UNIVERSITÉ DE SHERBROOKE
Faculté des sciences appliquées
Département de génie électrique et de génie informatique

CONTRIBUTION À L'INTÉGRATION DE C++ ET DE PROLOG À TRAVERS LA MACHINE ABSTRAITE DE WARREN

LE SYSTÈME COP-COMPILÉ

Mémoire de maîtrise es sciences appliquées
Spécialité: génie informatique

Mounir EL MOKHTARI

Sherbrooke (Québec), Canada

Avril 1997

À la mémoire de l'illustre savant musulman AL KHAWARIZMI *(770 – 840 AC), fondateur de l'algèbre et développeur du système décimal. Son nom latinisé* (ALGORITHME) *est, sans aucun doute, présent à l'esprit de chaque informaticien.*

RÉSUMÉ

Ce mémoire présente le système COP-Compilé. C'est un système permettant l'intégration de la programmation procédurale en C++ et de la programmation logique en Prolog. Le noyau de ce système est un compilateur de Prolog basé sur la machine abstraite de Warren (WAM). Cette machine qui définit une organisation de la mémoire, un ensemble de registres et un jeu d'instructions adaptés à Prolog, est actuellement la référence par excellence dans le domaine d'implantation de compilateurs Prolog.

Ce projet de recherche s'insère dans le cadre d'une série de travaux qui visent à rapprocher la programmation procédurale et la programmation logique. L'objectif principal de ce projet est d'augmenter la performance d'une première version d'un compilateur réalisant l'intégration des deux styles de programmation où le code Prolog était interprété.

Le système COP-Compilé a réussi à augmenter la performance du système COP original et à améliorer ses possibilités en diminuant son temps d'exécution et en étendant les programmes Prolog qu'il est capable de reconnaître.

REMERCIEMENTS

Je remercie toutes les personnes dont l'aide a permis à ce projet de voir le jour et de progresser, et en particulier:

- mon directeur de recherche RUBEN GONZALEZ-RUBIO qui a suivi avec intérêt et confiance l'élaboration de ce travail;
- la communauté INTERNET qui a mis à ma disposition la plupart des références dont j'avais besoin;
- mon épouse RAHMA, MES PARENTS ainsi que toute MA FAMILLE pour leur patience, leur compréhension et surtout leurs sacrifices;
- tous MES AMIS au Canada pour leur soutien moral et financier. Je cite, parmi bien d'autres, IBRAHIMA, MOHAMMED, NABIL, ABDELILAH, SAÏD, LOTFI et AHMED.

TABLE DES MATIÈRES

1 INTRODUCTION **1**

1.1 Cadre du projet de recherche 1

1.2 Présentation de la problématique 3

1.3 Réalisations 3

1.3.1 Diminution du temps d'exécution 4

1.3.2 Extension des programmes Prolog reconnus 5

1.4 Grandes lignes du travail 5

2 LE SYSTÈME COP*i* **7**

2.1 Langages intégrés par COP*i* 8

2.1.1 Le langage C++ 8

2.1.2 Le langage Prolog 9

2.2 Le langage COP 11

2.3 Structure de COPi 13

2.3.1 Le séparateur COP 14

2.3.2 Le convertisseur COP 17

2.3.3 Le compilateur Prolog-COP 20

2.4 Schéma de compilation de COPi 21

3 L'AMÉLIORATION DE COPi 24

3.1 Limitations de COPi 25

3.1.1 Représentation interne des données 25

3.1.2 Ensemble des prédicats prédéfinis 26

3.1.3 Exécution du code Prolog 27

3.2 Améliorations à apporter 27

3.2.1 Objectifs de conception et de réalisation 27

3.2.2 Représentation interne des données 28

3.2.3 Ensemble des prédicats prédéfinis 30

3.2.4 Compilation de Prolog 32

4 LA MACHINE ABSTRAITE DE WARREN 34

4.1 Organisation de la mémoire dans la WAM 35

4.2 Ensemble des registres de la WAM 41

4.3 Jeu d'instructions de la WAM 42

4.3.1 Instructions de récupération des registres 44

4.3.2 Instructions de chargement des registres 49

4.3.3 Instructions de contrôle 51

4.3.4 Instructions d'indexation 52

5 LA CONCEPTION DE COP-COMPILÉ **55**

5.1 Structure du système COP-Compilé 56

5.1.1 Séparateur COP 56

5.1.2 Convertisseur COP 57

5.2 Compilateur Prolog-COP 57

5.2.1 Optimisateur Prolog 58

5.2.2 Compilateur Prolog-WAM 60

5.2.3 Émulateur WAM 60

5.3 Schéma de compilation de COP-Compilé 60

6 LA RÉALISATION DE COP-COMPILÉ **63**

6.1 Compilation de Prolog en WAM 63

6.1.1 Compilation du programme Prolog 64

6.1.2 Compilation d'un prédicat du programme 65

6.1.3 Compilation d'une clause du prédicat 65

6.2 Exécution du code WAM 72

6.2.1 Interface entre le code C++ et le code WAM 72

6.2.2 Émulation du code WAM 76

6.3 Tests et validation de COP-Compilé 80

6.3.1 Description des tests 81

6.3.2 Résultats des tests 83

6.3.3 Comparaison avec un Prolog classique 85

7 CONCLUSION **87**

7.1 Bilan de COP-Compilé 87

7.1.1 Diminution du temps d'exécution de COPi 88

7.1.2 Extension des programmes Prolog reconnus par COPi 88

7.2 Travaux futurs 89

7.2.1 Améliorations quantitatives 89

7.2.2 Améliorations qualitatives 90

7.2.3 Suite suggérée du projet 90

A Exemple complet **91**

A.1 Fichier source COP 91

A.2 Fichier Bridge 92

A.3 Fichier Prolog 92

A.4 Fichier C++ 92

A.5 Fichier des bridge functions 93

A.6 Fichier WAM 93

A.7 Fichier Byte_Code WAM 94

A.8 Résultat de l'exécution 96

B Méthodes de la classe TProlog **97**

C Méthodes de la classe TBridgeVariable **99**

D Programmes de test **101**

D.1 Le programme boresea 101

D.2 Le programme cre_env 103

D.3 Le programme general_unif 104

D.4 Le programme choice_point 106

D.5 Le programme deep_back 107

D.6 Le programme shallow_back 109

D.7 Le programme nrev 110

D.8 Le programme fib 112

TABLE DES FIGURES

2.1 La grammaire Prolog du séparateur COP 15

2.2 La grammaire C++ du séparateur COP 16

2.3 La grammaire C++ du convertisseur COP 18

2.4 Flux des données et schéma de compilation de COPi 23

4.1 Représentation interne des termes dans le *heap* 38

4.2 Représentation du heap pour p(Z, h(Z, W), f(W)). 40

4.3 Organisation de la mémoire dans la WAM 43

5.1 Flux des données dans le compilateur Prolog-COP de COP-Compilé 58

5.2 Flux des données et schéma de compilation de COP-Compilé 62

6.1 Code squelette d'une bridge-function 75

LISTE DES TABLEAUX

6.1 TEMPS D'EXÉCUTION DES PROGRAMMES DE TEST 84

6.2 COMPARAISON AVEC UN PROLOG CLASSIQUE 86

Chapitre 1

INTRODUCTION

Ce mémoire présente le système COP-Compilé. C'est un système permettant l'intégration de la programmation procédurale en C++ et de la programmation logique en Prolog. Le noyau de ce système est un compilateur de Prolog basé sur la machine abstraite de Warren (WAM). Cette machine qui définit une organisation de la mémoire, un ensemble de registres et un jeu d'instructions adaptés à Prolog, est actuellement la référence par excellence dans le domaine d'implantation de compilateurs Prolog.

1.1 Cadre du projet de recherche

Ce projet s'insère dans le cadre d'une série de travaux de recherche au sein du département de génie électrique et de génie informatique de l'université de Sherbrooke qui visent à rapprocher la programmation procédurale et la programmation logique.

Une première tentative de réalisation d'un compilateur intégrant C++ et Prolog a montré

la validité de l'idée de rapprochement des deux types de programmation. Ce compilateur, qui portait le nom de COP (C++ Ou Prolog) [1] a servi à prouver la faisabilité et l'utilité d'une intégration de C++ et de Prolog.

Il offrait, en outre, plusieurs avantages parmi lesquels:

- permettre au programmeur de choisir le style de programmation qu'il désire: procédural ou logique;
- permettre au programmeur de choisir l'interface graphique qu'il veut utiliser;
- donner au programmeur la possibilité d'utiliser des librairies existantes;
- le code C++ et le code Prolog pouvaient être développés indépendamment l'un de l'autre;
- le code final généré était portable sur n'importe quel type de machine.

Néanmoins, cette première version du compilateur COP, qui sera notée COPi dans la suite de ce mémoire afin de rappeler que le code Prolog y était interprété, souffrait de certaines limitations qui ne représentaient pas une préoccupation majeure lors de la conception du système. En effet, COPi ne visait pas la performance d'exécution mais plutôt la preuve de la faisabilité d'une intégration des styles de programmation procédural et logique.

Le projet de recherche présenté dans ce mémoire a pour objectif principal d'augmenter la performance de COPi en remédiant à ses limitations.

1.2 Présentation de la problématique

Malgré ses nombreux avantages et originalités, COP*i* présentait certaines limitations au niveau de la performance. En effet, le système COP*i* ne pouvait traiter que des programmes de petite taille. De plus, sa vitesse d'exécution ainsi que les programmes Prolog qu'il reconnaissait étaient limités.

Une analyse plus détaillée du système COP*i* a montré que ses limitations ont trois causes principales:

1. La représentation interne des termes et des règles du code Prolog est peu efficace. Elle ne permet de représenter qu'un nombre limité de règles et de termes. Aussi, la taille du code Prolog devrait-elle être petite.

2. L'ensemble des prédicats prédéfinis de Prolog qui peuvent être reconnus par COP*i* est minimal. Des programmes Prolog qui utilisent certains prédicats prédéfinis usuels ne peuvent être compilés par COP*i*.

3. Le module compilateur Prolog-COP de COP*i* qui sert à convertir le code Prolog en code C++ est basé sur un interpréteur Prolog qui reçoit du code Prolog et l'interprète séquentiellement. Ce mode d'implantation de Prolog, en plus d'être devenu obsolète, rend la vitesse d'exécution de COP*i* très lente.

1.3 Réalisations

Le système COP-Compilé, conçu dans le cadre de ce projet de recherche, permet d'améliorer l'efficacité du système COP*i*. Il remédie à ses principales limitations tout en respectant

sa philosophie globale et son schéma de compilation.

Le système COP-Compilé augmente la performance du système COPi en diminuant son temps d'exécution et en étendant l'ensemble des programmes Prolog qu'il peut reconnaître.

1.3.1 Diminution du temps d'exécution

L'amélioration du temps d'exécution dans le système COP-Compilé a été atteinte grâce à un bon choix de la représentation interne des données en mémoire et à la compilation du code Prolog à la place de son interprétation.

L'utilisation de la technique de représentation étiquetée pour coder les termes du code Prolog a permis de réduire le temps d'accès à ces données en mémoire, réduisant ainsi le temps global d'exécution.

La compilation du code Prolog en convertissant les prédicats Prolog en instructions de la WAM a permis de réduire le temps d'exécution puisque les instructions de la WAM ont été conçues de manière à accélérer la procédure d'unification de Prolog.

De plus, le module compilateur Prolog-COP a été redéfini en introduisant, notamment, un sous-module d'optimisation du code Prolog qui va localiser l'ensemble des prédicats qui doivent être compilés parmi ceux du code entier. Ainsi, seuls les prédicats pertinents seront compilés et non pas la totalité du code Prolog.

1.3.2 Extension des programmes Prolog reconnus

Les programmes Prolog compilés par le système COP-Compilé peuvent contenir des prédicats prédéfinis qui n'étaient pas reconnus par le système COP*i*. Leur taille peut dépasser la taille maximale des programmes Prolog reconnus par système COP*i*. Ils peuvent en fait être de n'importe quelle taille.

L'ensemble des prédicats prédéfinis de Prolog reconnus a été étendu dans le système COP-Compilé afin d'inclure les prédicats arithmétiques, logiques, métalogiques et de contrôle. Des programmes contenant des prédicats de ces types ne pouvaient être compilés par le système COP*i*.

L'utilisation de la technique de tableaux dynamiques pour le stockage des termes du code Prolog en mémoire a permis au système COP-Compilé de s'adapter à la taille du code Prolog.

1.4 Grandes lignes du travail

Ce mémoire décrit en détail les étapes de conception et de réalisation du système COP-Compilé.

Le chapitre 2 décrit brièvement le système COP*i* en présentant les langages intégrés par le système, les différents modules du système et son schéma de compilation.

Le chapitre 3 relate les principales limitations du système COP*i*. Après un rappel du cadre dans lequel une amélioration du système doit être faite, ce chapitre explicite les améliorations à apporter. Le choix de la machine abstraite de Warren (WAM) pour implanter le compilateur Prolog y est également justifié.

Le chapitre 4 décrit en détail la machine abstraite de Warren. Il présente l'organisation de la mémoire au sein de la machine, ses registres et son jeu d'instructions.

Le chapitre 5 définit la structure du système COP-Compilé et explicite les nouveaux modules introduits par le système ainsi que le nouveau schéma de compilation.

Le chapitre 6 présente les principales étapes de réalisation du système COP-Compilé. Il met en relief les phases de compilation du code prolog en instructions de la WAM et de l'exécution de ce code WAM par un émulateur. Il fournit, enfin, une validation du système COP-Compilé à travers les résultats des tests de sa performance.

Le chapitre 7 conclut ce mémoire en rappelant les originalités du système COP-Compilé et en proposant des extensions futures de ce travail.

Chapitre 2

LE SYSTÈME COP*i*

COP*i* est un compilateur permettant l'intégration de C++ et de Prolog en interprétant le code Prolog, d'où son nom (C++ Ou Prolog *i*nterprété). Un programmeur peut utiliser à sa guise C++ ou Prolog, c'est-à-dire travailler de manière procédurale ou de manière déclarative.

Un programme de COP*i* est formé d'un ensemble de définitions de variables, d'objets et de fonctions C++ ainsi que de prédicats Prolog. Comme dans C++, ces définitions peuvent apparaître dans n'importe quel ordre dans le programme.

Seul le code C++ peut appeler du code Prolog et ceci est réalisé grâce à des *bridge goals*. Un *bridge goal* est un but Prolog mais qui est écrit comme n'importe quel énoncé de C++, au milieu d'une fonction.

Après une présentation des deux langages intégrés par le système COP*i* ainsi que les caractéristiques du langage COP, ce chapitre, adapté en grande partie de [1], décrit les modules qui forment le système COP*i* ainsi que son schéma de compilation.

2.1 Langages intégrés par COP*i*

2.1.1 Le langage C++

Le langage C++ est un langage procédural, fortement typé et à objets. Il a été développé par Bjarne Stroustrup [2] et est dérivé du langage C [3].

La base de la programmation par objets est la notion de classe. Une classe définit un nouveau type de données et permet de lui associer des opérations. La création d'objets, aussi appelés instances, se fait en matérialisant une classe. Les opérations de la classe, appelées aussi méthodes, sont applicables à tout objet de cette classe.

L'héritage permet de définir une nouvelle classe à partir d'une autre. Une hiérarchie de classes est ainsi créée et on dit que la sous-classe est dérivée de sa super-classe. Le développement de la classe dérivée est réalisé par héritage et en ajoutant au besoin des attributs et/ou de nouvelles méthodes.

La programmation en C++ définit également les concepts de surdéfinition des opérateurs et de polymorphisme. Lorsqu'une fonction ou un opérateur est redéfini, on dit qu'il est surdéfini; le choix de la fonction ou de l'opérateur correspondant est fait, généralement, à la compilation grâce à sa classe d'appartenance et si besoin grâce à la liste des types des arguments. La liaison peut être résolue dynamiquement pendant l'exécution; l'idée est d'avoir un même nom associé, dans la hiérarchie des classes, à une multitude de méthodes et la méthode effectivement utilisée est déterminée à l'exécution selon le type effectif de l'objet et des arguments de la méthode. Ce mécanisme de liaison dynamique fait apparaître la hiérarchie des classes sous plusieurs formes, à partir d'un même nom de méthode; on parle alors de polymorphisme.

En ce qui concerne la mise en oeuvre du langage C++, certains compilateurs comportent un préprocesseur produisant du code C. Ce code est par la suite transformé en code machine par un compilateur C. Un compilateur C++ utilisant cette approche doit aussi prendre comme entrée du code C et le passer sans changement au compilateur C. En particulier, il est possible de définir des fonctions C et d'appeler ces fonctions à partir de méthodes écrites en C++.

2.1.2 Le langage Prolog

Le langage Prolog est un langage déclaratif inspiré de la logique et conçu par Alain Colmerauer [4]. Une bonne référence du langage Prolog est le livre de Clocksin et Mellish [5] qui a popularisé et standardisé la syntaxe d'Edinburgh.

Un programme Prolog est formé par un ensemble de clauses de Horn ou règles de la forme:

$$\text{T:- } Q_1, Q_2, ..., Q_n. \text{ où } n \geq 0$$

On dit que T est la tête de la règle, que Q_1, Q_2, ..., Q_n constituent le corps de celle-ci et que le symbole ":-" se lit *si*. La règle se lit donc comme suit: T est vrai si Q_1 et Q_2 et ... et Q_n sont vrais. La virgule entre les Q se lit donc comme un *et* logique. Lorsque le corps de la règle est vide, le cas où n=0, il s'agit d'une règle unité ou d'un fait.

La tête de la règle T est de la forme:

$$p(A_1, A_2, ..., A_m) \text{ où } m \geq 0$$

On dit alors que p est un prédicat ou foncteur et que les A_1, A_2, ..., A_m sont les arguments de la règle.

Les Q_1, Q_2, ..., Q_n et les A_1, A_2, ..., A_m sont des termes. Un terme est une constante, une variable ou un terme composé. Un terme composé comporte un foncteur suivi d'une séquence d'un ou plusieurs termes appelés arguments.

