(num_stack(Nouv_no)).

call_recursive_goals :-
```

```
    recursive_goals(Buts),
    call_list(Buts).

call_reflex_goals :-
    forget(reflex_goals(List)),
    call_reflex_goals(List),
    assume(reflex_goals([])).

call_reflex_goals([]).
call_reflex_goals([B | RBs]) :-
    B,
    call_reflex_goals(RBs).

insert_reflex(But) :-
    forget(reflex_goals(Lst)),
    assume(reflex_goals([But | Lst])).

insert_recursive(But) :-
    forget(recursive_goals(Lst)),
    assume(recursive_goals([But | Lst])).

call_list([]).
call_list([But | RButs]) :-
    But,
    call_list(RButs).

metacallsys(Buts) :- Buts.
reflect_goal(But) :- But.

tpl :-
    callgoals(true).

notpl :-
    metacallsys((goal_stack(Buts),
    new_metagoals(Buts, [ref(0)]))).

delete_reflex(But) :-
    forget(reflex_goals(List)),
    delete_list(But, List, NList),
    assume(reflex_goals(NList)).

delete_recursive(But) :-
    forget(recursive_goals(List)),
    delete_list(But, List, NList),
    assume(recursive_goals(NList)).
```

APPENDICE B

Différents programmes

1. TRANSPOSE/2 ET GRAND_PARENT/2 EN 2-PROLOG

```
% grand_parent(?GRAND-PARENT, ?ENFANT).

grand_parent(X, Z) :- grand_mere(X, Z).
grand_parent(X, Z) :- grand_pere(X, Z).

grand_pere(A, C) :- pere(A, B), parent(B, C).
grand_mere(A, C) :- mere(A, B), parent(B, C).

parent(A, B) :- pere(A, B).
parent(A, B) :- mere(A, B).

pere(daniel,lucie).
pere(andre, daniel).
pere(jean,denis).

% transpose( +LISTE(de listes), -LISTE(de listes) ).
transpose( M, []) :- nullrows( M).
transpose( M, [R | Rs]) :- colMat( R, M1, M), transpose( M1, Rs).

colMat([], [], []).
colMat([X|Y], [Xs|Ys], [[X|Xs] | T]) :- colMat( Y, Ys, T).

nullrows( [] ).
nullrows( [[] | Rows] ) :- nullrows( Rows ).
```

2. VAR/1, NONVAR/1, ==/2 ET FINDALL/4 EN 3-PROLOG$_U$

```
reflexif_implicite_U(_) :- fail.

reflexif_explicite_U(var/1).
reflexif_explicite_U('=='/2).
reflexif_explicite_U(nonvar/1).
reflexif_explicite_U(findall/4).

findall(Struct, But, Liste, Prog, Niv, u(No, Unif), u(No, [(Liste=NListe)|Unif])) :-
    findall(But, Prog, Niv, u(No, Unif), [], Liste_unif),
    construit_liste(Liste_unif, Niv, Struct, NListe).

% findall(+BUT, +PROGRAMME, +NIVEAU, +UNIF, +LISTE, -NOUV_LISTE).
findall(But, Prog, Niv, u(No, Unif), Liste, NListe) :-
    ((solve_refl(Niv, But, No, NNo, Unif, NUnif, Prog),
    notmember(u(NNo, NUnif), Liste)) ->
```

```
    NListe = [ u(NNo, NUnif) | NListe_x ],
    Liste_x = [ u(NNo, NUnif) | Liste ],
    findall(But, Prog, Niv, u(No, Unif), Liste_x, NListe_x);
    NListe = []).

construit_liste([], _, _, []).
construit_liste([ u(_, Unif) | RUnifs ], Niv, Struct,[Struct_unif|RStruct_unif]) :-
    unify(Struct, Struct_unif, Niv, Unif),
    construit_liste(RUnifs, Niv, Struct, RStruct_unif).


var(v(Niv,No), Niv, u(N, Unif), u(N, Unif)) :-
    (member((v(Niv, No)=Term), Unif) ->
        functor(Term, Pred),
        not(Pred = mv/1);
        No =< N).

nonvar(Term1, Niv, u(N, Unif), u(N, Unif)) :-
    (Term1 = v(Niv, No) ->
        (member((v(Niv, No)=Term2), Unif) ->
            functor(Term2, Pred),
            Pred = mv/1;
            No >= N);
            true).

