Université Catholique de Louvain

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique

Étude de l'interopérabilité de deux langages de programmation basée sur la machine virtuelle de Java

Promoteur : Baudouin Le Charlier

Lecteurs : Gustavo Ospina, Kim Mens

Mémoire présenté en vue
de l'obtention du Grade de
diplômé d'études spécialisées
en sciences appliquées par
Frédéric Minne.

Louvain-la-Neuve
Année académique 2002-2003

Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie). [Dave Small]

En informatique, il n'existe aucun problème qui ne puisse être résolu en ajoutant suffisamment de degrés d'indirection. [Anonyme]

Je tiens tout d'abord à remercier mon promoteur, Baudouin Le Charlier, ainsi que Gustavo Ospina pour leurs conseils avisés et leurs critiques qui m'ont permis de mener à bien ce travail.

Je remercie Claudine et Valérie pour leur relecture patiente et attentive du texte de ce mémoire.

Un grand merci également à Philippe pour son aide et à toute l'équipe du Kot Astro pour leur soutient.

Merci à Partick Alin pour le compte internet et à toute l'équipe d'ACM Student Chapter de Louvain-la-Neuve.

Enfin, je tiens à remercier ma famille pour leur aide et leur soutien.

à Valérie...

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Dans cette introduction, je vais définir la notion d'interopérabilité entre deux langages de programmation et donner les motivations pour sa réalisation. Je présenterai ensuite quelques mécanismes d'interopérabilité et je décrirai celui que nous allons développer dans ce mémoire. Mais dans un premier temps, je vais indiquer l'organisation du reste de ce texte.

1.1 Organisation de ce mémoire

Ce mémoire se décompose en sept parties :

- Introduction : Le Chapitre 1 donne un aperçu du problème de l'interopérabilité de deux langages de programmation, en général, et de Java et Prolog, en particulier. Le Chapitre 2 introduit les concepts de base du langage Prolog, concepts qui seront utiles pour la suite de ce mémoire.
- Partie I : Les Chapitres 3 et 4 sont une petite discussion préliminaire à la description du mécanisme d'interopérabilité entre Java et Prolog basé sur la machine virtuelle de Java. Le Chapitre 3 s'intéresse aux différences entre la machine virtuelle de Java et une machine Prolog particulière, la machine de Warren, et se penche sur les problèmes de la gestion de la mémoire entre les deux. Le Chapitre 4 se penche sur le problème de la conversion des données entre programme Java et programme Prolog.
- Partie II : Les Chapitres 5 à 8 décrivent et spécifient ce mécanisme d'interopérabilité. Après les remarques générales du Chapitre 5, les trois chapitres suivants vont s'intéresser à la spécification des trois aspects d'un mécanisme d'interopérabilité entre Java et Prolog basé sur le machine virtuelle de Java. Le Chapitre 6 se penche sur la spécification des classes et méthodes permettant la conversion des données entre Java et Prolog. Le Chapitre 7 s'intéresse au mécanisme d'interopérabilité permettant l'appel à un programme Prolog depuis un programme Java. Quant au Chapitre 8, il s'intéresse au mécanisme permettant l'interopérabilité de Prolog vers Java, c'est-à-dire l'invocation de méthodes Java depuis un programme Prolog.
- Partie III : Les Chapitres 9 à 12 se concentrent sur les problèmes et choix d'implémentation des trois parties du mécanisme d'interopérabilité. Le Chapitre 9 donne quelques choix généraux pour l'implémentation. Quant aux Chapitres 10, 11 et 12, ils se concentrent sur les choix et problèmes propres à l'implémentation de chacune des trois parties du mécanisme d'interopérabilité.
- Partie IV : Les Chapitres 13 et 14 donnent quelques exemples d'applications du mécanisme d'interopérabilité défini dans les chapitres précédents. Le Chapitre 13 donne

quelques scénarios d'utilisation très généraux du mécanisme d'interopérabilité. Le Chapitre 14 décrit un exemple complet d'application du mécanisme d'interopérabilité de Java vers Prolog : un interpréteur Prolog simple.

- Partie V : Les Chapitres 15 à 16 terminent ce mémoire en donnant les conclusions et quelques perspectives et améliorations envisageables pour le mécanisme d'interopérabilité présenté dans les chapitres précédents.
- Annexes : en annexes se trouvent les spécifications complètes des méthodes du mécanisme d'interopérabilité entre Java et Prolog définies dans la partie II, une table des implémentations de Prolog permettant l'interopérabilité avec Java et précisant, pour chacune, la technique d'interopérabilité utilisée, et, enfin, le code source complet de l'exemple de l'interpréteur présenté au Chapitre 14.