Un prédicat est défini comme un ensemble de règles définissant une relation. Dans ce cas, toutes ses règles doivent avoir le même foncteur et la même arité (nombre d'arguments).

L'exécution d'un programme Prolog se fait selon une stratégie particulière de résolution appelée *résolution de buts* [6]. Un programme Prolog est un ensemble de règles et de faits et l'exécution est déclenchée par une requête ou but. De manière simplifiée, pour une requête et un ensemble de règles, Prolog essaie d'attribuer des valeurs aux variables de la requête pour qu'elle soit vraie. L'exécution s'effectue en profondeur d'abord, les règles du programme sont essayées dans l'ordre où elles apparaissent dans le programme et les appels aux prédicats dans le corps de chaque règle sont effectués de gauche à droite.

Le langage Prolog offre quatre caractéristiques particulières:

1. Les variables logiques. Dans le langage Prolog une variable est le nom d'une inconnue. À l'exécution la valeur est inconnue, mais l'instanciation permet à une variable d'avoir une valeur. Toute variable est locale à la règle où elle figure. Les variables peuvent être passées comme arguments d'un terme ou d'un terme composé.

2. Le typage dynamique. Les variables peuvent contenir des valeurs de n'importe quel type:

   - un atome (constante);
   - un entier;
   - une liste;

- un terme composé.

3. L'unification. L'unification est une opération spéciale cherchant à trouver l'instance commune la plus générale entre deux termes. Elle est utilisée pour construire et avoir accès aux termes d'un terme composé. Elle est aussi utilisée pour lier une variable à une autre. Lorsqu'une des variables liées est instanciée, toutes les variables qui lui sont liées sont aussi instanciées à cette valeur.

4. Le retour en arrière. Pendant l'exécution, Prolog essaie de satisfaire les clauses dans l'ordre d'apparition dans le programme, c'est-à-dire trouver des valeurs aux variables afin que les clauses soient vraies. Si un prédicat possédant plus d'une clause est appelé, Prolog se souvient qu'il y a un point de choix. Si Prolog ne peut pas rendre la clause choisie vraie (échec à l'unification), un retour en arrière est fait sur le point de choix le plus récent afin de poursuivre l'exécution avec la prochaine clause du prédicat. De cette manière Prolog peut donner toutes les solutions possibles pour un but à résoudre.

2.2 Le langage COP

Le langage COP n'est rien d'autre qu'une réunion du langage C++ et du langage Prolog. Un programme écrit en COP est formé de définitions de variables, d'objets et de fonctions de C++ ainsi que de prédicats de Prolog.

Le langage C++ est , cependant, le langage maître puisque seul le code C++ d'un programme COP peut appeler le code Prolog et non pas l'inverse.

Le code C++ peut donc contenir des appels de buts Prolog, appelés *bridge-goals*. Du côté C++, les *bridge-goals* ressemblent à des appels de fonctions alors que du côté Prolog c'est

des buts Prolog.

Un fichier intégrant du code C++ avec des *bridge-goals* et du code Prolog est appelé *fichier source COP*. La section A.1 des annexes présente un exemple de *fichier source COP*. L'instruction `if(power(X,Y,Z))` contient un bridge-goal.

Une *fichier source COP* contient du code C++ et du code Prolog. Les types de code source soumis au compilateur COP sont donc de trois catégories:

1. Type bridge: du code source C++ contenant des *bridge-goals* (Voir la section A.2 des annexes).
2. Type Prolog: du code source Prolog (Voir la section A.3 des annexes).
3. Type C++: du code source C++ (Voir la section A.4 des annexes).

Le code du *fichier source COP* est séparé en deux, d'un côté tout le code C++ contenant des *bridge-goals* et de l'autre tout le code Prolog. La partie du compilateur COP réalisant cette tâche de séparation du code est appelée *séparateur COP*. Le code Prolog, issu d'un fichier de type Prolog ou généré par le *séparateur COP*, doit être converti en code compilable par un compilateur C++, la partie du compilateur COP réalisant cette transformation est appelée *compilateur Prolog-COP*. Un compilateur C++ génère le programme exécutable par la compilation de tous les fichiers de type C++, qu'ils soient du code source du programmeur ou générés par le *compilateur Prolog-COP*.

Les *bridge-goals* dans le code C++ sont convertis par le *convertisseur COP* en des appels de fonctions C++. Ces fonctions sont appelées des *bridge-functions*. Chaque *bridge-goal* a une *bridge-function* qui lui est exclusive. Le but premier des *bridge-functions* est de réaliser le pont entre le code C++ et le code Prolog. La section A.5 présente la *bridge-function* `power_03_00`

correspondant au *bridge-goal* `power(X, Y, Z)`.

Le langage Prolog peut générer plusieurs solutions à un appel d'un but en relançant ce même but plusieurs fois successives. Dans COP, ce mécanisme est réalisé en mettant un *bridge-goal* dans un énoncé de boucle (`while`, `do` ou `for`). Chaque itération engendre une solution différente, si elle existe, du but Prolog correspondant.

Les variables en paramètre d'un *bridge-goal* permettent l'échange des données entre le code C++ et le code Prolog. Ces variables sont appelées des *bridge-variables* afin de les distinguer des variables C++ classiques dans le code C++ et des variables Prolog dans le code Prolog. Le programmeur dispose d'une interface permettant de manipuler les *bridge-variables*. C'est la classe `TBridgeVariable`. Les paramètres `X`, `Y` et `Z` du *bridge-goal* `power(X, Y, Z)` sont des *bridge-variables*.

2.3 Structure de COP*i*

Le compilateur COP*i* est formé de trois principaux modules:

1. Le séparateur COP qui permet de séparer les fichiers sources de type COP (fichiers intégrant du code C++ et du code Prolog) en deux ensembles de fichiers: un ensemble de fichiers de type *bridge* et un ensemble de fichiers de type Prolog.

2. Le convertisseur COP qui a pour tâche principale de convertir les fichiers de type *bridge* (fichiers contenant du code C++ faisant appel au code Prolog) en fichiers de type C++.

3. Le compilateur Prolog-COP qui permet de convertir tous les fichiers de type Prolog générés par le séparateur COP en *byte-code* qui sert d'entrée à un interpréteur Prolog.

2.3.1 Le séparateur COP

Le séparateur COP réalise deux tâches:

1. Séparer les fichiers de type COP en deux ensembles de fichiers: un ensemble de fichiers de type *bridge* et un ensemble de fichiers de type Prolog.
2. Bâtir la table de symboles qui contient les foncteurs et les arités de tous les prédicats Prolog reconnus.

Séparation des fichiers de type COP

La séparation des deux types de fichiers se fait comme suit:

- si le séparateur COP reconnaît un prédicat Prolog, ce prédicat est copié dans un fichier de type Prolog;
- sinon, le séparateur considère que la partie de code rencontrée est un énoncé C++ et le prochain bloc de code C++ est copié dans le fichier de type C++;
- cette reconniassance est répétée jusqu'à ce que la fin du fichier COP soit atteinte.

Pour reconnaître un prédicat Prolog dans un fichier de type COP, le séparateur COP se base sur la grammaire de Prolog de la figure 2.1. Dans cette figure et les deux suivantes, la grammaire est représentée selon la notation de *grammaire non contextuelle* et les unités lexicales respectent la notation des *expressions régulières* [7].

prog	→	clause prog	struct	→	IDENT (s_args)
prog	→	clause	struct	→	VARIABLE (s_args)
prog	→	error prog	struct	→	plist
clause	→	rule	plist	→	[ ]
clause	→	fact	plist	→	[items]
rule	→	litt :- right .	items	→	term , items
fact	→	litt .	items	→	term — term
			items	→	term
right	→	litt_r , right			
right	→	litt_r	s_args	→	term , s_args
			s_args	→	term
litt_r	→	IDENT (args)			
litt_r	→	IDENT	args	→	term , args
litt_r	→	VARIABLE (args)	args	→	term
litt_r	→	VARIABLE			
litt_r	→	!	simple	→	const
			simple	→	VARIABLE
litt	→	IDENT (args)			
litt	→	IDENT	const	→	IDENT
			const	→	INTEGER
term	→	struct	const	→	STRING
term	→	simple			

IDENT	[a-z][a-zA-Z-0-9_]*	VARIABLE	[A-Z][a-zA-Z-0-9_]*
INTEGER	[0-9]+	STRING	"[^"]*"

Figure 2.1 - *La grammaire Prolog du séparateur COP*

Côté C++, le séparateur COP a seulement besoin de connaître la syntaxe terminant un énoncé C++ et la syntaxe d'un bloc C++. La grammaire du C++ reconnu est celle de la figure 2.2

```
code_c  ⟶  bloc
code_c  ⟶  stmt
code_c  ⟶  T bloc

bloc    ⟶  { TB }
bloc    ⟶  { TB bloc }
bloc    ⟶  { bloc TB }
bloc    ⟶  { TB bloc TB }

stmt    ⟶  T ;


T       [^{};]+
TB      [^{}]+
```

Figure 2.2 - *La grammaire C++ du séparateur COP*

Construction de la table des symboles

La construction de la table des symboles se fait au fur et à mesure de la séparation des fichiers de type COP. Si une règle ou un fait est reconnu, son foncteur et son arité sont introduits dans la table des symboles.

2.3.2 Le convertisseur COP

Le convertisseur COP permet de générer le code liant le code C++ et le code Prolog. Il effectue les tâches suivantes:

1. Convertir les fichiers de type *bridge* en fichiers de type C++.
2. Générer le fichier de prototypes.
3. Générer le fichier de fonctions *bridge*.

Conversion de fichiers de type *bridge* en fichiers de type C++

La conversion de fichiers de type *bridge* en fichiers de type C++ est faite en localisant les *bridge-goals*, c'est à dire les buts Prolog mais écrits au milieu de fonctions de C++, dans les fichiers de type *bridge* à l'aide de la table des symboles. Les *bridge-goals* sont convertis en des appels de fonctions *bridge* uniques.

Les *bridge-goals* sont localisés dans les fichiers de type *bridge* à l'aide de la table des symboles qui permet de connaître le foncteur et l'arité des appels possibles aux prédicats dans le code Prolog. La grammaire qui permet de reconnaître un appel de *bridge-goal* dans le code C++ est celle de la figure 2.3.

Une fois qu'un appel est localisé, le foncteur et son arité sont comparés à ceux dans la table des symboles. S'il y a coïncidence le *bridge-goal* est substitué par un appel de fonction C++. C'est la fonction *bridge* associée à ce *bridge-goal*.

Pour associer une fonction *bridge* unique à chaque *bridge-goal*, le convertisseur COP associe un compteur, initialisé à zéro, à chaque couple prédicat/arité dans la table des symboles.

```
prog    ⟶  prog p_any
prog    ⟶  prog IDENT
prog    ⟶  prog block
prog    ⟶  /* rien */

p_any   ⟶  OTHER
p_any   ⟶  (
p_any   ⟶  )
p_any   ⟶  ,

block   ⟶  { body }

body    ⟶  body call
body    ⟶  body b_any
body    ⟶  body block
body    ⟶  /* rien */

b_any   ⟶  OTHER
b_any   ⟶  ,

call    ⟶  IDENT ( parms )
call    ⟶  IDENT

parms   ⟶  sparm , parms
parms   ⟶  sparm

sparm   ⟶  OTHER sparm
sparm   ⟶  call sparm
sparm   ⟶  /* rien */

IDENT   [a-z][a-zA-Z-0-9_]*
OTHER   [^,(){}]
```

Figure 2.3 - *La grammaire C++ du convertisseur COP*

Chaque appel de *bridge-goal* qui coïncide avec un couple prédicat/arité augmente de un la valeur du compteur associé à ce couple. Le nom de la fonction *bridge* associée au *bridge-goal* à convertir est composé de l'union du nom du prédicat, de son arité et de la valeur courante du compteur associé à ce couple prédicat/arité.

Par exemple, pour un prédicat de nom `pere`, ayant une arité de deux et une valeur de

compteur de quatre, la fonction *bridge* aura le nom `pere_02_04` et le compteur associé au couple sera incrémenté à cinq.

Cette méthode de compteur assure que deux *bridge-goals* différents faisant appel au même prédicat Prolog auront des noms différents de fonctions *bridge*. L'arité dans la composition du nom est nécessaire puisqu'elle permet de faire la distinction entre deux prédicats de même nom ayant des arités différentes mais une même valeur de compteur.

Génération du fichier de prototypes

Afin de générer le fichier de prototypes, un fichier est ouvert et pour chaque prédicat/arité dans la table des symboles ayant un ou plusieurs appels de fonctions *bridge* générés, les prototypes correspondants sont créés dans le fichier.

Le nombre de prototypes pour chaque couple prédicat/arité dans la table des symboles est indiqué par la valeur finale du compteur.

Par exemple, pour un prédicat de nom `pere`, d'arité deux et d'une valeur de compteur trois, les prototypes pour les fonctions *bridge* suivantes sont produits: `pere_02_00`, `pere_02_01` et `pere_02_02`.

Génération du fichier des fonctions *bridge*

Les noms des fonctions *bridge* à générer sont déterminés de la même façon que pour les prototypes. Le corps de ces fonctions est généré à partir d'un même squelette en C++ (Voir la figure 6.1) puisque toutes les fonctions *bridge* sont identiques sauf sur deux points: le nom de la fonction *bridge* et ses paramètres.

2.3.3 Le compilateur Prolog-COP

Ce compilateur est responsable de la conversion de tous les fichiers Prolog générés par le séparateur COP en *byte-code*. Ce *byte-code* est, en fait, un tableau d'entiers qui sont la représentation interne des termes et des règles du programme Prolog. Ce tableau est écrit en C++ et sera parcouru lors de l'exécution par l'interpréteur Prolog. Cet interpréteur est un programme écrit en C++ qui est inspiré d'un mini interpréteur de `PrologII` [8].

Le compilateur Prolog-COP fonctionne en deux phases: une phase de génération et une phase d'interprétation.

Phase de génération

Le compilateur Prolog-COP lit le fichier source contenant les prédicats prédéfinis et les prédicats conçus par le programmeur. Les termes de ces prédicats ainsi que l'enchaînement des règles est codifié sous forme d'entiers. Un tableau rassemblant cette représentation interne est généré. C'est le *byte-code*.

Le *byte-code* ainsi que la table des symboles Prolog sont sauvegardés dans un fichier qui va être joint aux autres parties constituant le programme COP complet.

Phase d'interprétion

C'est la phase d'exécution du code Prolog transformé en *byte-code* par l'interpréteur. L'interface entre l'interpréteur et le programme COP est réalisée grâce aux méthodes d'une

classe C++ spécifique, `TProlog`:

1. La méthode `setupGoal` permet de fournir à l'interpréteur le but à résoudre ainsi que les paramètres de celui-ci.
2. La méthode `run` exécute le but spécifié par `setupGoal`. Elle trouve uniquement la première solution et conserve ses points de choix.
3. La méthode `getVariables` transforme les résultats obtenus par la machine Prolog dans la représentation des variables *bridge*.
4. La méthode `reset` détruit tous les points de choix et s'assure que la machine Prolog est prête à traiter un autre but.

2.4 Schéma de compilation de COP*i*

La compilation d'un programme de COP*i* se fait en huit étapes:

1. Séparation du programme de COP*i* en fichiers Prolog et fichiers *bridge*.
2. Génération de la table des symboles contenant les prédicats Prolog et leur arité.
3. Création des fichiers C++, avec les *bridge-goals* transformés en des appels de fonctions *bridge*.
4. Création d'un fichier entête *bridge* qui contient les prototypes des fonctions *bridge*.
5. Création d'un fichier de fonctions *bridge* qui contient le code des fonctions *bridge*.

6. Conversion des fichiers Prolog en *byte-code* qui est mis en relation avec le code C++ de l'interpréteur Prolog.

7. Compilation de tous les fichiers: les fichiers C++ et les fichiers de fonctions *bridge*.

8. Édition de liens des fichiers objets et des librairies COP pour générer le programme exécutable.

La figure 2.4 montre la structure générale des fichiers et le schéma complet de compilation COP. Les rectangles identifient des fichiers et les ovales des applications.