'=='(Terme1, Terme2, Niv, u(N, Unif), u(N, Unif)) :-
    unify(Terme1, NTerme1, Niv, Unif),
    unify(Terme2, NTerme2, Niv, Unif),
    NTerme1 = NTerme2.
```

3. Coupe-choix en 3-Prolog*

```
getbacktrack(No) :- num_stack(No), cut_floor.

cutbacktrack(No) :-
    (No = 0 -> true;
        stack(Pile),
        functor(Pile, Pred),
        (Pred = []/0 ->
            true;
            forget(stack(Pile)),
            cutbacktrack(Pile, No, NPile),
            assume(stack(NPile)))),
            cut_floor.

cutbacktrack([], _, []).
cutbacktrack([ Pt | RPile ], No, NPile) :-
    clause_x(num_stack(Num_stack),_, Pt),
    (Num_stack = No ->
        NPile = RPile;
        cutbacktrack(RPile, No, NPile)).

recherche_cc :-
    forget(goal_stack(Res)),
    num_stack(No),
    forget(goal(But)),
    recherche_cc(But, NBut, No),
    recherche_cc(Res, NRes, No),
    assume(goal(NBut)),
    assume(goal_stack(NRes)).

recherche_cc(Res, NRes, No) :-
    functor(Res, Pred1),
    (Pred1 = '!'/0 ->
        NRes = cutbacktrack(No);
        ((Pred1 = ';'/2; Pred1 = ','/2) ->
            Res =.. [ F, C, D],
            NRes =.. [F, NC, ND],
            recherche_cc(C, NC, No),
            recherche_cc(D, ND, No);
            (Pred1 = ifcut/3 ->
                Res = (C->D;F),
                recherche_cc(D, ND, No),
                recherche_cc(F, NF, No),
                NRes = (C->ND;NF);
                NRes = Res))).
```

RÉFÉRENCES

[Abr87] *Abstract interpretation of declarative languages*, s. abramsky and c. hankin, editors, ellis horwood ed., 1987.

[AK91] H. Ait-Kaci, *Warren's abstract machine: A tutorial reconstruction*, MIT Press, Cambridge, M.A., 1991.

[B+76] L. Beckman et al., *A partial evaluator, and its use as a programming tool*, Artificial Intelligence **7** (1976), no. 4, 319–357.

[Baw88] A. Bawden, *Reification without Evaluation*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, 1988, pp. 342–351.

[BBP+93] B. L. Bowen, L Byrd, F C N Pereira, L M Pereira, and D H D Warren, *Sicstus prolog user's manual*, Draft version, August 1993.

[BCL+88] M. Bugliesi, M. Cavalieri, E. Lamma, P. Mello, A. Natali, and F. Russo, *Flexibility and efficiency in a prolog programming environment: Exploiting meta-programming and partial evaluation*, Proceeding of the 5th Annual ESPRIT Conference, Brussels, 1988.

[BdSM91] M. Bruynooghe, D. de Schreye, and B. Martens, *A general criterion for avoiding infinite unfolding during partial deduction of logic programs*, Logic Programming: International Symposium (V. Saraswat and K. Ueda, eds.), Cambridge, MA: MIT Press, 1991, pp. 117–131.

[BL89] K. Benkerimi and J.W. Lloyd, *A procedure for the partial evaluation of logic programs*, Tech. Report TR-89-04, Department of Computer Science, University of Bristol, Bristol, England, May 1989.

[BL90] K. Benkerimi and J.W. Lloyd, *A partial evaluation procedure for logic programs*, Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990 (S. Debray and M. Hermenegildo, eds.), Cambridge, MA: MIT Press, 1990, pp. 343–358.

[Bou92] D.Y. Boulanger, *Deep logic program transformation using abstract interpretation*, Logic Programming, Irkutsk, Russia, September 1990, and St. Petersburg, Russia, September 1992 (Lecture Notes in Artificial Intelligence, vol. 592) (A. Voronkov, ed.), Berlin: Springer-Verlag, 1992, pp. 79–101.

[Bow82] Bowen, Logic Programming, ch. Amalgamating Language and Metalanguage in Logic Programming, academic press, london and new york ed., 1982.

[Bow85] Bowen, *Meta-Level Programming and Knowledge Representation*, New Generation Computing **3** (1985), 359–383.