Ce mémoire est accompagné d'un CD-Rom contenant le code source et l'API du mécanisme d'interopérabilité décrit dans ce mémoire, ainsi qu'une version électronique de ce texte. Le contenu de ce CD-Rom, ainsi que certains textes de référence, sont également disponibles en ligne sur le site internet consacré à ce mémoire : *http://blc-interop.be.tf*. De plus amples informations sur les langages de programmation compilés pour la machine virtuelle de Java peuvent être trouvées sur le cite internet *Programming Languages for the Java Virtual Machine*, à l'adresse *http://grunge.cs.tu-berlin.de/ tolk/vmlanguages.html*.

La table ci-dessous donne quelques abréviations qui seront utilisées dans ce mémoire.

Abréviation	Signification
API	Application Programming Interface
JVM	Java Virtual Machine
WAM	Warren's Abstract Machine
JNI	Java Native Interface

1.2 Une définition de l'interopérabilité de deux langages de programmation

Avant de présenter les motivations et les mécanismes permettant l'interopérabilité entre deux langages de programmation, je pense qu'il est bon de nous mettre d'accord sur ce que j'entends par *interopérabilité*.

L'interopérabilité entre deux langages de programmation est la capacité d'un programme (ou d'un fragment de programme) écrit dans un langage de programmation X d'utiliser un programme (ou un fragment de programme) écrit dans un langage de programmation Y différent de X.

Nous dirons qu'un mécanisme d'interopérabilité entre deux langages de programmation X et Y est *transparent* si le programmeur ne doit pas tenir compte des détails *intimes* du langage Y (tels que son implémentation, les structures de données utilisées...) pour pouvoir le faire interagir avec le langage X.

Un exemple de détail *intime* d'un langage de programmation est l'implémentation de son interpréteur ou la représentation de ses structures de données en mémoire.

1.3 Motivation

1.3.1 Pourquoi vouloir faire interagir deux langages de programmation ?

Il existe aujourd'hui un très grand nombre de langages de programmation. Certains d'entre eux sont plus ou moins spécialisés et adaptés pour résoudre différentes classes de problèmes.

Certains langages, comme COBOL, sont plus indiqués pour les problèmes de gestion et de finance, d'autres, comme FORTRAN ou ADA, pour les logiciels scientifiques et rigoureux, d'autres encore, comme Delphi ou Visual Basic, sont plus orientés vers le développement d'applications de bureautique, d'autres, comme C, sont des langages adaptés pour la programmation de fonctions de bas niveaux, d'autres, enfin, comme C# ou Java, permettent le développement simple d'applications réseau ou celui d'applications s'exécutant dans un environnement protégé (par exemple les applets Java).

Le problème de l'interopérabilité de deux langages de programmation est un problème majeur lors de la fusion de deux entreprises. En effet, un objectif lors de ces fucions est de pouvoir conserver les différentes applications de chaque entreprise et, par conséquent, de les faires communiquer entre elles. Ces différents systèmes d'information sont souvent implémentés dans des langages différents qu'il faudra donc interfacer [BBE98].

Devant la grande popularité d'internet et des applications distribuées via les browsers[1], il peut être nécessaires de pouvoir intégrer d'anciennes applications écrites dans des langages différents au sein d'une même application basée sur la technologie d'internet.

Une autre raison pour s'intéresser à l'interopérabilité est l'idée que, puisque de nombreux langages sont conçus pour des domaines assez spécialisés, il pourrait être utile d'implémenter les différentes parties d'une application dans le langage le plus adéquat et de faire ensuite interagir ces différentes parties entre elles.

1.3.2 Interopérabilité entre Java et Prolog

Dans le cadre de cet article, nous allons nous intéresser au cas particulier de l'interopérabilité entre le langage Java et le langage Prolog. Mais pourquoi nous intéresser à ces deux langages en particulier ? Pour répondre à cette question, nous allons examiner les différentes possibilités offertes par ces deux langages et voir les avantages qu'il peut y avoir à les utiliser ensemble.

Pourquoi Prolog ?