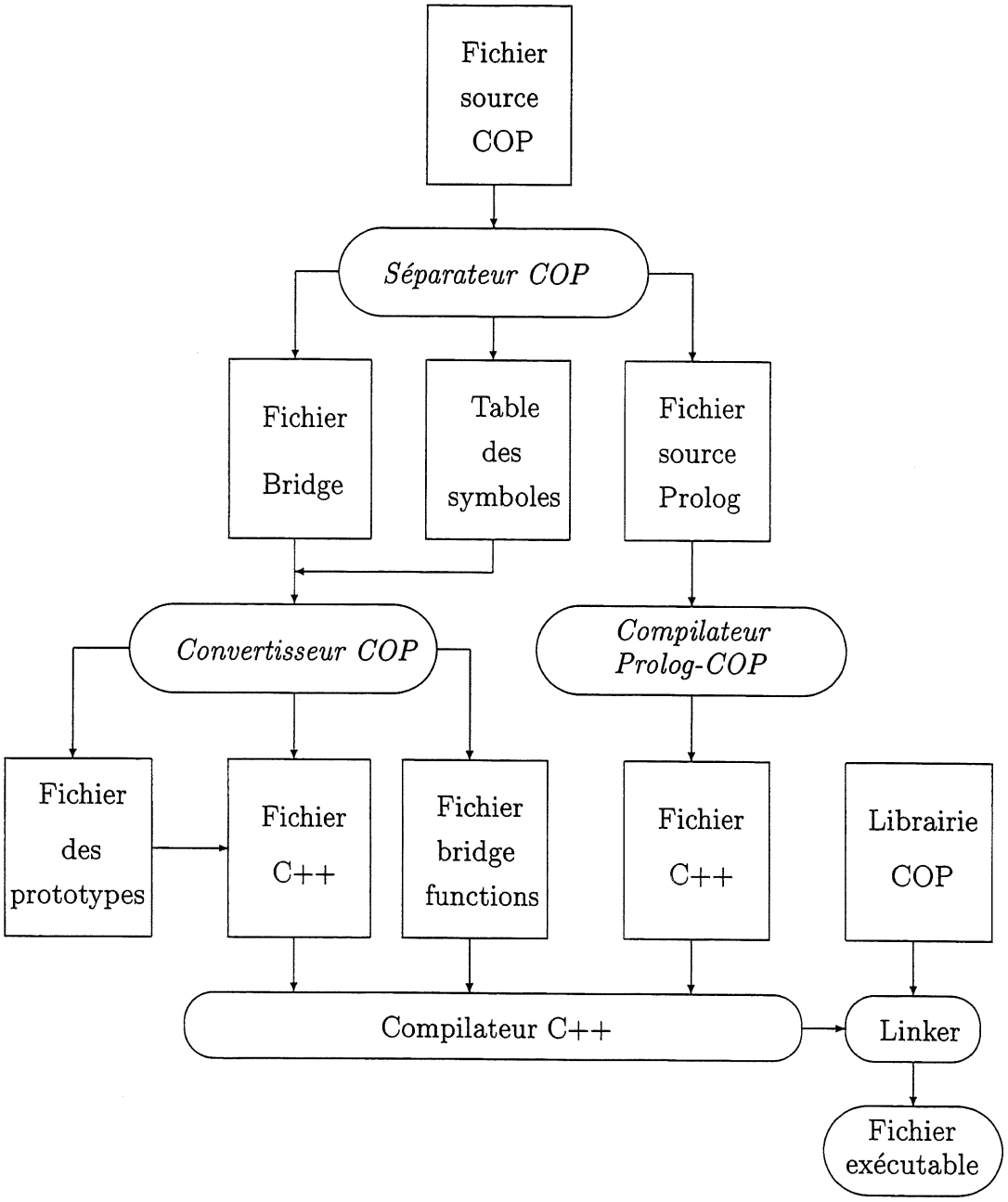

Figure 2.4 - *Flux des données et schéma de compilation de COPi*

Chapitre 3

L'AMÉLIORATION DE COPi

Malgré les nombreuses qualités de COPi et notamment la preuve de la faisabilité et de l'utilité d'une intégration de la programmation procédurale et de la programmation logique, le compilateur présente encore quelques limitations au niveau de la représentation interne des données, des prédicats prédéfinis et de l'exécution du code Prolog.

Le but de ce chapitre est de présenter ces limitations du système COPi ainsi que les améliorations qui doivent être apportées afin d'y remédier tout en rappelant le cadre dans lequel ces améliorations devront se faire, à savoir les objectifs de conception et de réalisation fixés au début du projet COP.

3.1 Limitations de COPi

3.1.1 Représentation interne des données

Codage des termes Prolog

L'interpréteur Prolog de COPi est inspiré d'un mini interpréteur de PrologII [8]. Ce dernier ayant été conçu pour fonctionner sur un micro-ordinateur à mémoire vive de 32Ko, il ne réservait qu'un espace de 8Ko pour coder les règles et les termes Prolog. Cet espace est subdivisé de manière statique entre les différents types de termes.

Les termes sont codés sous forme d'entiers de 16 bits dont la valeur est comprise entre -100 et 19999.

Le type du terme est déterminé par l'intervalle d'entiers auquel appartient la représentation du terme:

- Variable: Entre -100 et -1.
- Entier: Entre 0 et 9999.
- Caractère: Entre 10000 et 10255.
- Identificateur: Entre 11000 et 11999.
- Séquence: Entre 12000 et 15999.
- Fonction: Entre 16000 et 19999.

La valeur du terme codé est déterminée en soustrayant de la représentation du terme la borne inférieure de l'intervalle correspondant au type du terme.

Ainsi, si un terme est représenté sous forme de l'entier 10032, alors il s'agit d'un caractère puisque 10032 appartient à l'intervalle [10000, 10255] et la valeur de ce caractère (son code ASCII) est 32, i.e. 10032 - 10000.

Ce choix de codage des termes est peu efficace puisque le nombre d'entités pouvant être codées est limité. Par exemple, le nombre d'identificateurs dans un programme Prolog ne doit pas dépasser 1000, de même le nombre de variables ne peut dépasser 100. De plus, la détermination du type du terme est très coûteuse en temps puisqu'elle fait intervenir l'opération de soustraction.

Utilisation de tableaux statiques

Le programme Prolog traité par COPi est stocké en mémoire, après codage de ses règles et termes, sous forme de tableau statique. Un tableau statique est caractérisé par sa taille qui est spécifiée lors de sa déclaration avant le premier accès au tableau. Cette taille ne peut être modifiée en cours d'exécution.

Du fait que les programmes Prolog traités par COPi peuvent être de taille quelconque, il y a réservation inutile de la mémoire si le programme est beaucoup plus petit que la taille allouée au tableau qui va recevoir les entités du programme, ou débordement si le programme dépasse la taille allouée.

3.1.2 Ensemble des prédicats prédéfinis

L'ensemble de prédicats prédéfinis de COPi est très restreint. Beaucoup des prédicats prédéfinis utilisés par les programmes Prolog usuels n'ont pas besoin d'être réalisés puisqu'ils traitent des entrées/sorties qui, dans COPi, se font dans le code C++.

Cependant, l'absence dans COP*i* de certains prédicats prédéfinis considérés standards tels que les prédicats arithmétiques, logiques, métalogiques et de contrôle représente un handicap majeur du système.

3.1.3 Exécution du code Prolog

La limitation majeure de COP*i* est l'utilisation d'un interpréteur pour exécuter le code Prolog. En effet, le code généré par le compilateur Prolog-COP est du *byte-code* qui doit être interprété.

Ce choix se justifie par le fait que COP*i* ne visait pas la performance d'exécution du code Prolog en premier lieu.

Néanmoins, la vitesse d'exécution du compilateur COP*i* pourra être améliorée si le code Prolog est compilé à la place d'être interprété.

3.2 Améliorations à apporter

3.2.1 Objectifs de conception et de réalisation

L'amélioration du système COP*i* doit s'effectuer dans le cadre des considérations fixées au début du projet COP.

Ainsi, la version améliorée de COP doit-elle respecter, d'abord, les objectifs de conception suivants:

1. Pouvoir intégrer dans un seul programme source du code C++ et du code Prolog.

2. Respecter la syntaxe et la sémantique des deux langages.
3. Pouvoir développer de façon indépendante les parties C++ et Prolog d'un programme.
4. Pouvoir utiliser les bibliothèques existantes de C++ et de Prolog.

Ensuite, les objectifs de réalisation suivants:

1. La portabilité: Le compilateur COP lui-même et le code généré par le compilateur doivent être portables.
2. La modularité: Le code source compilé par COP doit pouvoir être modulaire afin de permettre de développer le code en C++ et le code en Prolog de façon indépendante. Elle permet aussi au programmeur d'utiliser les bibliothèque existantes de C++ et de Prolog.

De plus, la version améliorée de COP doit:

1. Garder le même schéma global de compilation de COP.
2. Garder la même structure des fichiers de COP sauf là où une amélioration s'impose.
3. Respecter les grammaires de Prolog et de C++ reconnues par le compilateur COP.

3.2.2 Représentation interne des données

Représentation étiquetée

Pour coder les termes du programme Prolog, la représentation étiquetée [9] sera utilisée. Cette technique permet de représenter les termes sous forme de mots de 32 bits. Ces 32 bits

sont subdivisés en deux champs:

1. Un champ étiquette sur les 3 bits du poids le plus fort.
2. Un champ valeur sur les 29 bits restants.

Le champ étiquette spécifie le type du terme. Chaque combinaison de bits correspond à un type particulier de termes. La détermination du type du terme se fait alors par une simple opération ET logique entre le mot et un masque de 32 bits où les trois premiers bits sont à 1 et les autres à 0.

Le champ valeur spécifie la valeur du terme. La détermination de la valeur du terme se fait alors par une simple opération ET logique entre le mot et un masque de 32 bits où les trois premiers bits sont à 0 et les autres à 1.

Une discussion détaillée de la représentation de chaque terme est donnée à la page 38.

Tableaux dynamiques

Pour remédier aux problèmes de débordement et de gaspillage de mémoire liés aux tableaux statiques utilisés pour stocker les termes du programme Prolog, la méthode d'allocation dynamique de tableaux [10] sera adoptée.

Cette technique permet d'allouer de la mémoire à des tableaux de taille arbitraire de sorte que cette taille augmente lors de l'exécution du programme.

Partant d'une taille minimale pour le tableau Prolog, si la limite du tableau est atteinte, sa taille est doublée par recopie dans un tableau plus grand.

3.2.3 Ensemble des prédicats prédéfinis

Pour étendre l'ensemble de programmes Prolog qui peuvent être compilés par COP, il faudrait doter ce dernier d'un mécanisme de reconnaissance des prédicats prédéfinis les plus utilisés.

Des prédicats de type arithmétique, logique, métalogique ou de contrôle apparaissent dans la plupart des programmes Prolog et COP devrait être capable de les reconnaître.

Partant de l'ensemble des prédicats prédéfinis de `C-Prolog` [11], un sous-ensemble de prédicats qui sont devenus des standards du fait de leur utilisation par la majorité des programmes Prolog a été retenu. Ce premier choix pourra être facilement étendu par la suite, puisqu'il suffit d'ajouter une entrée à la table des prédicats prédéfinis et écrire le code correspondant à ce prédicat.

Prédicats arithmétiques

Ces prédicats prennent comme argument des expressions arithmétiques et les évaluent. Au moment de l'évaluation, chaque variable dans une expression arithmétique doit être liée à un nombre ou à une expression arithmétique.

Les prédicats arithmétiques reconnus sont les suivants, où `X` et `Y` sont des expressions arithmétiques et `Z` un terme quelconque:

- `Z is X`: L'expression arithmétique `X` est évaluée et le résultat est unifié avec `Z`.
- `X+Y`: Addition de `X` et `Y`.
- `X-Y`: Soustraction de `Y` de `X`.

- `X*Y`: Multiplication de `X` et `Y`.
- `X/Y`: Division de `X` par `Y`.

Prédicats logiques

Ces prédicats évaluent leurs arguments et les comparent. Au moment de l'évaluation, chaque variable doit être liée à un nombre ou à une expression arithmétique.

Les prédicats logiques reconnus sont les suivants, où `X` et `Y` sont des expressions arithmétiques:

- `X==Y`: Succès si les valeurs de `X` et `Y` sont égales.
- `X\ =Y`: Succès si les valeurs de `X` et `Y` ne sont pas égales.
- `X<Y`: Succès si la valeur de `X` est inférieure à celle de `Y`.
- `X>Y`: Succès si la valeur de `X` est supérieure à celle de `Y`.
- `X=<Y`: Succès si la valeur de `X` est inférieure ou égale à celle de `Y`.
- `X>=Y`: Succès si la valeur de `X` est supérieure ou égale à celle de `Y`.

Prédicats métalogiques

Ces prédicats permettent de tester la nature de leur argument.

- `var(X)`: Teste si `X` est actuellement instanciée à une variable.

- `nonvar(X)`: Teste si `X` est actuellement instanciée à un terme non variable.
- `atom(X)`: Teste si `X` est actuellement instanciée à un atome, c.à.d. un terme non variable d'arité `0` qui n'est pas un nombre.
- `integer(X)`: Teste si `X` est actuellement instanciée à un entier.
- `atomic(X)`: Teste si `X` est actuellement instanciée à un atome ou à un nombre.

Prédicats de contrôle

Ces prédicats permettent de modifier le flux d'exécution d'un programme Prolog.

- `true`: Il réussit à chaque appel.
- `fail`: Il échoue à chaque appel.

3.2.4 Compilation de Prolog

L'amélioration principale qui doit être apportée au système COP*i* est la compilation du code Prolog à la place de son interperétation.

Afin de respecter l'objectif de portabilité cité plus haut, le code Prolog ne devra pas être compilé en assembleur ou en langage machine. Il doit être plutôt traduit en langage C.

Il existe, actuellement, deux approches pour compiler Prolog en C:

1. L'approche basée sur la continuation [12, 13];
2. L'approche basée sur la machine abstraite de Warren (WAM)[14].

L'approche basée sur la continuation consiste à compiler chaque prédicat de Prolog en une fonction C qui possède un argument additionnel: une fonction de continuation. Si la fonction échoue, elle effectue un retour normal. Par contre, si elle réussit, elle exécute la fonction de continuation [13]. Cette approche est très complexe et consomme beaucoup de mémoire à cause des appels récursifs des fonctions C [15].

L'approche basée sur la WAM consiste à compiler les prédicats et les requêtes Prolog en un ensemble d'instructions de la WAM. Cette machine abstraite est alors réalisée en C.

L'approche basée sur la WAM sera adoptée pour compiler le code Prolog. Ce choix est justifié essentiellement par les faits suivants:

1. Tous les travaux en cours dans le domaine de compilation de Prolog utilisent la WAM comme cible [16].

2. On dispose d'une riche bibliographie traitant de la compilation basée sur la WAM.

3. Les meilleurs compilateurs de Prolog qui existent actuellement sont basés sur la WAM ou sur des architectures qui en dérivent [16].

4. Pouvoir profiter des résultats des recherches effectuées séparément sur les techniques de la WAM telles que les méthodes d'optimisation de code [17] ou les algorithmes de gestion de mémoire [18].

Chapitre 4

LA MACHINE ABSTRAITE DE WARREN

En 1983, David H. D. Warren a conçu une machine virtuelle pour l'exécution de Prolog qui définit une organisation de la mémoire, un ensemble de registres et un jeu d'instructions adaptés à Prolog. Cette machine, connue sous le nom de *machine abstraite de Warren* (WAM), est devenue une solide référence en matière d'implantation de compilateurs Prolog.

La WAM a aidé de nombreux chercheurs à acquérir une meilleure compréhension de l'exécution de Prolog et à développer des systèmes de programmation logique efficaces. De plus, la WAM s'est avérée suffisamment flexible pour servir de base à diverses extensions de la programmation logique telles que les contraintes, la concurrence, le parallélisme, l'ordre supérieur, etc.

Le rapport original de Warren [14] décrivait la WAM dans un style qui rendait la compréhension difficile pour un lecteur moyen même s'il a des connaissances préalables sur les

opérations de Prolog. En 1991, H. Aït Kaci a remédié à cette lacune en publiant un ouvrage qui a rendu les concepts de la WAM plus abordables [19]. L'un des aspects positifs de ce livre est qu'il présente l'implantation de chaque opération de la WAM sous forme de procédure de type Pascal.

La compilation d'un prédicat, dans la WAM, consiste à transformer celui-ci en une suite d'instructions réalisant l'indexation des clauses qui le forment, ainsi que la gestion du retour-arrière qui lui est associé, avec notamment la création et la suppression des points de choix.

La compilation d'une clause consiste à générer les instructions réalisant l'unification de sa tête, ainsi que l'appel des différents buts que comporte sa partie droite [20].

Les sections suivantes, adaptées de [21], décrivent la manière avec laquelle la mémoire est organisée dans la WAM, le rôle des différents registres de la WAM ainsi que son jeu d'instructions.

4.1 Organisation de la mémoire dans la WAM

L'espace mémoire de la WAM est subdivisé en quatre zones:

1. Une zone de code où les instructions du code WAM sont stockées.
2. Une pile locale pour sauvegarder les environnements et les points de choix.
3. Une pile de restauration (*trail*) utilisée pour défaire les liaisons des variables lors du retour-arrière.
4. Une pile de recopie (*heap*) qui contient les termes créés au cours de l'exécution.

Pile locale (ou de contrôle)

Cette pile permet d'automatiser le parcours de l'arbre de recherche Prolog. Elle contient les triplets correspondants aux états de recherche. Lorsqu'un échec survient (aucune tête de clause ne s'unifie avec le littéral courant) la procédure de retour-arrière dépile un certain nombre d'éléments jusqu'à l'obtention d'un triplet pour lequel il existe une alternative. En vue de diminuer cette recherche et accéder en temps constant à l'élément susceptible de fournir une nouvelle solution, il est possible de déterminer, lors de l'avancée, si le noeud courant donnera lieu à un retour-arrière et, dans ce cas, d'empiler un élément particulier appelé *point de choix.*

Ainsi, la pile locale gère le contrôle de Prolog que l'on peut séparer en deux phases:

1. Avancée: Appels imbriqués de procédure. On utilise un environnement (ou bloc d'activation) où sont stockées les variables (locales) de la clause et les informations de contrôle utiles à la sortie du bloc courant (i.e. de la clause). Ces environnements sont similaires à ceux nécessaires lors de l'implantation d'un langage qui gère des variables locales, tel que C. Ainsi, une clause peut être comparée à une fonction, et ses variables (nouvelles instances à chaque utilisation de la clause) à des variables locales.

2. Retour-arrière (*backtracking*): Il consiste à reprendre le calcul à la dernière alternative non encore exploitée. Ainsi, un point de choix stocke les informations nécessaires à la reprise de calcul. On y trouve donc la sauvegarde de la plupart des registres de base ainsi que l'adresse de la nouvelle clause à essayer.

Ces deux types de blocs de contrôle sont entrelacés et donc chaque bloc contient un pointeur vers le bloc précédent de même type pour permettre le dépilement.

Deux registres de base `E` et `B` pointent respectivement sur le dernier environnement et sur le dernier point de choix. Ainsi, la pile locale contient deux piles entrelacées avec deux sommets de pile et la possibilité, à partir de chaque bloc, d'accéder au bloc précédent de même type. Lorsqu'un nouveau bloc doit être alloué, le calcul de son adresse revient au calcul du maximum entre les deux sommets de la pile.

Cette façon de faire simplifie la synchronisation des points de choix et des environnements. Ainsi, lorsqu'un environnement est libéré alors qu'un point de choix a été créé au-dessus de lui (par un de ses fils), l'espace de cet environnement n'est pas réutilisé puisqu'il sera nécessaire lors du retour-arrière.