[BR89] M. Bugliesi and F. Rossi, *Partial evaluation in Prolog: Some improvements about cut*, Logic Programming: Proceedings of the North American Conference 1989, Cleveland, Ohio, October 1989 (E.L. Lusk and R.A. Overbeek, eds.), Cambridge, MA: MIT Press, 1989, pp. 645–660.

[CC77] P. Cousot and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints*, Proceedings 4th ACM POPL Symposium, 1977, pp. 238–252.

[CD91] C. Consel and O. Danvy, *Static and dynamic semantics processing*, Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, New York: ACM, January 1991, pp. 14–24.

[CD93] C. Consel and O. Danvy, *Tutorial notes on partial evaluation*, Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993, ACM, New York: ACM, 1993, pp. 493–501.

[CFL+88] P. Coscia, P. Francheschi, G. Levi, G. Sardu, and L. Torre, *Object Level Reflection of Inference Rule by Partial Evaluation*, In Maes and Nardi [MN88], pp. 313–327.

[CK91] C. Consel and S.C. Khoo, *Semantics-directed generation of a Prolog compiler*, Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP '91, Passau, Germany, August 1991 (Lecture Notes in Computer Science, vol. 528) (J. Maluszyński and M. Wirsing, eds.), Berlin: Springer-Verlag, 1991, pp. 135–146.

[CL89] Stefania Costantini and Gaetano Aurelio Lanzarone, *A metalogic programming language*, Proceedings of the 6th International Conference in Logic Programming Workshop, 1989.

[Cos90] Stefania Costantini, *Semantics of a metalogic programming language*, International Journal of Foundations of Computer Science **1** (1990), no. 3.

[CvS89] M.H.M. Cheng, M.H. van Emden, and P.A. Strooper, *Complete sets of frontiers in logic-based program transformation*, Meta-Programming in Logic Programming (H. Abramson and M.H. Rogers, eds.), MIT Press, 1989, pp. 283–297.

[CW89] D. Chan and M. Wallace, *A treatment of negation during partial evaluation*, Meta-Programming in Logic Programming (H. Abramson and M.H. Rogers, eds.), Cambridge, MA: MIT Press, 1989, pp. 299–318.

[Dan88] O. Danvy, *Across the bridge between reflection and partial evaluation*, Partial Evaluation and Mixed Computation (D. Bjørner, A.P. Ershov, and N.D. Jones, eds.), Amsterdam: North-Holland, 1988, pp. 83–116.

[dJGS93] Neil d. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial evaluation and automatic program generation*, Prentice Hall, 1993.

[Dr88] O. Danvy and K. Malmkjær, *Intensions and Extensions in a Reflective Tower*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, 1988.

[dWG92a] D. A. de Waal and J. Gallagher, *Specialising a theorem prover*, Tech. Report CSTR-92-33, University of Bristol, November 1992.

[dWG92b] D. A. de Wall and J. Gallagher, *Specialisation of a unification algorithm*, Logic Program Synthesis and Transformation: Workshops in Computing (Manchester 1991) (T. Clement and K.-K. Lau, eds.), Springer-Verlag, 1992, pp. 205–221.

[Ers77] A.P. Ershov, *On the partial computation principle*, Information Processing Letters **6** (1977), no. 2, 38–41.

[Ers80] A.P. Ershov, *Mixed computation: Potential applications and problems for study*, Mathematical Logic Methods in AI Problems and Systematic Programming, Part 1, Vil'nyus, USSR, 1980, (In Russian), pp. 26–55.

[Esh86] K. Eshghi, *Meta-language in logic programming*, Ph.D. thesis, Department of Computing, Imperial College, 1986.

[FA88] D.A. Fuller and S. Abramsky, *Mixed computation of Prolog programs*, New Generation Computing **6** (1988), no. 2,3, 119–141.

[Fef62] S. Feferman, *Transfinite Recursive Progressions of Axiomatic Theories*, Journal Symbolic Logic **27** (1962), 259–316.

[FF88] H. Fujita and K. Furukawa, *A self-applicable partial evaluator and its use in incremental compilation*, New Generation Computing **6** (1988), no. 2,3, 91–118.

[Fuj87] H. Fujita, *An algorithm for partial evaluation with constraints*, Technical Report TM-367, ICOT, Tokyo, Japan, 1987.