Le langage Prolog est un langage logique, de haut niveau, utilisé pour les applications d'intelligence artificielle telles que les systèmes experts, le traitement de la langue naturelle [TD01] [Tha91], pour les bases de données relationnelles, le datamining, et bien d'autres applications faisant intervenir des problèmes de parcours d'arbres de résolution.

Il permet de programmer de manière simple des problèmes souvent non déterministes qui peuvent être représentés sous forme d'un ensemble de règles et de faits, et dont la résolution se fait en parcourant un arbre de solutions. En Prolog, il n'est pas nécessaire, comme c'est le cas dans les langages impératifs, de construire la résolution explicite d'un problème. Il suffit de modéliser une base de connaissances[2] qui sera utilisée par un moteur d'inférence pour construire les solutions à un problème donné (Voir figure 1.1) [SS97].

Les domaines dans lesquels Prolog est particulièrement adapté sont, entre autres, l'analyse syntaxique, les problèmes de recherche exhaustive avec heuristique (Huit reines, Tour de Hanoï...), le traitement de la langue naturelle, les parcours de graphes, les systèmes experts, les logiciels adaptatifs...

Malgré tous ses avantages pour la description de systèmes complexes ou de problèmes non-déterministes en terme de règles d'inférences et d'axiomes logiques, Prolog est, pour diverses raisons que nous allons quelque peu détailler, peu adéquat pour développer des applications orientées utilisateurs ou des services Web.

En effet, l'intégration d'un interpréteur Prolog dans un serveur web ou un browser est parfois difficile à cause du manque d'interfaces entre Prolog et les autres langages et de l'implémentation native de l'interpréteur. Ensuite, de nombreuses implémentations de Prolog ne fournissent pas

[1] Par exemple des applications pour l'intranet et les groupwares

[2] Un ensemble de règles et de faits.

de fonctions prédéfinies pour construire une interface graphique utilisateur, ce qui rend difficile la conception de programmes interactifs "user-friendly". Ajoutons à cela que le langage Prolog n'est pas bien adapté pour la réalisation d'applications réseau.

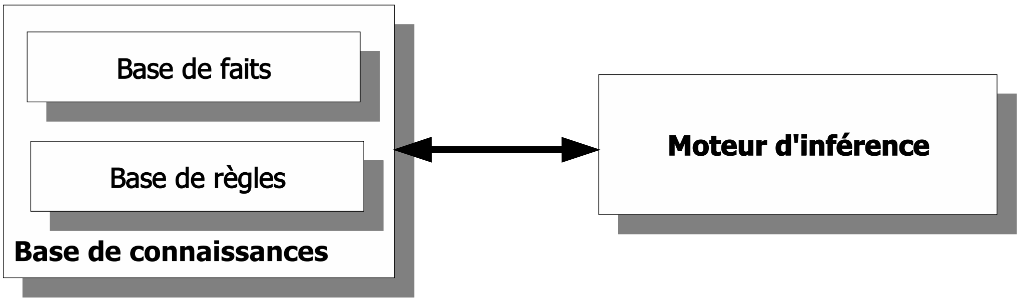

FIG. 1.1 – Architecture de Prolog

Pourquoi Java ?

Java est un langage orienté objet devenu très populaire en raison de sa simplicité de mise en oeuvre, de sa portabilité et de son intégration simple dans des applications orientées Web comme les applets et servlets. Java intègre la plupart des grandes fonctions des langages modernes : multithreading, interfaces graphiques, programmation réseau, garbage collection... Il possède également la propriété de faire s'exécuter certains programmes dans un environnement sécurisé, configurable grâce à un gestionnaire de sécurité très flexible.

Un autre avantage est que le développement d'applications en Java est grandement facilité par l'encapsulation des appels à des fonctions de bas niveaux dans les classes des bibliothèques Java telles que AWT (Abstract Window Toolkit), SWT (le toolkit graphique de IBM), java.net...

Java possède des bibliothèques pour la cryptographie (JCE - Java Cryptography Extension) et de connexion sécurisée (JSSL - Java Secure Socket Extension), l'utilisation d'objet répartis et l'invocation de méthodes à distance comme RMI, SOAP, JavaORB (un implémentation de CORBA) ou encore JavaRPC, l'accès aux bases de données relationnelles (JDBC - Java DataBase Connectivity), les technologies liées à XML... Toutes ces bibliothèques permettent le développement relativement simple d'applications réparties, de services Web...

En ce qui concerne l'interopérabilité avec d'autres langages, Java fournit une interface pour l'invocation de méthodes natives écrites dans d'autres langages tels que C ou C++ et possède de nombreuses bibliothèques performantes pour le traitement des documents XML.