Pile de restauration (*trail*)

Chaque état de l'arbre de recherche Prolog donne lieu à un environnement où sont stockés les variables locales de la clause ainsi qu'un vecteur de substitution qui associe de manière bi-univoque un élément à une variable de la clause. Néanmoins, il arrive que certaines substitutions affectent des variables n'apparaissant pas dans la clause courante. Ces liaisons affectent alors des éléments du vecteur de substitution d'un environnement antérieur à l'environnement courant. Lors du retour-arrière, si cet environnement est également antérieur à celui du dernier point de choix, il faudra défaire ces liaisons pour pouvoir relancer, *avec les mêmes données*, le calcul sur une autre alternative.

La solution pour remettre à l'état initial (non lié) de telles variables est d'utiliser une autre pile, appelée pile de restauration (ou *trail*). Lorsqu'une variable antérieure au dernier point de choix doit être liée, sa référence est empilée. Ainsi, à condition de mémoriser le sommet de cette pile dans les points de choix, il suffit lors du retour-arrière de remettre à l'état libre toutes les variables référencées dans la trail entre le sommet actuel et celui enregistré dans le

dernier point de choix.

Pile globale (*heap*)

Cette pile permet de stocker les termes structurés. Elle a une structure de pile dans la mesure où les nouveaux termes sont empilés sur son sommet et qu'elle est dépilée lors du retour-arrière. Elle a, également, une structure de tas du fait que des liaisons existent aussi bien du bas de la pile vers le haut que du haut vers le bas.

Les termes à l'interieur de la pile sont codés par des mots étiquetés de la forme <*identificateur, valeur*>. La figure 4.1 schématise la représentation interne de chaque terme, à savoir:

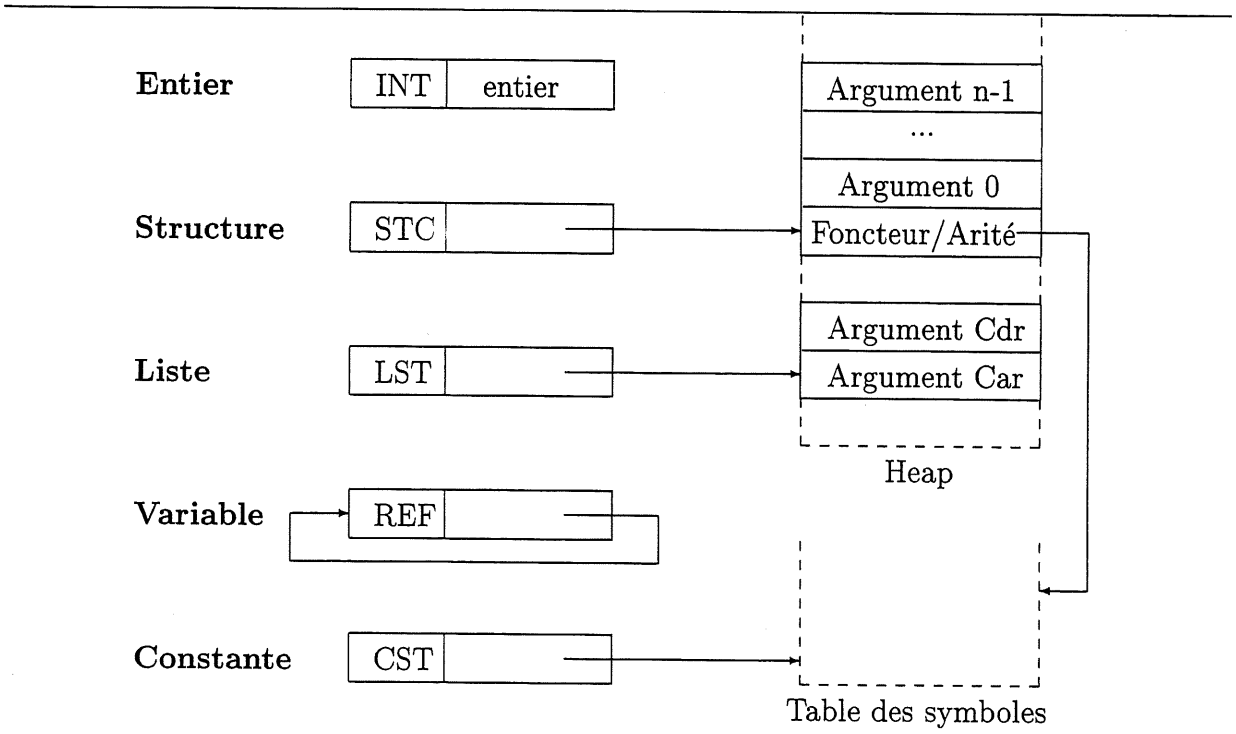

Figure 4.1 - *Représentation interne des termes dans le heap*

variable : La partie *valeur* est une référence vers le terme auquel est liée la variable. Une variable libre est simplement représentée comme une auto-référence. Il faut noter que l'affectation d'un tel mot à un registre crée un lien du registre vers la variable. On appelle *déréférenciation* l'opération consistant à suivre un chaînage de variables liées jusqu'à ce qu'une variable libre ou un terme différent d'une variable soit rencontré.

constante : La partie *valeur* pointe dans une table de *hash-code* stockant toutes les constantes, ramenant ainsi la comparaison de deux constantes à la comparaison de deux entiers.

entier : La partie *valeur* code l'entier.

liste non vide : La partie *valeur* pointe vers une cellule du *heap* contenant le *Car* (la tête de la liste), la suivante contenant le *Cdr* (la queue de la liste).

structure : La partie *valeur* pointe vers une cellule du *heap* contenant le foncteur (une adresse dans la table de *hash-code* des constantes) et l'arité (nombre n de sous-termes). Consécutivement à ce mot viennent les n mots associés aux sous-termes.

Une des principales caractéristiques de la WAM est due au choix de la représentation des termes composés (i.e. les listes et les structures) par *recopie de structure*. En effet, un terme est traité différemment suivant qu'il est *décomposé* (accès à une instance déjà existante) ou *construit* (création d'une nouvelle instance à partir d'un modèle). Pour une liaison en décomposition, la variable sera simplement liée à l'instance déjà existante alors qu'en construction, la variable est liée à une nouvelle copie du modèle.

La WAM définit ainsi deux modes lors de l'unification de termes structurés:

READ: Correspondant à une décomposition. Dans ce cas le terme existe dans le *heap* et un registre de base, nommé `S`, contient son adresse. L'unification des sous-termes peut avoir lieu par rapport à `S`.

WRITE: Correspondant à une construction. Dans ce cas le terme est construit sur le sommet du *heap*.

Pour éviter l'utilisation d'un registre spécifique pour coder le mode, le registre `S` est mis à `NULL` en mode WRITE.

La figure 4.2 représente le contenu du *heap* dans le cas simple d'un programme Prolog formé d'un seul fait: `p(Z, h(Z, W), f(W)).`

0	STR	1
1	h/2	
2	REF	2
3	REF	3
4	STR	5
5	f/1	
6	REF	3
7	STR	8
8	p/3	
9	REF	2
10	REF	1
11	STR	5

Figure 4.2 - *Représentation du heap pour p(Z, h(Z, W), f(W)).*

4.2 Ensemble des registres de la WAM

La WAM possède plusieurs registres qui représentent l'état de la machine à tout instant. Ils peuvent être classés en quatre grandes catégories:

1. Registres de continuation:

   CP: Prochaine suite de buts à prouver;

   E: Bloc local de CP;

2. Registres de retour-arrière:

   B: Dernier bloc de choix;

   HB: Sommet de la pile de recopie associé à B;

3. Sommets de piles:

   A: Sommet de la pile locale;

   H: Sommet de la pile de recopie;

   TR: Sommet de la pile de restauration;

4. Registres de gestion de l'appel du but courant:

   P: Prochain but.

   A[i]: Sauvegarde des arguments du but courant;

Les registres arguments (notés `A[i]`) servent d'interface pour les données entre l'appelant et l'appelé. Ces registres sont chargés par l'appelant et sont unifiés à la tête de clause par l'appelé. Ceci a pour effet de charger son environnement (les variables de la clause reçoivent

en effet leurs valeurs grâce à l'unification). Si l'unification réussit, la clause est utilisable; pour chacun des prédicats du corps de la clause les registres sont chargés avec les arguments appropriés et le contrôle est transféré au prédicat concerné. S'il est possible de détecter les variables de sorte qu'entre leur première et dernière occurrence aucun appel à un prédicat ne sera fait, alors celles-ci peuvent être gérées directement dans les registres plutôt que dans l'environnement. De telles variables sont qualifiées de *temporaires* (notées `X[i]`) par opposition aux variables gérées à travers l'environnement qui sont dites *permanentes* (notées `Y[i]`).

Une variable temporaire est donc une variable n'apparaissant que dans un seul but, la tête et le premier but ne comptant que pour un.

Évidemment, il n'y a aucune différence entre les registres `A[i]` et `X[i]`. Physiquement, il s'agit du même et unique registre. On distingue les deux notations uniquement pour bien préciser les concepts utilisés. L'avantage des variables temporaires réside dans le fait que certaines instructions de chargement et de récupération d'arguments pourront donner lieu uniquement à des instructions de copie, par exemple charger `A[1]` avec le contenu de `X[1]`.

La figure 4.3 montre l'organisation de la mémoire dans la WAM.

4.3 Jeu d'instructions de la WAM

Du fait de l'utilisation de registres arguments pour assurer l'échange des données, un prédicat peut être compilé de manière indépendante de tout contexte. Ainsi, l'unité de compilation est le prédicat. De plus, la compilation d'une des clauses du prédicat ne nécessite aucune connaissance des autres clauses. Seules les instructions de gestion des points de choix (i.e. d'indexation) nécessitent une "vision globale" de toutes les clauses.

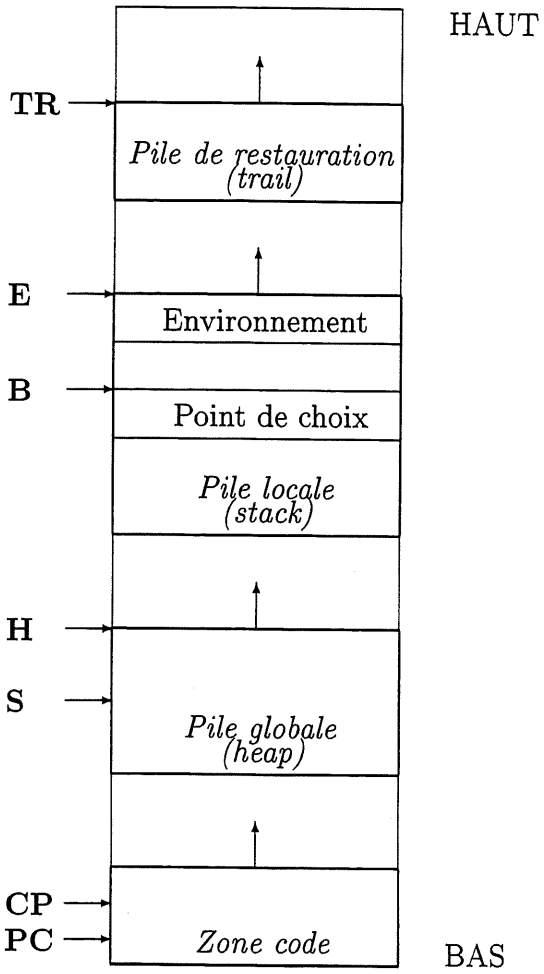

Figure 4.3 - *Organisation de la mémoire dans la WAM*

La principale caractéristique du jeu d'instructions de la WAM est basée sur l'étude statique du prédicat pour déterminer statiquement à la compilation des situations qui, sans cela, ne seraient détectées que dynamiquement à l'exécution. Pour chaque cas détectable une séquence d'instructions adaptée est générée. Par exemple, la gestion des points de choix est prise en charge par des instructions spécialisées (création du point de choix par la première clause, mise à jour par les autres excepté la dernière qui se charge de sa suppression). De même l'unification est décomposée en fonction des arguments de la clause pour éviter l'appel à la fonction générale d'unification entre deux termes quelconques.

Les instructions de la WAM peuvent être classées en quatre groupes:

1. Instructions de récupération des registres.
2. Instructions de chargement des registres.
3. Instructions de contrôle.
4. Instructions d'indexation (gestion des points de choix).

4.3.1 Instructions de récupération des registres

Ces instructions sont produites par la compilation de la tête d'une clause. Rappelons que l'unification d'une tête de clause avec les arguments a deux buts distincts quoique réalisés conjointement: vérifier que la clause est utilisable et initialiser ses variables. En particulier, une variable *singleton* (i.e. n'ayant qu'une occurrence dans la clause) ne demande aucun traitement puisqu'elle est unifiable avec tout terme et qu'il ne sert à rien de la renseigner du fait qu'elle n'a qu'une seule occurrence.

Warren a décomposé l'unification pour des raisons évidentes de performance. En effet, lors de la compilation il est possible de distinguer les cas suivants pour chaque argument de la tête de la clause:

- première occurrence d'une variable (donc pas encore liée);
- autre occurrence d'une variable (donc liée);
- constante;
- entier;
- liste vide;
- liste non vide (elle contient un *Car* et un *Cdr*);
- structure.

Dans le cas de termes composés, l'unification des sous-termes est également décomposée grâce à des instructions spécifiques. Dans la présentation qui suit le i^{eme} registre est noté `A` plutôt que `Ai`. Quant à l'écriture `V`, elle dénote une variable temporaire `Xj` ou permanente `Yj`.

Le code WAM généré pour unifier le i^{eme} registre avec le i^{eme} argument de la tête dépend de ce dernier comme suit:

- première occurrence d'une variable `V`:

  ```
  get_variable V, A
  ```

- autre occurrence d'une variable `V`:

  ```
  get_value V, A
  ```

- constante `C`:

  `get_constant C, A`

- entier `N`:

  `get_integer N, A`

- liste vide:

  `get_nil A`

- liste non vide:

  `get_list A`

  `unify_...` (unification du *Car*)

  `unify_...` (unification du *Cdr*)

- structure `F`:

  `get_structure F, A`

  `unify_...` (unification du premier sous-terme)

  ...

  `unify_...` (unification du dernier sous-terme)

L'instruction `get_variable V, A` effectue une simple copie de `V` dans `A`. L'instruction `get_value V, A` applique la fonction d'unification sur `V` et `A`.

L'instruction `get_constant C, A` vérifie que `A` est lié à la constante `C` ou à une variable libre, auquel cas elle lie celle-ci à `C`.

L'instruction `get_integer N, A` procède de manière similaire.

La récupération d'un terme structuré utilise les instructions spécialisées `unify_....` L'unification relative à un terme composé a deux comportements distincts suivant la nature de l'argument à unifier avec le terme:

- si c'est un terme de même foncteur et de même arité, alors une vraie unification doit avoir lieu sur les sous-termes (mode READ);
- si c'est une variable libre, il y a création du terme sur le *heap* et liaison de la variable à celui-ci (mode WRITE).

L'instruction `get_list A` déréférence `A`; si le mot obtenu est une variable libre elle est liée à une liste créée sur le *heap* (les instructions `unify_...`créeront le *Car* et le *Cdr*). Si le mot est une liste, les instructions `unify_...` unifieront le *Car* et le *Cdr*.

L'instruction `get_structure F, A` s'exécute de manière similaire.

Le code associé à la compilation d'un sous-terme dépend de sa nature comme suit:

- première occurrence d'une variable `V`:

  si `V` n'est pas une variable singleton: `unify_variable V`; sinon, soit `K` le nombre de variables singletons successives: `unify_void K`.

- autre occurrence d'une variable `V`:

  s'il est possible de déterminer statiquement si sa première occurrence a conduit à une liaison avec le *heap* i.e. si sa première occurrence est dans une structure, ou si elle est temporaire, sa première occurrence est dans le corps: `unify_value V`; sinon: `unify_local_value V`.

- constante `C`:

  `unify_constant C`

- entier `N`:

  `unify_integer N`

- liste vide:

  `unify_nil`

L'instruction `unify_variable V` lie `V` au contenu du registre `S` en mode READ et à une variable empilée sur le *heap* en mode WRITE. L'instruction `unify_value V` unifie `V` au contenu du registre `S` en mode READ et empile `V` sur le *heap* en mode WRITE. Dans ce cas, il faut être certain que cela n'entraînera pas de liaison du *heap* vers la pile locale. C'est la raison d'être de l'instruction `unify_local_value V`, qui, en mode READ, opère comme `unify_value V` et, en mode WRITE, commence par déréférencer `V`; si le mot obtenu est une variable libre permanente alors il est nécessaire de la globaliser sinon il y a empilement de ce mot (et non pas de `V`) sur le *heap*.

L'instruction `unify_void K` permet d'optimiser les variables singleton dans les structures. En mode READ cette instruction ajoute `K` à `S`, en mode WRITE, elle empile `K` variables libres sur le *heap*.

L'instruction `unify_constant C` est similaire à `get_constant C, <contenu de S>` en mode READ, et en mode WRITE, elle empile la constante `C` sur le *heap*.

L'instruction `unify_integer N` procède de manière similaire.

4.3.2 Instructions de chargement des registres

Comme pour les instructions de récupération, il faudra distinguer le type de l'argument à charger. Il sera nécessaire de prendre en compte les variables dangereuses. Un variable dangereuse est une variable permanente dont la première occurrence n'est ni dans la tête de la clause ni dans une structure. Dans la clause suivante: `p(X) :- q(r(Y), Z), s(Y, Z).` la variable `Z` est dangereuse alors que les variables `X` et `Y` ne le sont pas.