[Fuj88] H. Fujita, *Abstract interpretation and partial evaluation of Prolog programs*, ICOT technical memorandum 484, Institute for New Generation Computer Technology, Tokyo, 1988.

[Fut71] Y. Futamura, *Partial evaluation of computation process – an approach to a compiler-compiler*, Systems, Computers, Controls **2** (1971), no. 5, 45–50.

[FW84] D.P. Friedman and M. Wand, *Reification: Reflection without Metaphysics*, Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, August 1984, pp. 348–355.

[Gal86] J. Gallagher, *Transforming logic programs by specialising interpreters*, ECAI-86. 7th European Conference on Artificial Intelligence, Brighton Centre, United Kingdom, Brighton: European Coordinating Committee for Artificial Intelligence, 1986, pp. 109–122.

[Gal91] J. Gallagher, *A system for specialising logic programs*, Tech. Report TR-91-32, University of Bristol, November 1991.

[Gal92] J. Gallagher, *Static analysis for logic program specialisation*, WSA–92: Analyse Statique (M. Billaud et al., ed.), Université de Bordeaux I, 1992, pp. 285–294.

[Gal93] J. Gallagher, *Tutorial on specialisation of logic programs*, Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993, New York: ACM, 1993, pp. 88–98.

[GB90] J. Gallagher and M. Bruynooghe, *Some low-level source transformations for logic programs*, Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium (M. Bruynooghe, ed.), Department of Computer Science, KU Leuven, Belgium, 1990, pp. 229–246.

[GB91] J. Gallagher and M. Bruynooghe, *The derivation of an algorithm for program specialisation*, New Generation Computing, vol. 6, 1991.

[GCS88] J. Gallagher, M. Codish, and E. Shapiro, *Specialisation of Prolog and FCP programs using abstract interpretation*, New Generation Computing **6** (1988), no. 2,3, 159–186.

[GS89] P. Gardner and J. C. Shepherdson, *Unfold-fold transformations of logic programs*, Tech. Report PM-89-01, School of Mathematics, University of Bristol, 1989.

[Gur92] C. A. Gurr, *Specialising the ground representation in the logic programming language goedel*, Tech. Report CSTR-92-30, Department of Computer Science, University of Bristol, 1992.

[Gur94a] C A Gurr, *A self-applicable partial evaluator for the logic programming language goedel*, Ph.D. thesis, Department of Computer Science, University of Bristol, January 1994.

[Gur94b] C. A. Gurr, *A self-applicable partial evaluator for the logic programming language goedel (extended abstract)*, Tech. Report Draft copy, Human Communication Research Centre, University of Edinburgh, 1994.

[Har77] A. Haraldsson, *A program manipulation system based on partial evaluation*, Ph.D. thesis, Linköping University, Sweden, 1977, Linköping Studies in Science and Technology Dissertations 14.

[Har78] A. Haraldsson, *A partial evaluator, and its use for compiling iterative statements in Lisp*, Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona, New York: ACM, 1978, pp. 195–202.

[HL88a] P. M. Hill and J. W. Lloyd, *Analysis of meta-programs*, Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, The MIT Press, June 1988.

[HL88b] P. M. Hill and J. W. Lloyd, *Meta-Programming for Dynamic Knowledge Bases*, Technical Report CS-88-18, Department of Computer Science, University of Bristol, 1988.

[HL92] P. M. Hill and J. W. Lloyd, *The gödel programming language*, Tech. Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992, Revised in May 1993. To be published by MIT Press.

[JF92] S. Jefferson and D.P. Friedman, *A Simple Reflective Interpreter*, Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture (A. Yonezawa and B. Smith, eds.), RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992, pp. 48–58.