Applications

L'interopérabilité entre Java et Prolog va nous permettre de tirer parti des avantages de ces deux langages (voir figure 1.2) et, ainsi, de réaliser des applications combinant les techniques de l'intelligence artificielle avec les attraits des applications réparties, des services web et des interfaces utilisateur graphiques.

On pourrait, par exemple, réaliser des outils de recherche sur le Web plus efficaces, des systèmes experts accessibles en ligne, des logiciels de construction et de vérification de programmes, des jeux intégrant une intelligence artificielle évoluée, des applications orientées agents, des smart bots guidant les visiteurs sur un site web et répondant à leurs questions...

1.3.3 Un exemple d'application réelle de l'interopérabilité entre Prolog et C++

Avant de décrire les différents mécanismes d'interopérabilité que j'ai envisagés, voici un exemple réel illustrant l'utilisation de Prolog au sein d'un autre langage de programmation. Il s'agit, en

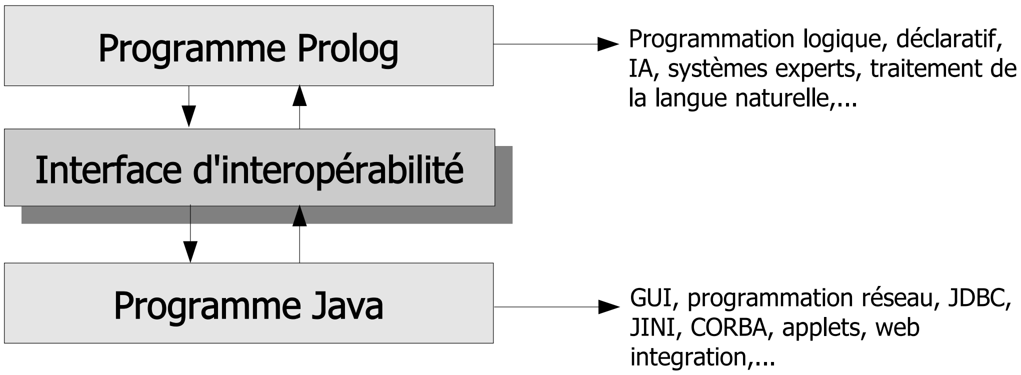

FIG. 1.2 – Interopérabilité entre Java et Prolog

l'occurrence, de l'interopérabilité entre les langages Prolog et C++ pour la configuration des interfaces réseaux dans Windows NT [Hov95].

L'objectif était d'obtenir un réseau dont les composants pourraient être chargés dynamiquement à la demande, sans devoir arrêter le système d'exploitation. Le problème qui se posait aux développeurs était qu'il n'y avait pas d'approche standardisée ni d'outils pour décrire la configuration d'un réseau.

L'idée va être de choisir un algorithme centralisé pour configurer le réseau, ce qui implique une approche déclarative dans laquelle, pour chaque composant, un fabriquant fournira un ensemble d'informations qui sera ajouté à une base de donnée de configuration, la base de registre. De plus, chaque composant devrait respecter les normes NDSI (Network Driver Interface Specification) et TDI (Transport Driver Interface)[3] pour garantir le comportement standard des composants réseau. Ces composants pourront alors être vus comme des boîtes noires caractérisées par leur interface d'entrée et leur interface de sortie.

Il reste maintenant à construire un maillage entre ces différents composants. Le programme de configuration devra tester les différentes liaisons possibles entre ces boîtes noires et choisir la bonne configuration. Pour simplifier, l'idée est de créer un maillage à partir d'un ensemble de noeuds et des règles permettant de les relier.

Pour réaliser cela, le choix s'est porté sur l'utilisation d'un interpréteur Prolog. Les informations de configuration contenues dans la base de registre vont être transformées en faits Prolog tandis que les règles pour lier les composants constitueront le programme. L'interopérabilité va être ici assurée par une redirection des entrées et sorties standards de C et un interpréteur Prolog léger écrit en C, le tout emballé dans une classe C++ fournissant les méthodes permettant le chargement d'un programme, l'interrogation de l'interpréteur et la gestion des exceptions. De plus, des prédicats prédéfinis sont fournis afin de simplifier l'interopérabilité avec l'API Win32 pour l'interrogation de la base de registre, le débuggage...