Ici encore l'instruction dépend du i^{eme} argument d'un but à charger comme suit:

- première occurrence d'une variable `V`:

  `put_variable V, A`

- autre occurrence d'une variable `V`:

  si `V` n'est pas dangereuse ou si le but courant n'est pas le dernier:
  `put_value V, A`; sinon: `put_unsafe_value V, A`

- constante `C`:

  `put_constant C, A`

- entier `N`:

  `put_integer N, A`

- liste vide:

  `put_nil A`

- liste non vide:

  `put_list A`

`unify_...` (chargement par recopie du *Car*)

`unify_...` (chargement par recopie du *Cdr*)

- structure `F`:

  `put_structure F, A`

  `unify_...` (chargement par recopie du premier sous-terme)

  ...

  `unify_...` (chargement par recopie dernier sous-terme)

L'instruction `put_variable V, A` initialise `V` et `A` avec une variable libre si `V` est permanente sinon elle lie `V` et `A` à une variable libre empilée sur le *heap*.

L'instruction `put_value V, A` effectue une simple copie de `V` dans `A`. Dans le cas d'une variable permanente du dernier but, il faut s'assurer que la copie ne liera pas `A` à l'environnement courant, pour pouvoir récupérer l'environnement de façon sûre. L'instruction `put_unsafe_value V, A` prend en charge les variables dangereuses susceptibles de créer des références fantômes. Cette instruction déréférence `V` et teste si le mot obtenu pointe vers l'environnement courant. Dans l'affirmative, il y a globalisation de la variable, sinon `put_unsafe_value V, A` copie ce même mot (et non pas V) dans `A`. Donc `put_unsafe_value V, A` ne se comporte jamais comme `put_value V, A` puisqu'elle copie le mot déréférencé dans `A` alors que `put_value V, A` y copie `V`. Une variable dangereuse ayant n occurences dans le dernier but nécessitera n instructions `put_unsafe_value V, A`. La première effectuera l'éventuelle globalisation et les autres copieront le mot déréférencé.

L'instruction `put_constant C, A` initialise `A` avec la constante `C` et `put_integer N, A` procède de manière similaire.

L'instruction `put_list A` initialise `A` avec <LST, H> et le mode à WRITE de manière à ce que les instructions `unify_...` qui suivent recopient le Car et le Cdr sur le *heap*.

L'instruction `put_structure F, A` effectue un traitement semblable.

4.3.3 Instructions de contrôle

Le rôle de ces instructions est de gérer les appels et retours de procédures ainsi que les environnements. De par la définition des variables permanentes, les faits et les clauses dont le corps est réduit à un seul but ne nécessitent pas d'environnement. De plus, l'appel du dernier but est distingué des autres dans la mesure où il doit se charger du retour. En fait, une instruction de retour existe (pour les faits), et on pourrait considérer l'appel du dernier but comme un appel quelconque suivi de l'instruction de retour, mais ce serait moins performant. Tout ceci conduit aux instructions de contrôle suivantes:

- pour un fait `p(...).` :

  <récupération des registres>

  `proceed`

- pour une clause `p(...) :- q(...).`:

  <récupération des registres>

  <chargement des registres pour le but `q`>

  `execute q`

- pour une clause `p(...) :-` q_1`(...),` q_2`(...), ...,` q_k`(...).`:

  `allocate`

<récupération des registres>

<chargement des registres pour le but q_1>

`call` q_1

<chargement des registres pour le but q_2>

`call` q_2

...

<chargement des registres pour le but q_k>

`deallocate`

`execute` q_k

L'instruction `allocate` crée un environnement sur la pile. L'instruction `deallocate` permet de récupérer cet environnement.

L'instruction `call p/n` initialise `CP` à l'adresse qui suit le `call` et donne le contrôle au prédicat `p/n`. L'instruction `execute p/n` procède pareillement sans toutefois modifier le contenu de `CP` préalablement restauré par l'instruction `deallocate`.

Le retour de procédure est assuré par l'instruction `proceed` qui se contente donc d'affecter `CP` à `PC`.

4.3.4 Instructions d'indexation

Ces instructions permettent de regrouper le code de chaque clause d'un prédicat et sont donc les instructions de plus haut niveau. Elles ont la responsabilité de gérer les points de choix.

Si un prédicat est formé des clauses C_1, C_2, ..., C_n, le code suivant est alors généré:

`try_me_else` L_2

<code pour C_1>

L_2:

`retry_me_else` L_3

<code pour C_2>

...

L_n :

`trust_me`

<code pour C_n>

L'instruction `try_me_else L` a la charge de la création d'un point de choix dans lequel elle désigne comme alternative le code d'adresse `L`. L'instruction `retry_me_else L` a pour rôle de restaurer les registres de base et de mettre à jour le point de choix en précisant que la nouvelle alternative est `L`. Enfin, l'instruction `trust_me` restaure les registres de base et supprime le point de choix. Pour ces trois instructions le contrôle se poursuit par l'instruction suivante du code.

Chapitre 5

LA CONCEPTION DE COP-COMPILÉ

En partant de la discussion des limitations de COPi et des améliorations proposées dans le chapitre 3, un nouveau système qui permet de concrétiser ses améliorations tout en respectant les objectifs initiaux du projet est conçu.

Ce chapitre définit la structure du système COP-Compilé. Il décrit les modules qui le forment ainsi que leur répercussion sur la structure des fichiers et le schéma de compilation COP.

5.1 Structure du système COP-Compilé

La structure globale du système COP-Compilé est identique à celle de COP*i*. Il est formé des mêmes trois principaux modules:

1. Le séparateur COP.
2. Le convertisseur COP.
3. Le compilateur Prolog-COP.

Les deux premiers modules, le séparateur COP et le convertisseur COP, sont restés inchangés puisqu'ils n'avaient aucun effet sur la performance de COP*i*. Par contre, le troisième module le compilateur Prolog-COP a été entièrement mis à jour.

5.1.1 Séparateur COP

Le séparateur COP réalise deux tâches essentielles:

1. Séparer les fichiers de type COP en deux ensembles de fichiers: un ensemble de fichiers de type *bridge* et un ensemble de fichiers de type Prolog.
2. Bâtir la table de symboles qui contient les foncteurs et les arités de tous les prédicats Prolog reconnus.

La section 2.3.1 décrit en détail le fonctionnement de ce module.

5.1.2 Convertisseur COP

Le convertisseur COP permet de générer le code liant le code C++ et le code Prolog. Il effectue les tâches suivantes:

1. Convertir les fichiers de type *bridge* en fichiers de type C++.
2. Générer le fichier de prototypes.
3. Générer le fichier de fonctions *bridge*.

La section 2.3.2 décrit en détail le fonctionnement de ce module.

5.2 Compilateur Prolog-COP

Le compilateur Prolog-COP du système COP-Compilé permet de compiler le fichier source Prolog. Cette compilation est réalisée grâce à une conversion du code Prolog en instructions de la WAM. Ces instructions seront exécutées par un émulateur de la machine WAM.

Le compilateur Prolog-COP est formé de trois principaux modules:

1. L'optimisateur Prolog.
2. Le compilateur Prolog-WAM.
3. L'émulateur WAM.

La figure 5.1 montre la structure du compialteur Prolog-COP du système COP-Compilé.

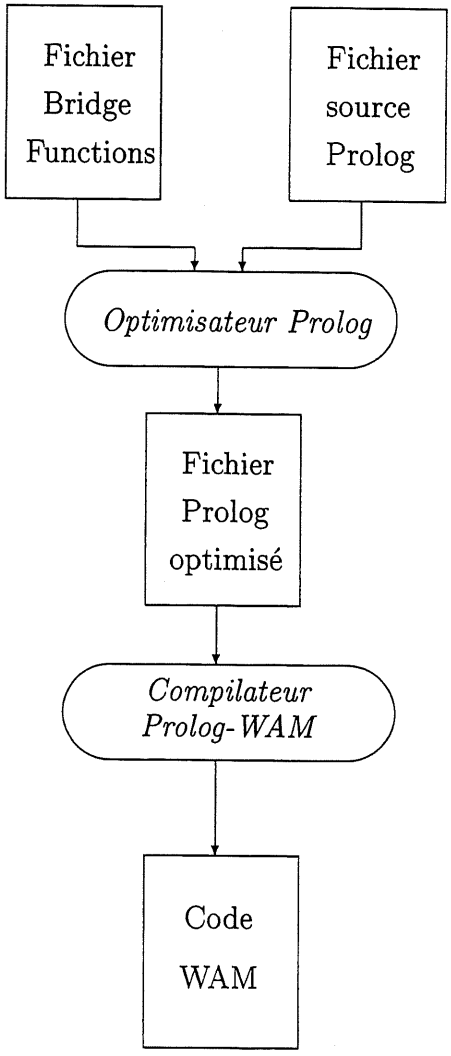

Figure 5.1 - *Flux des données dans le compilateur Prolog-COP de COP-Compilé*

5.2.1 Optimisateur Prolog

Ce module du compilateur Prolog-COP est une originalité par rapport au système COP*i*. Il permet d'optimiser l'étape de compilation du code Prolog en instructions WAM. Il localise

l'ensemble des prédicats qui doivent être compilés parmi ceux du code Prolog entier. Ainsi, seuls les prédicats pertinents seront compilés et non pas la totalité du code Prolog.

L'opération d'optimisation du code Prolog se fait en deux étapes:

1. Détection des dépendances entre les prédicats.
2. Sélection des prédicats à compiler.

Détection des dépendances

Lors de cette première étape, le fichier source Prolog est parcouru et son graphe de dépendances est construit au fur et à mesure en mémoire.

Ce graphe de dépendances reflète les liens qui existent entre les prédicats. À chaque prédicat du programme Prolog on associe les prédicats correspondants aux buts qu'il faut effacer pour que le prédicat réussisse.

Sélection des prédicats

Lors de cette deuxième étape, le prédicat requête c'est à dire celui correspondant à une fonction *bridge* est déterminé à partir du fichier des *bridge-functions*.

Le graphe de dépendances est parcouru pour sélectionner tous les prédicats qui, de manière récursive, sont en relation avec le prédicat requête.

Ce sous-ensemble de prédicats ainsi déterminé sera le seul à être compilé en instructions de la WAM pour cette fonction *bridge*.

5.2.2 Compilateur Prolog-WAM

La compilateur Prolog-WAM reçoit en entrée le fichier Prolog optimisé et convertit ses prédicats en instructions de la WAM.

5.2.3 Émulateur WAM

L'émulateur WAM est un programme en C++ qui représente l'implantation de la machine abstraite de Warren.

Il reçoit en entrée le code WAM correspondant au programme Prolog et l'exécute.

Le code WAM sous forme de *byte-code* en C++ et le code de l'émulateur de la WAM représentent ensemble le fichier C++ à la sortie du compilateur Prolog-COP.

5.3 Schéma de compilation de COP-Compilé

Le schéma de compilation du système COP-Compilé est identique à celui de COP*i*. Une seule différence majeure réside au niveau de l'étape de compilation du code Prolog qui se faisait au niveau de COP*i* par interprétation de *byte-code* et qui, au niveau de COP-Compilé, passe par la conversion du code Prolog en instructions de la WAM qui vont être exécutées par un émulateur de la WAM.

La figure 5.2 montre la structure des fichiers et le schéma de compilation de COP-Compilé. Elle ressemble à la figure 2.4 qui représente la structure de COP*i*. La seule différence est que dans COP-Compilé le fichier des *bridge-functions* est reçu en entrée du compilateur Prolog-COP (la ligne en pointillés sur la figure) afin de permettre à l'optimisateur Prolog de

déterminer les prédicats Prolog à compiler.

L'annexe A présente un exemple complet d'un programme COP compilé par COP-Compilé. On y retrouve les différents fichiers schématisés à la figure 5.2.

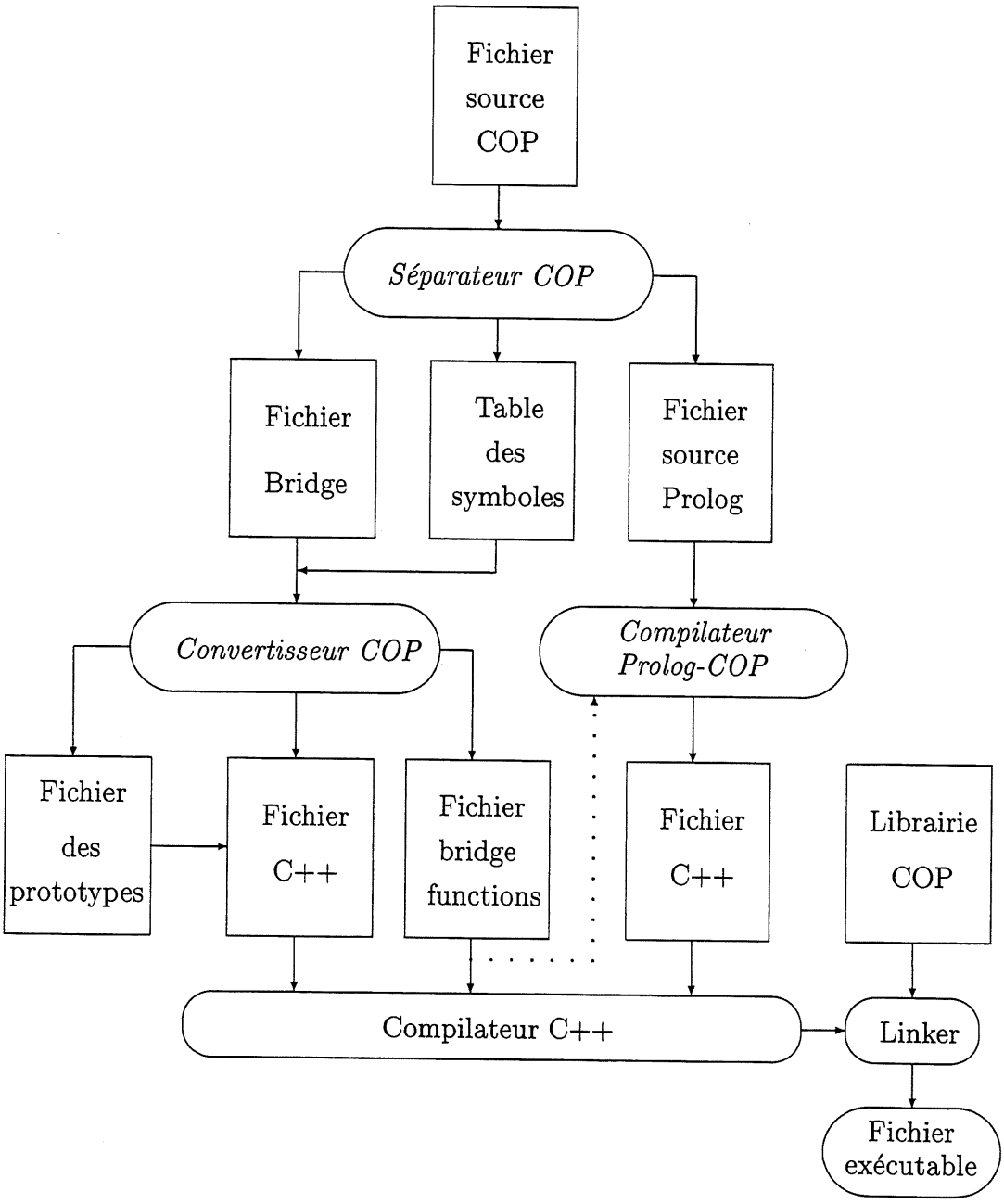

Figure 5.2 - *Flux des données et schéma de compilation de COP-Compilé*

Chapitre 6

LA RÉALISATION DE COP-COMPILÉ

Ce chapitre présente les grandes lignes de réalisation du système COP-Compilé. Il explique les détails des deux plus grandes tâches effectuées par le système, à savoir la compilation du code Prolog en instructions de la WAM et l'exécution de ces instructions par un émulateur de la machine WAM. De plus, il décrit de manière quantitative les performances du système COP-Compilé par rapport au système COP*i*.

6.1 Compilation de Prolog en WAM

Le module compilateur Prolog-WAM permet de générer le programme WAM équivalent à un programme Prolog.

Il reçoit en entrée un fichier formé de prédicats Prolog qu'il analyse avant de produire en

sortie un fichier formé d'instructions WAM.

Le module est écrit en `C++`. Il est formé d'un analyseur lexical en `Lex` [22] et d'un analyseur syntaxique en `Yacc` [23].

L'analyseur lexical permet de détecter les unités lexicales de Prolog suivantes:

1. Les variables.
2. Les identificateurs.
3. Les chaînes de caractères.
4. Les entiers.

L'analyseur syntaxique permet de compiler le programme Prolog entré au fur et à mesure qu'il vérifie que le programme respecte la grammaire de Prolog reconnue.

6.1.1 Compilation du programme Prolog

Les instructions WAM correspondant à tous les prédicats du programme Prolog sont mises les unes à la suite des autres pour former le programme WAM équivalent au programme Prolog entré.

Chaque ensemble d'instructions WAM correspondant à un prédicat Prolog est précédé par une étiquette signalant le nom et l'arité du prédicat.

6.1.2 Compilation d'un prédicat du programme

Les instructions WAM résultant de la compilation de toutes les clauses d'un même prédicat du programme Prolog sont reliées ensemble grâce aux instructions d'indexation.

6.1.3 Compilation d'une clause du prédicat

Une clause d'un prédicat peut être compilée indépendamment des autres clauses du même prédicat.

Le code WAM correspondant à une clause est formé à partir des instructions WAM correspondant à chaque littéral de la clause reliées par des instructions de contrôle.

La compilation d'une clause de Prolog vers WAM est réalisée selon les étapes suivantes:

Numérotation des variables

Les variables de chaque clause du programme Prolog sont remplacées par un numéro qui correspond à leur ordre d'apparition dans la clause. Ce numéro sera, par la suite, celui du registre de la machine WAM qui va contenir la variable.