[Jon88] N. D. Jones, *Challenging problems in partial evaluation and mixed computation*, New Generation Computing, OHSMHA LTD. and Springer-Verlag, 1988, pp. 291–302.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard, *An experiment in partial evaluation: The generation of a compiler generator*, Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202) (J.-P. Jouannaud, ed.), Berlin: Springer-Verlag, 1985, pp. 124–140.

[KC84] K.M. Kahn and M. Carlsson, *The compilation of Prolog programs without the use of a Prolog compiler*, International Conference on Fifth Generation Computer Systems, Tokyo, Japan, Tokyo: Ohmsha and Amsterdam: North-Holland, 1984, pp. 348–355.

[KF86] T. Kanamori and H. Fujita, *Unfold/fold transformation of logic programs with counters*, Tech. Report TR-179, ICOT, May 1986.

[Kle52] S.C. Kleene, *Introduction to metamathematics*, Princeton, NJ: D. van Nostrand, 1952.

[Kom81] H.J. Komorowski, *A specification of an abstract Prolog machine and its application to partial evaluation*, Ph.D. thesis, Linköping University, Sweden, 1981, Linköping Studies in Science and Technology Dissertations 69.

[Kom82] H.J. Komorowski, *Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog*, Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, 1982, pp. 255–267.

[Kom89] J. Komorowski, *Synthesis of programs in the framework of partial deduction*, Reports on Computer Science and Mathematics, Ser. A 81, Department of Computer Science, Åbo Akademi, Finland, 1989.

[Kom92] J. Komorowski, *An introduction to partial deduction*, Meta-Programming in Logic, Uppsala, Sweden, June 1992 (Lecture Notes in Computer Science, vol. 649) (A. Pettorossi, ed.), Berlin: Springer-Verlag, 1992, pp. 49–69.

[Kur87] P. Kursawe, *How to invent a Prolog machine*, New Generation Computing **5** (1987), 97–114.

[Kur88] P. Kursawe, *Pure partial evaluation and instantiation*, Partial Evaluation and Mixed Computation (D. Bjørner, A.P. Ershov, and N.D. Jones, eds.), Amsterdam: North-Holland, 1988, pp. 283–298.

[Lak89] A. Lakhotia, *A workbench for developing logic programs by stepwise enhancement*, Ph.D. thesis, Department of Computer Engineering and Science, Case Western Reserve University, 1989.

[Lak90] A. Lakhotia, *To pe or not to pe*, Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium (M. Bruynooghe, ed.), Department of Computer Science, KU Leuven, Belgium, 1990, pp. 218–228.

[Lam89] J. Lam, *Control structures in partial evaluation of pure Prolog*, Master's thesis, Department of Computational Science, University of Saskatchewan, Canada, 1989.

[Llo87] J. W. Lloyd, *Foundations of logic programming: 2nd edition*, Springer-Verlag, 1987.

[LMN91] Evelina Lamma, Paola Mello, and Antonio Natali, *Reflection mechanisms for combining prolog databases*, Software-Practice and Experience, volume 21(6), 1991, pp. 603–624.

[LO91] K. Läufer and M. Odersky, *Reflection in Type Systems*, Informal Proceedings of the Second Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'91, October 1991.

[Lom67] L.A. Lombardi, *Incremental computation*, Advances in Computers, vol. 8 (F.L. Alt and M. Rubinoff, eds.), New York: Academic Press, 1967, pp. 247–333.

[LR64] L.A. Lombardi and B. Raphael, *Lisp as the language for an incremental computer*, The Programming Language Lisp: Its Operation and Applications (E.C. Berkeley and D.G. Bobrow, eds.), Cambridge, MA: MIT Press, 1964, pp. 204–219.

[LS88] G. Levi and G. Sardu, *Partial evaluation of metaprograms in a multiple worlds logic language*, New Generation Computing **6** (1988), no. 2,3, 227–247.

[LS90] A. Lakhotia and L. Sterling, *How to control unfolding when specializing interpreters*, New Generation Computing **8** (1990), no. 1, 61–70.

[LS91a] A. Lakhotia and L. Sterling, *ProMiX: A Prolog partial evaluation system*, The Practice of Prolog (L. Sterling, ed.), Cambridge, MA: MIT Press, 1991, pp. 137–179.

[LS91b] J.W. Lloyd and J.C. Shepherdson, *Partial evaluation in logic programming*, Journal of Logic Programming **11** (1991), 217–242.

[Mae87] P. Maes, *Computational reflection*, Ph.D. thesis, Vrije Universiteit Brussel, 1987.

[MB93] T. Mogensen and A. Bondorf, *Logimix: a self-applicable partial evaluator for prolog*, (K.K. Lau and T. Clement, eds.), vol. Logic Program Synthesis and Transformation, Springer-Verlag, 1993.

[MCD91] J. Malenfant, P. Cointe, and C. Dony, *Reflection in Prototype-Based Object-Oriented Programming Languages*, Informal Proceedings of the Second Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'91, October 1991.

[MD93] Jacques Malenfant and François-Nicola Demers, *évaluation partielle pour l'implantation efficace de la réflexion de comportement*, Les actes de la IIième Journées Francophones de Programmation Logique (JFPL'93) (Nîmes-Avignon, France), 1993.

[MN88] P. Maes and D. Nardi (eds.), North-Holland, 1988.

[Nil93] U. Nilsson, *Towards a methodology for the design of abstract machines*, Journal of Logic Programming **16** (1993), no. 1, 2, 163–189.

[Owe89] S. Owen, *Issues in the partial evaluation of meta-interpreters*, Meta-Programming in Logic Programming (H. Abramson and M.H. Rogers, eds.), Cambridge, MA: MIT Press, 1989, pp. 319–340.

[Per85] D. Perlis, *Languages with self-reference I: Foundations (or: We can have everything in first-order logic!)*, Artificial Intelligence **25** (1985), 301–322.

[Per88] D. Perlis, *Languages with self-reference II: Knowledge, Belief, and Modality*, Artificial Intelligence **34** (1988), 179–212.

[PP90] M. Proietti and A. Pettorossi, *Construction of efficient logic programs by loop absorption and generalisation*, Proceedings of the 2nd Workshop on Meta-Programming, 1990, pp. 57–81.

[PP91] M. Proietti and A. Pettorossi, *Semantics preserving transformation rules for prolog*, SIGPLAN NOTICES, vol. 26, September 1991.

[Pre90] S. Prestwich, *Partial evaluation by unfold/fold with generalisation*, ECRC GmbH, Munich, Germany, November 1990.

[Pre92] Steven Prestwich, *The paddy user guide – draft version*, Tech. report, European Computer-Industry Research Center GmbH, 1992.

[Pre93] S. Prestwich, *Online partial deduction of large programs*, Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993, New York: ACM, 1993, pp. 111–118.

[Sah90] D. Sahlin, *The Mixtus approach to automatic partial evaluation of full Prolog*, Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990 (S. Debray and M. Hermenegildo, eds.), Cambridge, MA: MIT Press, 1990, pp. 377–398.

[Sah91] D. Sahlin, *An automatic partial evaluator for full prolog*, Ph.D. thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991, Report TRITA-TCS-9101.

[Sat90] T. Sato, *An equivalence preserving first order unfold/fold transformation system*, Algebraic and Logic Programming, Second International Conference, Nancy, France, October 1990. (Lecture Notes in Computer Science, vol. 463) (H. Kirchner and W. Wechler, eds.), Berlin: Springer-Verlag, 1990, pp. 173–188.

[SB86] L. Sterling and R.D. Beer, *Incremental flavor-mixing of meta-interpreters for expert system construction*, Proc. 3rd Symposium on Logic Programming, Salt Lake City, Utah, New York: IEEE Computer Society, 1986, pp. 20–27.

[SB89] L. Sterling and R.D. Beer, *Metainterpreters for expert system construction*, Journal of Logic Programming **6** (1989), 163–178.

[Sek93] H. Seki, *Unfold/fold transformation of general logic programs for the well-founded semantics*, Journal of Logic Programming **16** (1993), no. 1, 2, 5–23.