Par exemple, la clause suivante:

```
p(X) :- q(r(Y, W), Z), s(Y, Z).
```

devient:

```
p(VAR(0)) :- q(r(VAR(1),VAR(2)),VAR(3)),s(VAR(1),VAR(3)).
```

Génération des instructions WAM de base

Chaque terme de la tête de la clause est remplacé par une instruction WAM qui correspond à sa nature:

- une constante est remplacée par l'instruction `get_constant`;
- une variable est remplacée par l'instruction `get_variable`;
- une liste est remplacée par l'instruction `get_list`;
- un entier est remplacé par l'instruction `get_integer`;
- une structure est remplacée par l'instruction `get_structure` et ses sous-termes par l'instruction correspondant à leur nature (`unify_constant`, `unify_variable` ou `unify_integer`).

Dans l'exemple précédent, La tête de la clause devient:

```
get_variable(0, 0)
```

Chaque terme dans les buts du corps de la clause est remplacé par une instruction WAM qui correspond à sa nature:

- une constante est remplacée par l'instruction `put_constant`;
- une variable est remplacée par l'instruction `put_variable`;
- une liste est remplacée par l'instruction `put_list`;
- un entier est remplacé par l'instruction `put_integer`;

- une structure est remplacée par l'instruction `put_structure` et ses sous-termes par l'instruction correspondant à leur nature (`unify_constant`, `unify_variable` ou `unify_integer`).

Dans l'exemple précédent, le but `q` du corps de la clause sera traduit en cette séquence d'instructions de base:

```
put_structure(r/2, 0)
unify_variable(1)
unify_variable(2)
put_variable(3, 1)
```

Le but `s` du corps de la clause sera traduit en cette séquence d'instructions de base:

```
put_variable(1, 0)
put_variable(3, 1)
```

Le deuxième paramètre des instructions est le numéro de registre auquel sera affectée la variable. Pour chaque but du corps de la clause, ce nombre correspond à un un compteur qui est initialisé à zéro et qui est incrémenté à chaque affectation.

Détermination des variables permanentes et temporaires

Les variables permanentes de la clause sont déterminées. Ce sont les variables qui apparaissent dans plusieurs buts de la clause, la tête et le premier but ne comptant que pour un. Les variables qui ne sont pas permanentes sont des variables temporaires.

En reprenant l'exemple précédent, on voit bien que les variables `Y` et `Z` sont permanentes alors que les variables `X` et `W` sont temporaires.

La tête de la clause devient alors:

```
get_variable(0, temp, 0)
```

Le but `q` du corps de la clause devient:

```
put_structure(r/2, 0)
unify_variable(1, perm)
unify_variable(2, temp)
put_variable(3, perm, 1)
```

Le but `s` du corps de la clause devient:

```
put_variable(1, perm, 0)
put_variable(3, perm, 1)
```

Détermination des variables dangereuses

Les variables dangereuses de la clause sont déterminées. Ce sont les variables permanentes dont la première apparition dans la clause n'a été ni dans la tête, ni dans une structure.

En reprenant l'exemple précédent, on voit bien que la variable `Z` est dangereuse.

Le but `q` du corps de la clause devient:

```
put_structure(r/2, 0)
```

```
unify_variable(1, perm)
unify_variable(2, temp)
put_variable(3, danger, 1)
```

Le but s du corps de la clause devient:

```
put_variable(1, perm, 0)
put_variable(3, danger, 1)
```

Détermination des variables singletons

Les variables singletons de la clause sont déterminées. Ce sont les variables qui n'apparaissent qu'une seule fois dans la clause.

Pour l'exemple considéré, la variable W est une variable singleton.

Le but q du corps de la clause devient:

```
put_structure(r/2, 0)
unify_variable(1, perm)
unify_variable(2, single)
put_variable(3, danger, 1)
```

Le but s du corps de la clause devient:

```
put_variable(1, perm, 0)
put_variable(3, danger, 1)
```

Raffinement des instructions de traitement des variables

Les instructions de base de traitement des variables (`get_variable`, `put_variable` et `unify_variable`) sont remplacées par des instructions plus spécifiques au contexte et à la nature de la variable:

- L'instruction `get_variable` est remplacée par `get_value` s'il s'agit d'une autre occurence de la même variable dans la tête de la clause.
- L'instruction `put_variable` est remplacée par `put_value` s'il s'agit d'une autre occurence de la même variable dans le corps de la clause et que cette variable n'est pas dangereuse.
- L'instruction `put_variable` est remplacée par `put_unsafe_value` s'il s'agit d'une autre occurence de la même variable dans le corps de la clause et que cette variable est dangereuse.
- L'instruction `unify_variable` est remplacée par `unify_value` s'il s'agit d'une autre occurence de la même variable.
- L'instruction `unify_variable` est remplacée par `unify_void` s'il s'agit d'une variable singleton.

Dans l'exemple considéré, le but `q` du corps de la clause devient:

```
put_structure(r/2, 0)
unify_variable(1, perm)
unify_void
put_variable(3, danger, 1)
```

Le but s du corps de la clause devient:

```
put_value(1, perm, 0)
put_unsafe_value(3, 1)
```

Après l'ajout des instructions de contrôle et l'adoption de la notation standard de la WAM, le code WAM correspondant à l'exemple traité devient le suivant:

```
p/1 :
get_variable X0,A0
put_structure r/2,A0
allocate
unify_variable Y1
unify_void 1
put_variable Y3,A1
call q/2
put_value Y1,A0
put_unsafe_value Y3,A1
deallocate
execute s/2
```

6.2 Exécution du code WAM

6.2.1 Interface entre le code C++ et le code WAM

Les *bridge-goals* dans le code C++ sont convertis par le compilateur COP en code C++. Ce code correspond à un appel de fonction, la *bridge-function*. Chaque *bridge-goal* a une *bridge-function* qui lui est exclusive. Le but des *bridge-functions* est de réaliser l'interface entre le code C++ et le code Prolog traduit en instructions WAM.

Une *bridge-function* est implantée grâce aux méthodes de deux classes importantes: `TProlog` et `TBridgeVariable`.

Classe TProlog

La classe `TProlog` permet d'échanger les informations entre le code C++ et le code Prolog, de demander l'exécution d'un but Prolog et d'implanter la machine WAM.

La classe `TProlog` permet de cacher aux programmes COP les détails de l'implantation de la machine Prolog. C'est grâce à cette classe que le passage de COP*i* (où le code Prolog est interprété) à COP-Compilé (où le code Prolog est compilé) est rendu facile puisque les changements à faire n'ont affecté que les méthodes de cette classe.

Les méthodes de la classe `TProlog` ne sont jamais appelées directement par le programmeur sauf pour la methode `reset`. C'est le compilateur COP qui les utilise pour constituer le code des *bridge-functions*.

L'annexe B fournit les détails de l'implantation de la classe `TProlog`.

Classe TBridgeVariable

La classe `TBridgeVariable` fournit au programmeur une interface pour manipuler les *bridge-variables*. Les *bridge-variables* sont des instances d'une classe C++ dans le code C++ et des variables logiques dans le code Prolog.

La classe `TBridgeVariable` permet de créer et de manipuler les types de données définis dans Prolog, à savoir: un atome, un entier, une chaîne de caractères, une liste et un terme. Ainsi, une *bridge-variable* possède un type et une valeur.

La manipulation d'une *bridge-variable* avec des valeurs de type simple, comme entier ou chaîne de caractères, se passe de façon assez régulière car ces types de données existent en C++. Par contre, les listes et les termes du langage Prolog sont convertis en une structure de liste doublement chainée.

L'annexe C fournit les détails de l'implantation de la classe `TBridgeVariable`.

Rôle d'une bridge-function

Une bridge-function réalise quatres principales tâches:

1. Le passage des arguments reçus et la spécification du but à exécuter à l'environnement WAM. Les paramètres de la *bridge-function* sont les paramètres du but Prolog à appeler. Ces paramètres sont des *bridge-variables* donc des instances de la classe `TBridgeVariable`. Les *bridge-variables* reçues en paramètres sont elles-mêmes passées en paramètres à une méthode transformant ces variables dans une représentation adéquate pour la machine WAM.

La méthode faisant cette transformation est `setupGoal` qui fait partie de la classe `Tprolog`. Elle permet également de spécifier le but à exécuter par la machine WAM. En effet, le nom du prédicat Prolog est passé en paramètre à la méthode `setupGoal` afin de pouvoir spécifier le but à exécuter. Après cette étape, l'environnement WAM possède les informations nécessaires pour l'exécution du but desiré: le nom du but et ses paramètres. La prochaine étape est d'exécuter ce but.

2. L'exécution du but Prolog. Cette étape est réalisée grâce à la méthode `run` de la classe `TProlog`. Cette méthode n'a pas d'arguments puisque l'environnement WAM a toutes les informations nécessaires pour l'exécution, par contre elle a une valeur de retour indiquant le succès ou l'échec du but exécuté.

   Lors de l'exécution d'un *bridge-goal*, seule la première solution est générée mais les points de choix doivent être conservés dans la machine WAM au cas où un retour-arrière est exigé en appelant successivement cette même *bridge-function.*

   Ainsi, les points de choix ne sont pas détruits automatiquement à la sortie d'une *bridge-function* mais seulement si une *bridge-function* différente de la précédente est appelée. La machine WAM doit donc pouvoir distinguer si oui ou non elle doit détruire ses points de choix avant l'exécution du but de cette *bridge-function.* Après l'exécution du but à l'aide de la méthode `run`, il faut récuperer les résultats dans la machine WAM.

3. La récuperation des résultats. Après l'exécution du but, si une solution est trouvée, la *bridge-function* doit transférer les résultats depuis les structures internes de la machine WAM vers les variables paramètres de la *bridge-function.* La méthode `getVariables` de la classe `TProlog` permet de réaliser cette tâche.

   L'effet de bord sur les arguments d'une *bridge-function* est réalisé avec le passage des paramètres par référence. Il est ainsi possible de passer des valeurs, variables en entrée, et de récupérer les résultats, variables en sortie, via les paramètres des *bridge-functions.*

4. Retour d'une valeur booléenne par la *bridge-function*. Ainsi, la fonction appelante permet de détecter le succès ou l'échec du but Prolog exécuté. La valeur de retour de la méthode run est une valeur adéquate pour cette tâche car c'est elle qui indique le résultat de l'exécution du but Prolog. Cette valeur est sauvegardée dans une variable temporaire pour servir de valeur de retour.

La figure 6.1 montre le code squelette en C++ d'une *bridge-function*. Elle met en relief les tâches décrites précédemment.

```
int FONCTEUR_ARITE_COMPTE( PARAMETRES ) {
  char *foncteur = "FONCTEUR_COMPTE";
  TBridgeVariable *parmList[] = { LISTE_PARAMETRES };
  int returnValue;

  prolog.setupGoal(fonctor, parmList);
  returnValue = prolog.run();
  if(returnValue)
    prolog.getVariables();
  return returnValue;
}
```

Figure 6.1 - *Code squelette d'une bridge-function*

6.2.2 Émulation du code WAM

Représentation de la mémoire

La mémoire de l'émulateur est implantée sous forme de deux tableaux dynamiques: un premier tableau pour stocker le code WAM et un autre pour représenter les piles de la machine. Ce deuxième tableau est subdivisé, de bas en haut, en trois zones:

1. La pile globale (*heap*).
2. La pile locale.
3. La pile de restauration (*trail*).

Cette organisation respecte la structure originale de la mémoire de la WAM telle que schématisée à la figure 4.3.

Représentation des données

Les termes sont codés dans la mémoire de l'émulateur sous forme d'entiers etiquetés de `32 bits`. Les trois bits de poids fort de l'entier représentent le champ étiquette qui permet de distinguer entre les différents types de termes suivants:

1. Une constante.
2. Une liste vide.
3. Une liste non vide.
4. Une variable non liée.

5. Une variable liée.

6. Une structure.

7. Un entier.

Les 29 bits restants représentent le champ valeur du terme. Le contenu de ce champ dépend du type du terme:

- Pour une variable liée, le champ valeur est une référence vers le terme auquel est liée la variable.
- Pour une variable libre, le champ valeur est une référence vers la variable elle-même.
- Pour une constante, le champ valeur pointe vers une table de symboles stockant toutes les constantes.
- Pour un entier, le champ valeur code l'entier lui-même.
- Pour une liste non vide, le champ valeur pointe vers la tête de la liste. La queue de la liste venant juste après la tête.
- Pour une liste vide, le champ valeur code la constante `'[]'`.
- Pour une structure, le champ valeur pointe vers le foncteur de la structure. Consécutivement à ce mot viennent les sous-termes de la structure.

Représentation des instructions

Une instruction WAM est représentée en mémoire de l'émulateur sous forme d'une structure à deux champs (`type_inst`). Le premier champ (`codeop`) correspond au code opération

de l'instruction et le deuxième (donnee) aux données manipulées par l'instruction:

```
struct type_inst {
   int   codeop;
   union {
      int              nb_void;
      int              num_registre;
      struct reg_vbl   r_v;
      struct reg_const r_c;
      int              predicat;
   } donnee;
};
```

Le champ donnee peut contenir différents types de données en fonction de l'instruction courante.

- Pour les instructions unify_nil, proceed, allocate et deallocate le champ donnee est vide.
- Pour l'instruction unify_void le champ donnee contient le nombre de variables singletons dans le sous-champ nb_void.
- Pour les instructions call, execute, try_me_else, retry_me_else, trust_me et built_in le champ donnee contient l'adresse du prédicat à exécuter dans le sous-champ predicat.
- Pour les instructions get_list, get_nil, put_list, put_nil, unify_variable et unify_value le champ donnee contient le numéro du registre concerné dans le sous-champ num_registre.

- Pour les instructions `put_unsafe_value`, `get_value`, `put_variable`, `put_value` et `get_variable` le champ `donnee` contient le numéro de la variable et le numéro du registre où elle doit être mise, dans le sous-champ `r_v` qui est une instance de la structure suivante:

```
struct reg_vbl {
   int  num_variable;
   int  num_registre;
};
```

- Pour les instructions `unify_constant`, `unify_integer`, `put_constant`, `put_integer`, `put_structure`, `get_constant`, `get_integer` et `get_structure` le champ `donnee` contient l'adresse de constante dans la table de symboles ou la valeur de l'entier lui-même et le numéro du registre où elle doit être mise, dans le sous-champ `r_c` qui est une instance de la structure suivante:

```
struct reg_const {
   int  num_const;
   int  num_registre;
};
```

Fonctionnement de l'émulateur

L'émulateur est écrit en C++. Il fonctionne en deux phases. Il charge, d'abord, les instructions WAM converties en *byte-code* en mémoire. Ensuite, il exécute ces instructions.

L'exécution des instructions de la WAM par l'émulateur se fait de façon séquentielle. L'émulateur détermine la nature de l'instruction à partir de son code opération puis appelle

la fonction qui implante l'instruction en lui fournissant les paramètres présents dans le code de l'instruction.

Les fonctions d'exécution des instructions de la WAM reprennent les fonctionnalités décrites à la section 4.3. Les détails d'implantation de ces fonctions sont fournis dans le livre d'Aït-Kaci [19] où l'algorithme de chaque fonction est donné explicitement.

Si l'exécution réussit, les résultats sont passés stockés dans la *bridge-variable* qui a servi à déclencher l'exécution; sinon une indication d'échec est retournée.

Les points de choix sont sauvegardés en vue d'un appel successif au même but.

6.3 Tests et validation de COP-Compilé

Afin de prouver l'efficacité du système COP-Compilé et quantifier les améliorations qui ont été apportées au système COP*i*, des tests de performance ont été appliqués aux deux systèmes.

Vu que la structure générale des systèmes est pratiquement identique, l'analyse de performance a porté uniquement sur l'implantation de la machine Prolog. Elle est basés sur un interpréteur de Prolog dans le système COP*i* et sur un compilateur de Prolog à travers la WAM dans le système COP-Compilé.

Une série de programmes de tests a été adaptée des *benchmarks* standards utilisés pour mesurer la performance de l'implantation des systèmes Prolog [24].

Cette série de programmes de test a servi à évaluer la vitesse avec laquelle la machine

Prolog de chacun des deux systèmes manipule les aspects majeurs du langage Prolog, à savoir:

1. Les appels.
2. Le retour-arrière.
3. La gestion des environnements.
4. L'unification.

6.3.1 Description des tests

Tous les tests ont été réalisés sur une station de travail `Sun SPARC 10` tournant avec le système d'exploitation `Solaris`. Tous les facteurs pouvant influencer les résultats des tests, tels que le nombres d'utilisateurs de la machine ou le nombre de processus lancés, ont été minimisés de manière à avoir une comparaison correcte des systèmes testés.

Chaque programme de test consiste en un programme COP qui est exécuté consécutivement sur le système COPi et le système COP-Compilé. Il est exécuté 1000 fois et un temps moyen d'exécution est calculé à partir des durées obtenues à chaque exécution.

Le temps d'exécution est mesuré à partir de l'appel de la machine Prolog jusqu'à la sortie de la machine. Il correspond donc à la durée d'exécution de la partie Prolog du programme COP.

Afin d'uniformiser les résultats des tests, les temps d'exécution ont été convertis en nombre d'instructions logiques par seconde (`Lips`) qui est une unité standard de mesure de performance des systèmes de programmation logique.

Les programmes de tests utilisés sont les suivants:

Le programme boresea

Ce programme consiste en une séquence de 200 prédicats qui ne possèdent pas d'arguments, ni de points de choix. Il permet de tester l'effet des appels purs dans le système Prolog. Le nombre de Lips trouvé correspond à la performance maximale que peut atteindre le système.

Le programme choice_point

Ce programme consiste en une séquence de 20 prédicats qui servent à tester l'effet des appels impliquant la création de points de choix pour le retour-arrière. Le programme n'effectue pas de retour-arrière lui-même.

Le programme deep_back

Ce programme consiste en une séquence de 20 prédicats qui servent à tester l'effet du retour-arrière profond.

Le programme shallow_back

Ce programme consiste en une séquence de 20 prédicats qui servent à tester l'effet du retour-arrière superficiel.

Le programme cre_env

Ce programme consiste en une séquence de 12 prédicats qui servent à tester l'effet de la création d'environnements.

Le programme general_unif

Ce programme consiste en une séquence de 3 prédicats qui servent à tester l'effet de l'unification.