[Smi82] B.C. Smith, *Reflection and Semantics in a Procedural Language*, Tech. Report 272, MIT Laboratory for Computer Science, 1982.

[Smi84] B.C. Smith, *Reflection and Semantics in Lisp*, Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, January 1984, pp. 23–35.

[SS86a] S. Safra and E. Shapiro, *Meta interpreters for real*, Information Processing 86, Dublin, Ireland (H.-J. Kugler, ed.), Amsterdam: North-Holland, 1986, pp. 271–278.

[SS86b] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

[SS88] P. Sestoft and H. Søndergaard, *A bibliography on partial evaluation*, Sigplan Notices **23** (1988), no. 2, 19–27.

[Sug90] Hiroyasu Sugano, *Meta and reflective computation in logic programs and its semantics*, Meta-90, Proceedings of the Second Workshop on Meta-Programming in Logic, 1990.

[SZ88] P. Sestoft and A.V. Zamulin, *Annotated bibliography on partial evaluation and mixed computation*, Partial Evaluation and Mixed Computation (D. Bjørner, A.P. Ershov, and N.D. Jones, eds.), Amsterdam: North-Holland, 1988, pp. 589–622.

[Tak86] A. Takeuchi, *Affinity between meta interpreters and partial evaluation*, Information Processing 86, Dublin, Ireland (H.-J. Kugler, ed.), Amsterdam: North-Holland, 1986, pp. 279–282.

[TF86] A. Takeuchi and K. Furukawa, *Partial evaluation of Prolog programs and its application to meta programming*, Information Processing 86, Dublin, Ireland (H.-J. Kugler, ed.), Amsterdam: North-Holland, 1986, pp. 415–420.

[TS83] H. Tamaki and T. Sato, *A transformation system for logic programs whick preserves equivalence*, Tech. Report TR-18, ICOT, August 1983.

[TS86] H. Tamaki and T. Sato, *A generalized correctness proof of the Unfold/Fold logic program transformation*, Tech. Report TR 86-04, Department of Information Science, Ibaraki University, June 1986.

[TW88] C. Talcott and R. W. Weyhrauch, *Partial evaluation, higher-order abstractions and reflection principles as system building tools*, Partial Evaluation and Mixed Computation IFIP (D. Bjorner, A. P. Ershov, and N. D. Jones, eds.), Elsevier Science Publishers (North-Holland), 1988.

[VD88] R. Venken and B. Demoen, *A partial evaluation system for Prolog: Some practical considerations*, New Generation Computing **6** (1988), no. 2,3, 279–290.

[Ven84] R. Venken, *A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation*, ECAI-84, Advances in Artificial Intelligence, Pisa, Italy (T. O'Shea, ed.), Amsterdam: North-Holland, 1984, pp. 91–100.

[vH89] F. van Harmelen, *The limitations of partial evaluation*, Logic-Based Knowledge Representation (P. Jackson, H. Reichgelt, and F. van Harmelen, eds.), Cambridge, MA: MIT Press, 1989.

[War83] D. H. D. Warren, *An abstract prolog instruction set*, Tech. Report 309, SRI Internationnal, 1983.

[Wey80] R.W. Weyhrauch, *Prolegomena to a Theory of Mechanized Formal Reasoning*, Artificial Intelligence **13** (1980), no. 1,2.

[Wey82] R.W. Weyhrauch, *An Example of FOL Using Metatheory*, 6th Conference on Automated Deduction, no. 138, Springer-Verlag, LNCS, 1982.

[WF86] M. Wand and D. P. Friedman, *The mystery of the tower revealed: a non-reflective description of the reflective tower*, Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming, August 1986, (also appears in [MN88], pp. 111–134).

Document Log:

Manuscript Version 4 — Avril 1994
Typeset by $\mathcal{AMS}$-LaTeX — 17 January 1995

Réflexion de comportement et évaluation partielle en Prolog

François-Nicola Demers

Laboratoire Incognito, D.I.R.O.
Université de Montréal, C.P. 6128
Succ. CENTRE-VILLE, Montréal, Québec, H3C 3J7, Canada
Tel. : (514) 343-6111 poste 1651
E-mail address: demers@iro.umontreal.ca