L'annexe D présente le code Prolog complet correspondant à chaque programme de test ainsi que le code WAM généré par le compilateur Prolog-WAM du système COP-Compilé.

6.3.2 Résultats des tests

Les résultats des tests décrits ci-dessus figurent au tableau 6.1. La première colonne fournit le nombre moyen d'instructions par seconde (`N1`) correspondant au programme test compilé par le système COPi. La deuxième colonne fournit le nombre moyen d'instructions par seconde (`N2`) correspondant au même programme test compilé par le système COP-Compilé. La troisième colonne présente, en pourcentage, l'amélioration du temps d'exécution obtenue. Cette amélioration est calculée grâce à la formule `(N2 - N1) / N1`.

Une analyse profonde des résultats illustrés par le tableau 6.1 conduit aux constatations suivantes:

TABLEAU 6.1 - *TEMPS D'EXÉCUTION DES PROGRAMMES DE TEST*

Programme de test	COPi	COP-Compilé	Amélioration
`boresea`	160 KLips	770 KLips	381%
`cre_env`	32 KLips	96 KLips	200%
`general_unif`	3 KLips	6 KLips	100%
`choice_point`	21 KLips	35 KLips	66%
`deep_back`	17 KLips	30 KLips	76%
`shallow_back`	26 KLips	40 KLips	53%

Le programme boresea

L'exécution de ce programme a pris approximativement `5 fois` moins de temps dans le système COP-Compilé que dans le système COPi. Cette performance est due essentiellement au bon choix de la représentation interne des prédicats dans le système COP-Compilé qui permet un parcours séquentiel plus rapide des prédicats.

Le programme cre_env

L'exécution de ce programme a pris `3 fois` moins de temps dans le système COP-Compilé que dans le système COPi. Ceci est dû essentiellement à la prise en charge de la création des environnements par la WAM.

Le programme general_unif

L'exécution de ce programme a pris `2 fois` moins de temps dans le système COP-Compilé que dans le système COPi. Ceci est dû essentiellement au fait que les instructions de la WAM

décompose la procédure d'unification de Prolog selon la nature des arguments à unifier. Cette performance peut toujours être améliorée en utilisant la technique d'optimisation de l'unification [25].

Les programmes choice_point, deep_back et shallow_back

L'exécution de ces trois programmes qui servent à tester les mécanismes du retour-arrière a été accélérée de 50% à 75%. Cette performance peut être davantage améliorée en utilisant la technique de retour-arrière intelligent [26].

Les résultats de ces tests valident donc le système COP-Compilé. Ils prouvent que le premier principal objectif escompté lors de la réalisation du système, à savoir la diminution du temps d'exécution, a été comblé avec satisfaction. Une accélération du temps d'exécution du système original allant de 50% à 400%, selon la nature des programmes, a été obtenue.

De plus, l'atteinte du deuxième principal objectif de réalisation du système, à savoir l'extension de l'ensemble de programmes Prolog reconnus, est confirmée par la panoplie de programmes Prolog contenant des prédicats prédéfinis arithmétiques, logiques, métalogiques ou de contrôle qui peuvent être compilés sur le système COP-Compilé sans qu'il soit même possible de les reconnaître par le système COP*i*.

6.3.3 Comparaison avec un Prolog classique

Après avoir validé le système COP-Compilé et prouvé que les objectifs escomptés de sa réalisation ont été tous atteints avec succès, on compare ses performances avec un système Prolog classique.

Cette comparaison a pour objectif de situer le système réalisé par rapport aux systèmes existants et évaluer l'effort qui reste à fournir pour améliorer la performance du système COP-Compilé à son maximum.

Le tableau 6.2 présente les résultats comparatifs des tests exécutés sur le système COP-Compilé et sur la version `1.5` de C-Prolog qui est considéré comme une référence dans le domaine des système de programmation logique.

TABLEAU 6.2 - *COMPARAISON AVEC UN PROLOG CLASSIQUE*

Programme de test	COP-Compilé	C-Prolog 1.5
`boresea`	770 KLips	800 KLips
`cre_env`	96 KLips	132 KLips
`general_unif`	6 KLips	30 KLips
`choice_point`	35 KLips	120 KLips
`deep_back`	30 KLips	70 KLips
`shallow_back`	40 KLips	102 KLips
`nrev`	3 KLips	12 KLips
`fib`	10 KLips	27 KLips

On constate que C-Prolog est plus rapide que COP-Compilé dans une proportion allant de 14% à 240% selon le programme test. Ces résultats sont satisfaisants puisqu'ils montrent qu'en travaillant sur l'optimisation du fonctionnement de COP-Compilé, ce dernier pourrait facilement atteindre la performance de C-Prolog et peut être même la dépasser.

Chapitre 7

CONCLUSION

Ce mémoire a présenté le système COP-Compilé. C'est un système qui permet l'intégration des langages C++ et Prolog en utilisant la machine abstraite de Warren pour la compilation du code Prolog en C++.

7.1 Bilan de COP-Compilé

L'objectif principal du système COP-Compilé était d'améliorer l'efficacité du système COP*i*. Il devait remédier aux limitations de COP*i* tout en respectant sa philosophie globale et son schéma de compilation.

Le système COP-Compilé a réussi à augmenter la performance du système COP*i* et améliorer ses possibilités en diminuant son temps d'exécution et en étendant les programmes Prolog qu'il est capable de reconnaître.

7.1.1 Diminution du temps d'exécution de COPi

L'amélioration du temps d'exécution dans le système COP-Compilé a été atteinte grâce à:

- un bon choix de la représentation interne des données en mémoire;
- la compilation du code Prolog à la place de son interprétation.

En effet, la technique de représentation étiquetée a été adoptée pour coder les termes du code Prolog. Ceci a permis de réduire le temps d'accès aux données en mémoire, réduisant ainsi le temps global d'exécution.

En outre, la compilation du code Prolog en convertissant les prédicats Prolog en instructions de la WAM a permis de réduire le temps d'exécution puisque les instructions de la WAM ont été conçues de manière à accélérer la procédure d'unification de Prolog.

7.1.2 Extension des programmes Prolog reconnus par COPi

L'ensemble des prédicats prédéfinis de Prolog reconnus a été étendu dans le système COP-Compilé afin d'inclure les prédicats arithmétiques, logiques, métalogiques et de contrôle.

L'utilisation de la technique de tableaux dynamiques pour le stockage des termes du code Prolog en mémoire a permis au système COP-Compilé de s'adapter à la taille du code Prolog permettant la compilation de programmes de grandeur réelle.

7.2 Travaux futurs

Le système COP-Compilé a amélioré quelques uns des aspects du projet COP global. Des travaux futurs pourront raffiner davantage le système. Ces travaux peuvent se faire sur deux principaux axes: un axe quantitatif et un autre qualitatif.

7.2.1 Améliorations quantitatives

1. Compilation directe en code C++ des prédicats Prolog au lieu de passer par un émulateur de la WAM. L'adoption de cette technique contribuerait à accélérer davantage la phase d'exécution. On pourrait s'inspirer des travaux de Levy et Horspool [27, 16], Codognet et Diaz [28, 29] et Demeon et Maris [30].

2. Adoption de différentes techniques d'optimisation de la WAM. Ces techniques qui résultent de recherches effectuées séparément sur la WAM permettront d'améliorer l'efficacité de la phase de compilation du code Prolog. On pourrait opter pour des techniques telles que:

   - l'optimisation du code [17, 31];
   - la gestion de la mémoire [18];
   - l'allocation des registres [32];
   - le retour-arrière intelligent [26];
   - l'optimisation de l'unification [25].

3. Possibilité d'appeler du code C++ à partir du code Prolog dans les programmes COP.

7.2.2 Améliorations qualitatives

1. Intégration de la partie objet de C++ dans le langage Prolog en permettant de passer des objets au code Prolog, les manipuler et les retourner au code C++.

2. Remplacement du langage C++ par le langage Java qui représente la tendance actuelle de la programmation procédurale; en d'autres termes, la conception d'un système JOP (Java Ou Prolog).

3. Adaptation du système aux nouvelles extensions de la programmation logique telles que:

   - les contraintes;
   - la concurrence;
   - le parallélisme;
   - l'ordre supérieur.

7.2.3 Suite suggérée du projet

Afin de suivre la tendance actuelle dans la domaine de la programmation procédurale, le langage Java devrait remplacer le langage C++ pour la partie procédurale de COP. Le passage du premier lanagage au deuxième ne devrait poser aucun problème particulier.

Pour la partie logique de COP, il faudrait doter le système de mécanismes de la progrmmation par contraintes qui représente également la tendance actuelle dans la domaine de la programmation logique.

Annexe A

Exemple complet

A.1 Fichier source COP

```
power(X,0,1).
power(X,N,V) :- N > 0,
                N1 is N - 1,
                power(X,N1,V1),
                V is X * V1.

main() {
  TBridgeVariable X,Y,Z;

  X.set(2,INTEGER);
  Y.set(10,INTEGER);
  Z.set("Z",FREE);
  if(power(X,Y,Z))
    cout<<Z<<"\n";
  return(0);
}
```

A.2 Fichier Bridge

```
main() {
  TBridgeVariable X,Y,Z;

  X.set(2,INTEGER);
  Y.set(10,INTEGER);
  Z.set("Z",FREE);
  if(power(X,Y,Z))
    cout<<Z<<"\n";
  return(0);
}
```

A.3 Fichier Prolog

```
power(X,0,1).
power(X,N,V) :- N > 0,
                N1 is N - 1,
                power(X,N1,V1),
                V is X * V1.
```

A.4 Fichier C++

```
main() {
  TBridgeVariable X,Y,Z;

  X.set(2,INTEGER);
  Y.set(10,INTEGER);
  Z.set("Z",FREE);
  if(power_03_00(X,Y,Z))
    cout<<Z<<"\n";
  return(0);
}
```

A.5 Fichier des bridge functions

```
extern TProlog prolog;

int power_03_00(TBridgeVariable& P1,
                TBridgeVariable& P2,
                TBridgeVariable& P3)
{
  char fonctor[10];
  TBridgeVariable *parmList[4];
  int returnValue;

  strcpy(fonctor, "power_00");

  parmList[0] = &P1;
  parmList[1] = &P2;
  parmList[2] = &P3;
  parmList[3] = NULL;

  prolog.setupGoal(fonctor, parmList);
  returnValue = prolog.run();
  if(returnValue) prolog.getVariables();
  return returnValue;
}
```

A.6 Fichier WAM

```
main/0 :
put_integer 2,A0
put_integer 10,A1
put_variable X2,A2
execute power/3

power/3 :
try_me_else 5
get_variable X0,A0
```

```
get_integer 0,A1
get_integer 1,A2
proceed
trust_me_else fail
allocate
get_variable Y4,A0
get_variable Y7,A1
get_variable Y2,A2
put_value Y7,A0
put_integer 0,A1
built_in gr_
put_value Y7,A0
put_integer 1,A1
put_variable Y6,A2
built_in sub_
put_variable Y5,A0
put_unsafe_value Y6,A1
built_in is
put_value Y4,A0
put_unsafe_value Y5,A1
put_variable Y3,A2
call power/3,4
put_value Y4,A0
put_unsafe_value Y3,A1
put_variable Y1,A2
built_in mul_
put_value Y2,A0
put_unsafe_value Y1,A1
deallocate
built_in is
proceed
```

A.7 Fichier Byte_Code WAM

```
struct instr_type codestore[] = {
{38, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{40, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
{41, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, -1},
{12, -1, -1, -1, -1, -1, -1, -1, 8388610, 0, -1},
{12, -1, -1, -1, -1, -1, -1, -1, 8388618, 1, -1},
{6, -1, -1, -1, -1, -1, 2, 2, -1, -1, -1},
{3, -1, 8, -1, -1, -1, -1, -1, -1, -1, -1},
{41, -1, -1, -1, -1, -1, -1, -1, -1, -1, 5, -1},
{34, -1, -1, -1, 13, -1, -1, -1, -1, -1, -1},
{16, -1, -1, -1, -1, -1, 0, 0, -1, -1, -1},
{21, -1, -1, -1, -1, -1, -1, -1, 8388608, 1, -1},
{21, -1, -1, -1, -1, -1, -1, -1, 8388609, 2, -1},
{4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{36, -1, -1, -1, 0, -1, -1, -1, -1, -1, -1},
{0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{15, -1, -1, -1, -1, -1, 4, 0, -1, -1, -1},
{15, -1, -1, -1, -1, -1, 7, 1, -1, -1, -1},
{15, -1, -1, -1, -1, -1, 2, 2, -1, -1, -1},
{7, -1, -1, -1, -1, -1, 7, 0, -1, -1, -1},
{12, -1, -1, -1, -1, -1, -1, -1, 8388608, 1, -1},
{37, -1, -1, -1, -1, -1, -1, -1, -1, -1, 14},
{7, -1, -1, -1, -1, -1, 7, 0, -1, -1, -1},
{12, -1, -1, -1, -1, -1, -1, -1, 8388609, 1, -1},
{5, -1, -1, -1, -1, -1, 6, 2, -1, -1, -1},
{37, -1, -1, -1, -1, -1, -1, -1, -1, -1, 16},
{5, -1, -1, -1, -1, -1, 5, 0, -1, -1, -1},
{9, -1, -1, -1, -1, -1, 6, 1, -1, -1, -1},
{37, -1, -1, -1, -1, -1, -1, -1, -1, -1, 19},
{7, -1, -1, -1, -1, -1, 4, 0, -1, -1, -1},
{9, -1, -1, -1, -1, -1, 5, 1, -1, -1, -1},
{5, -1, -1, -1, -1, -1, 3, 2, -1, -1, -1},
{2, -1, 8, 4, -1, -1, -1, -1, -1, -1, -1},
{7, -1, -1, -1, -1, -1, 4, 0, -1, -1, -1},
{9, -1, -1, -1, -1, -1, 3, 1, -1, -1, -1},
{5, -1, -1, -1, -1, -1, 1, 2, -1, -1, -1},
{37, -1, -1, -1, -1, -1, -1, -1, -1, -1, 17},
{7, -1, -1, -1, -1, -1, 2, 0, -1, -1, -1},
{9, -1, -1, -1, -1, -1, 1, 1, -1, -1, -1},
{1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{37, -1, -1, -1, -1, -1, -1, -1, -1, -1, 19},
```

```
{4, -1, -1, -1, -1, -1, -1, -1, -1, -1}
};
```

A.8 Résultat de l'exécution

```
1024
```

Annexe B

Méthodes de la classe TProlog

1. `int setupGoal(TBridgeVariable *list[], char *fonctor)`

   Cette méthode a deux paramètres. Le premier paramètre est une liste de pointeurs à des bridge-variables qui représente la liste des paramètres du but à exécuter. La fin de la liste est indiquée par un pointeur nul. Cette liste est bâtie avec les bridge-variables passées en paramètres à la bridge-function. Le deuxième paramètre est le nom du prédicat à exécuter.

2. `int run()`

   Cette méthode exécute le but spécifié par `setupGoal`. Un appel à cette méthode tente de satisfaire le but et indique un échec par une valeur de retour de zéro ou génère une seule solution et a une valeur de retour différente de zéro. Les points de choix, s'ils existent, sont conservés dans la machine WAM en cas de retour arrière sur des appels successifs de cette même bridge-function. Par contre, les points de choix des appels précédents sont détruits s'ils ne sont pas issus de la même bridge-function.

3. `int getVariables()`

   Cette méthode fait la conversion des résultats obtenus par la machine WAM dans les variables paramètres au bridge-goal. Elle transforme donc les résultats dans la représentation de la machine WAM en représentation des bridge-variables.

4. `int reset()`

   Cette méthode détruit tous les points de choix et s'assure que la machine WAM est prête pour traiter un autre but. Cette méthode est la seule de la classe `TProlog` qui peut être appelée par le programmeur. Elle peut être nécessaire au programmeur lorsqu'il veut explicitement détruire les points de choix par nécessité ou par assurance.

Toutes ces méthodes ont zéro comme valeur de retour en cas d'échec et une valeur de retour différente de zéro en cas de réussite.

Annexe C

Méthodes de la classe TBridgeVariable

`enum TBridgeType {EMPTY, FREE, ATOM, INTEGER, STRING, LIST, TERM}`

1. Méthodes d'écriture de valeurs et de types
   - `int set(int valeur, TBridgeType type)`
   - `int set(char *valeur, TBridgeType type)`
   - `int set(TBridgeVariable var)`
2. Méthodes d'écriture de valeurs
   - `int setValue(int valeur)`
   - `int setValue(char *valeur)`

3. Méthode d'écriture de types

   - `int setType(int TBridgeType type)`

4. Méthodes de lecture de valeurs et de types

   - `int get(int *valeur, TBridgeType *type)`
   - `int get(char *valeur, TBridgeType *type)`

5. Méthodes de lecture de valeurs

   - `int getValue(int *valeur)`
   - `int getValue(char *valeur)`

6. Méthode de lecture de types

   - `int getType(int TBridgeType *type)`

Annexe D

Programmes de test

D.1 Le programme boresea

- Code Prolog:

```
main :-  lips1.
lips1 :- lips2.
lips2 :- lips3.
lips3 :- lips4.
lips4 :- lips5.
lips5 :- lips6.
lips6 :- lips7.
lips7 :- lips8.
lips8 :- lips9.
lips9 :- lips10.
...
lips190 :- lips191.
lips191 :- lips192.
lips192 :- lips193.
lips193 :- lips194.
lips194 :- lips195.
lips195 :- lips196.
lips196 :- lips197.
lips197 :- lips198.
lips198 :- lips199.
lips199 :- lips200.
lips200.
```

- Code WAM:

```
main/0 :
execute lips1/0
lips1/0 :
execute lips2/0
lips2/0 :
execute lips3/0
lips3/0 :
execute lips4/0
lips4/0 :
execute lips5/0
lips5/0 :
execute lips6/0
lips6/0 :
execute lips7/0
lips7/0 :
execute lips8/0
lips8/0 :
execute lips9/0
lips9/0 :
execute lips10/0
....
lips190/0 :
execute lips191/0
lips191/0 :
execute lips192/0
lips192/0 :
execute lips193/0
lips193/0 :
execute lips194/0
lips194/0 :
execute lips195/0
lips195/0 :
execute lips196/0
lips196/0 :
execute lips197/0
lips197/0 :
execute lips198/0
lips198/0 :
execute lips199/0
lips199/0 :
execute lips200/0
lips200/0 :
proceed
```

D.2 Le programme cre_env

- Code Prolog:

```
main :- env0(X,Y,Z).
env0(X,Y,Z):-env1(Z,X,Y),env2(Y,Z,X).
env1(X,Y,Z):-env3(Z,Y,X),env4(Y,Z,X).
env2(X,Y,Z):-env3(Z,Y,X),env4(Y,Z,X).
env3(X,Y,Z):-env5(Z,Y,X),env6(Y,Z,X).
env4(X,Y,Z):-env5(Z,Y,X),env6(Y,Z,X).
env5(X,Y,Z):-env7(Z,Y,X),env8(Y,Z,X).
env6(X,Y,Z):-env7(Z,Y,X),env8(Y,Z,X).
env7(X,Y,Z):-env9(Z,Y,X),env10(Y,Z,X).
env8(X,Y,Z):-env9(Z,Y,X),env10(Y,Z,X).
env9(X,Y,Z):-env11(Z,Y,X),env12(Y,Z,X).
env10(X,Y,Z):-env12(Z,Y,X),env12(Y,Z,X).
env11(X,Y,Z):-env12(Z,Y,X),env12(Y,Z,X).
env12(X,Y,Z).
```

- Code WAM:

```
main/0 :
put_variable X0,A0
put_variable X1,A1
put_variable X2,A2
execute env0/3
env0/3 :
allocate
get_variable Y1,A0
get_variable Y3,A1
get_variable Y2,A2
put_value Y2,A0
put_value Y1,A1
put_value Y3,A2
call env1/3,3
put_value Y3,A0
put_value Y2,A1
put_value Y1,A2
deallocate
execute env2/3
...
env11/3 :
allocate
get_variable Y1,A0
get_variable Y3,A1
get_variable Y2,A2
put_value Y2,A0
put_value Y3,A1
put_value Y1,A2
call env12/3,3
put_value Y3,A0
```

```
put_value Y2,A1
put_value Y1,A2
deallocate
execute env12/3
env12/3 :
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
proceed
```

D.3 Le programme general_unif

- Code Prolog:

```
main :- nested_structure1(A),
        nested_structure2(B),
        unify(A,B).
unify(X,X).
nested_structure1(
[a([a1([1,2,3],a),a2([4,5,6],b),a3([7,8,9],c)])]).
nested_structure2(
[a([a1([1,2,3],a),a2([4,5,6],b),a3([7,8,9],c)])]).
```

- Code WAM:

```
main/0 :
allocate
put_variable Y2,A0
call nested_structure1/1,2
put_variable Y1,A0
call nested_structure2/1,2
put_unsafe_value Y2,A0
put_unsafe_value Y1,A1
deallocate
execute unify/2
unify/2 :
get_variable X0,A0
get_value X0,A1
proceed
nested_structure1/1 :
get_list A0
unify_variable X0
put_value X0,A1
unify_nil
get_structure a/1,A1
unify_variable X0
put_value X0,A2
get_list A2
```

```
unify_variable X0
put_value X0,A3
unify_variable X0
put_value X0,A4
get_structure a1/2,A3
unify_variable X0
put_value X0,A5
unify_const a
get_list A5
unify_integer 1
unify_variable X1
put_value X1,A6
get_list A6
unify_integer 2
unify_variable X2
put_value X2,A7
get_list A7
unify_integer 3
unify_nil
get_list A4
unify_value X0
put_value X0,A5
unify_value X1
put_value X1,A6
get_structure a2/2,A5
unify_value X2
put_value X2,A7
unify_const b
get_list A7
unify_integer 4
unify_variable X0
put_value X0,A8
get_list A8
unify_integer 5
unify_variable X1
put_value X1,A9
get_list A9
unify_integer 6
unify_nil
get_list A6
unify_value X2
put_value X2,A7
unify_nil
get_structure a3/2,A7
unify_value X0
put_value X0,A8
unify_const c
get_list A8
unify_integer 7
unify_value X1
```

```
put_value X1,A9
get_list A9
unify_integer 8
unify_variable X0
put_value X0,A10
get_list A10
unify_integer 9
unify_nil
proceed
nested_structure2/1 :
get_list A0
unify_variable X0
put_value X0,A1
unify_nil
...
get_list A10
unify_integer 9
unify_nil
proceed
```

D.4 Le programme choice_point

- Code Prolog:

```
main :- ccp1(0,0,0).
ccp1(X,Y,Z):-ccp2(X,Y,Z).
ccp1(X,Y,Z).
ccp2(X,Y,Z):-ccp3(X,Y,Z).
ccp2(X,Y,Z).
ccp3(X,Y,Z):-ccp4(X,Y,Z).
ccp3(X,Y,Z).
ccp4(X,Y,Z):-ccp5(X,Y,Z).
ccp4(X,Y,Z).
ccp5(X,Y,Z):-ccp6(X,Y,Z).
ccp5(X,Y,Z).
ccp6(X,Y,Z):-ccp7(X,Y,Z).
ccp6(X,Y,Z).
ccp7(X,Y,Z):-ccp8(X,Y,Z).
ccp7(X,Y,Z).
ccp8(X,Y,Z):-ccp9(X,Y,Z).
ccp8(X,Y,Z).
ccp9(X,Y,Z):-ccp10(X,Y,Z).
ccp9(X,Y,Z).
ccp10(X,Y,Z).
ccp10(X,Y,Z).
```

- Code WAM:

```
main/0 :
put_integer 0,A0
put_integer 0,A1
put_integer 0,A2
execute ccp1/3
ccp1/3 :
try_me_else 8
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
put_value X0,A0
put_value X1,A1
put_value X2,A2
execute ccp2/3
trust_me_else fail
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
proceed
...
ccp10/3 :
try_me_else 5
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
proceed
trust_me_else fail
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
proceed
```

D.5 Le programme deep_back

- Code Prolog:

```
main :- pd(X1,X2,X3).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
```

```
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3) :- q(X1,X2,a).
pd(X1,X2,X3).
q(X1,X2,b).
```

- Code WAM:

```
main/0 :
put_variable X0,A0
put_variable X1,A1
put_variable X2,A2
execute pd/3
pd/3 :
try_me_else 9
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
put_value X0,A0
put_value X1,A1
put_value X2,A2
put_const a,A2
execute q/3
retry_me_else 9
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
put_value X0,A0
put_value X1,A1
put_value X2,A2
put_const a,A2
...
trust_me_else fail
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
proceed
q/3 :
get_variable X0,A0
get_variable X1,A1
get_const b,A2
proceed
```

D.6 Le programme shallow_back

- Code Prolog:

```
main :- :- ps(X1,X2,X3).
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3) :- fail.
ps(X1,X2,X3).
```

- Code WAM:

```
main/0 :
put_variable X0,A0
put_variable X1,A1
put_variable X2,A2
execute ps/3
ps/3 :
try_me_else 6
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
built_in fail
proceed
retry_me_else 6
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
built_in fail
proceed
...
```

```
trust_me_else fail
get_variable X0,A0
get_variable X1,A1
get_variable X2,A2
proceed
```

D.7 Le programme nrev

- Code Prolog:

```
main :- nrev([a,b,c,d,e,f,g,h,i,j], R).
nrev([], []).
nrev([H | T], R) :-
  nrev(T, TR),
  append(TR, [H], R).
append([], L,L).
append([H | T], L, [H | TL]) :- append(T, L, TL).
```

- Code WAM:

```
main/0 :
put_list A10
unify_const j
unify_nil
put_list A9
unify_const i
get_variable X10,A10
unify_value X10
put_list A8
unify_const h
get_variable X9,A9
unify_value X9
put_list A7
unify_const g
get_variable X8,A8
unify_value X8
put_list A6
unify_const f
get_variable X7,A7
unify_value X7
put_list A5
unify_const e
get_variable X6,A6
unify_value X6
put_list A4
unify_const d
get_variable X5,A5
```

```
unify_value X5
put_list A3
unify_const c
get_variable X4,A4
unify_value X4
put_list A2
unify_const b
get_variable X3,A3
unify_value X3
put_list A0
unify_const a
get_variable X2,A2
unify_value X2
put_variable X1,A1
execute nrev/2
nrev/2 :
try_me_else 4
get_nil A0
get_nil A1
proceed
trust_me_else fail
get_list A0
allocate
unify_variable Y2
unify_variable X0
get_variable Y1,A1
put_value X0,A0
put_variable Y3,A1
call nrev/2,3
put_unsafe_value Y3,A0
put_list A1
unify_value Y2
unify_nil
put_value Y1,A2
deallocate
execute append/3
append/3 :
try_me_else 5
get_nil A0
get_variable X1,A1
get_value X1,A2
proceed
trust_me_else fail
get_list A0
unify_variable X0
unify_variable X3
get_variable X1,A1
get_list A2
unify_value X0
unify_variable X0
get_variable X2,A0
```

```
put_value X3,A0
put_value X1,A1
put_value X2,A2
execute append/3
```

D.8 Le programme fib

- Code Prolog:

```
main :- fib(5,Z).
fib(0,1).
fib(1,1).
fib(X,Y):- X1 is X-1,
           X2 is X-2,
           fib(X1,Y1),
           fib(X2,Y2),
           Y is Y1+Y2.
```

- Code WAM:

```
main/0 :
put_integer 5,A0
put_variable X1,A1
execute fib/2
fib/2 :
try_me_else 4
get_integer 0,A0
get_integer 1,A1
proceed
retry_me_else 4
get_integer 1,A0
get_integer 1,A1
proceed
trust_me_else fail
allocate
get_variable Y8,A0
get_variable Y2,A1
put_value Y8,A0
put_integer 1,A1
put_variable Y9,A2
built_in sub_
put_variable Y6,A0
put_unsafe_value Y9,A1
built_in is
put_value Y8,A0
put_integer 2,A1
put_variable Y7,A2
built_in sub_
```

```
put_variable Y5,A0
put_unsafe_value Y7,A1
built_in is
put_unsafe_value Y6,A0
put_variable Y4,A1
call fib/2,5
put_unsafe_value Y5,A0
put_variable Y3,A1
call fib/2,4
put_unsafe_value Y4,A0
put_unsafe_value Y3,A1
put_variable Y1,A2
built_in add_
put_value Y2,A0
put_unsafe_value Y1,A1
deallocate
built_in is
proceed
```

BIBLIOGRAPHIE

[1] BRUNET, C.-A. *Contribution à une intégration de la programmation procédurale et de la programmation logique (le langage COP).* Mémoire de maîtrise es sciences appliquées, Université de Sherbrooke, Sherbrooke, mars 1994. 104 p.

[2] STROUSTRUP, B. *The C++ Programming Language.* Addison–Wesley, New York, 1991. 669 p.

[3] KERNIGHAN, B., RITCHIE, D. *The C Programming Language.* Prentice–Hall, Englewood Cliffs, 1978. 228 p.

[4] COLMERAUER, A., KANOUI, H., PASERO, R., ROUSSEL, P. *Un système de communication homme-machine en français.* Rapport technique, G.I.A Université d'Aix–MarseilleII, 1972.

[5] CLOCKSIN, W.F., MELLISH, C.S. *Programming in Prolog.* Springer–Verlag, Berlin, 1981. 279 p.

[6] LLOYD, J.W. *Foundations of Logic Programming.* Springer–Verlag, Berlin, 1987. 212 p.

[7] AHO, A.V., SETHI, R., ULLMAN, J.D. *Compilers: Principles, Techniques and Tools.* Addison–Wesley, New York, 1986. 796 p.

[8] VAN CANEGHEM, M. *L'anatomie de Prolog.* InterEditions, Paris, 1986. 191 p.

[9] GUDEMAN, D. *Representing Type Information in Dynamically Typed Languages.* Technical Report 93-27, University of Arizona, Arizona, octobre 1993. 38 p. `ftp://ftp.cs.arizona.edu/reports/1993/TR93-27.ps`.

[10] WEISS, M.A. *Algorithms, Data Structures and Problem Solving with C++.* Addison–Wesley, New York, 1995. 820 p.

[11] PEREIRA, F. *C–Prolog User's Manual.* Menlo Park, 1995. 35 p. `ftp://gate.ee.lsu.edu/pub/koppel/ee4785/prolog.ps`.

[12] BOYD, J.L., KARAM, G.M. *Prolog in C.* Technical Report, Carleton University, mars 1988.

[13] WEINER, J.L., RAMAKRISHNAN, S. *A Piggy-Back Compiler for Prolog.* In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 288–296, Atlanta, 1988.

[14] WARREN, D.H.D. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, Menlo Park, octobre 1983. 30 p.

[15] DEDKOV, A.F., EADLINE, D.J. *Design and Implementation of a Prolog-to-C Compiler.* In *International Logic Programming Symposium*, Ithaca, 1994. `ftp://ftp.elis.rug.ac.be/pub/prolog/ilps94_workshop/dedkov.ps.Z`.

[16] LEVY, M.R., HORSPOOL, R.N. *C as a Target for WAM-based Prolog compilation.* In *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing*, pages 131–134, Victoria, 1995.

[17] TURK, A.K. *Compiler Optimizations for the WAM.* In *3rd International Conference on Logic Programming*, pages 657–662, London, 1986.

[18] OLDER, W.J., RUMMEL, J.A. *An Incremental Garbage Collector for WAM-Based Prolog.* In *Joint International Conference and Symposium on Logic Programming*, pages 369–383, Washington, 1992.

[19] AÏT-KACI, H. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, 1991. 114 p.

[20] BOIZUMAULT, P. *Prolog – L'implantation.* Masson, Paris, 1988. 303 p.

[21] DIAZ, D. *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: Le système clp(FD).* Thèse de doctorat en informatique, Université d'Orléans, janvier 1995. 270p.
`ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/publications/diaz-french-thesis.ps.gz`.

[22] SUN Microsystems. *LEX – a Lexical Analyzer Generator.* In *Programming Utilities and Librairies*, chapter 9. Sun Microsystems, Mountain View, 1990.

[23] SUN Microsystems. *YACC – Yet Another Compiler Compiler.* In *Programming Utilities and Librairies*, chapter 10. Sun Microsystems, Mountain View, 1990.

[24] SYRE, J.C. *Benchmark Programs for Prolog Systems.* European Computer Industry Research Center, Munich, 1988.

[25] MARIËN, A., DEMOEN, B. *A new Scheme for Unification in WAM.* In *International Symposium on Logic Programming*, pages 257–271, San Diego, 1991.

[26] CODOGNET, P., SOLA, T. *Extending the WAM for Intelligent Backtracking.* In *8th International Conference on Logic Programming*, pages 127–141, Paris, 1991.

[27] LEVY, M.R., HORSPOOL, R.N. *Translating Prolog to C: a WAM-based Approach.* In *2nd Compulog Network Area Meeting on Programming Languages*, Pisa, 1993. `ftp://csr.uvic.ca/pub/Publications/Horspool/prolog2c.ps.gz`.

[28] CODOGNET, P., DIAZ, D. *wamcc: Compiling Prolog to C.* In *12th International Conference on Logic Programming*, pages 317–331, Tokyo, 1995. `ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/publications/wamcc.ps`.

[29] DIAZ, D. *wamcc 2.21 User's Manual*, juillet 1994. 27 p. `ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/wamcc/wamcc2.21.tar.Z`.

[30] DEMOEN, B., MARIS, G. *A Comparison of some Schemes for Translating Logic to C.* In *11th International Conference on Logic Programming*, Santa Margherita, 1994. `ftp://ftp.csd.uu.se/pub/papers/reports/0078/7-demoen+maris.ps.gz`.

[31] ZHOU, N.-F. *Global Optimizations in a Prolog Compiler for the TOAM.* In *The Journal of Logic Programming*, pages 275–294, New York, 1993.

[32] MATYSKA, L., JERGOVÁ, A., TOMAN, D. *Register Allocation in WAM.* In *8th International Conference on Logic Programming*, pages 142–156, Paris, 1991.

[33] BEVEMYR, J. *The Luther WAM Emulator.* Technical Report 72, Uppsala University, mars 1992. 120 p. `ftp://ftp.csd.uu.se/pub/papers/reports/0072.ps.gz`.

[34] GUDEMAN, K., DE BOSSCHERE, K., DEBRAY, S. *jc: An Efficient and Portable Sequential Implementation of Janus.* In *Joint International Conference and Symposium on Logic Programming*, pages 399–413, Washington, 1992.

[35] HAUSSMAN, B. *Turbo Erlang: Approaching the Speed of C.* In *Implementations of Logic Programming Systems.* Kluwer, Deventer, 1994.

[36] TARAU, P., DEMEON, B., DE BOSSCHERE, K., DEBRAY, S. *The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog.* In *ACM Symposium on Applied Computing*, pages 152–176, Nashville, 1995.
`ftp://dia.fi.upm.es/pub/papers/COMPULOG/94Area_meeting/partrans.ps.Z`.

[37] VAN ROY, P. *Can Logic Programming Execute as Fast as Imperative Programming?* Ph.D. Thesis, University of California, Berkeley, 1990. 225 p.
`http://www.info.ucl.ac.be/people/PVR/Peter.thesis/Peter.thesis.html`.

[38] VAN ROY, P. *1983–1993: The Wonder Years of Sequential Prolog.* Report 36, Digital Equipment Corporation, Paris, décembre 1993. 69 p.
`http://www.info.ucl.ac.be/people/PVR/official_report.ps`.

[39] VAN ROY, P. *Issues in Implementing Constraint Logic Languages.* École de Printemps, Châtillon–sur–Seine, 1994.
`http://www.info.ucl.ac.be/people/PVR/impltalk.ps`.