1.4 Mécanisme d'interopérabilité entre Java et Prolog

Nous allons maintenant nous intéresser aux différentes possibilités qui nous sont offertes pour réaliser l'interopérabilité entre Java et Prolog. Je vais dans un premier temps présenter les différents mécanismes. Enfin, pour conclure cette section, je vais discuter des avantages et inconvénients de chacun et décrire l'approche que nous allons suivre dans le reste de ce mémoire.

Je vais faire la distinction entre deux types de mécanismes d'interopérabilité entre Java et Prolog : ceux qui la réalisent en se basant sur la machine virtuelle de Java et les autres.

[3]Standards pour la spécification des pilotes d'interface réseau et pour la spécification des pilote des protocoles de transfert.

1.4.1 Mécanismes basés sur la machine virtuelle de Java

Il s'agit ici de réaliser l'interopérabilité entre Prolog et Java en utilisant des mécanismes de la machine virtuelle de Java. Trois approches sont possibles :

- l'utilisation d'un interpréteur Prolog implémenté en Java;
- l'utilisation de méthodes natives et de l'interface native java [Gor98] pour invoquer un interpréteur Prolog natif;
- l'utilisation d'un compilateur Prolog vers bytecode Java;

Interopérabilité basée sur l'utilisation d'un interpréteur Prolog écrit en Java

Ce mécanisme se base sur l'exécution des programmes Prolog par une machine virtuelle Prolog écrite en Java comme, par exemple, une machine abstraite de Warren[4] [AK91] [Hod90] [Kra] [Roy91].

Le programme Java sera exécuté par la machine virtuelle Java alors que le programme Prolog le sera par une machine virtuelle Prolog, ou interpréteur, implémenté en Java. Un mécanisme d'interopérabilité entre les deux pourra alors être écrite en Java et utilisera les mécanismes de la JVM pour l'échange et la conversions des données entre le programme Java et le programme Prolog.

La figure 1.3 illustre un tel mécanisme d'interopérabilité entre une machine virtuelle Java et un interpréteur Prolog.

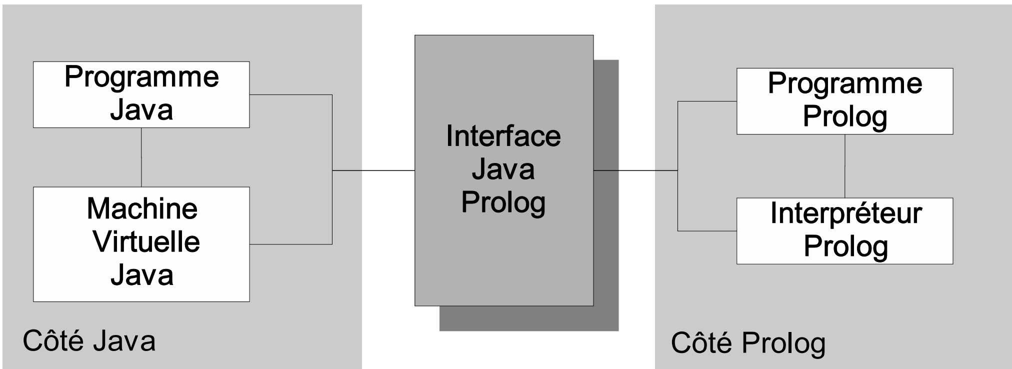

FIG. 1.3 – Interopérabilité basée sur un interpréteur Prolog écrit en Java

Une variation du mécanisme basé sur un interpréteur Prolog écrit en Java est la réalisation d'un mécanisme d'interopérabilité utilisant un compilateur Prolog vers JCode[5].
L'idée est de transformer un programme Prolog de manière à ce qu'il soit exécutable par la JVM en le compilant en un programme séquentiel en bytecode Java. Ce programme compilé pourra alors éventuellement faire appel à des bibliothèques implémentant les fonctions normalement dévolues à l'interpréteur, des fonctions telles que l'unification, le déréférencement de variables, la gestion des environnements d'exécution...
Une telle implémentation peut être trouvée dans [Eng99], bien que le compilateur Prolog proposé soit assez fragmentaire.

L'utilisation d'un compilateur Prolog vers du bytecode Java possède un avantage sur l'utilisation d'un interpréteur : il permet d'obtenir des programmes Prolog persistants, c'est-à-dire qui ne devront pas être recompilés à chaque utilisation. En effet, le code du programme Prolog compilé sera conservé sous la forme de fichiers '.class' Java.

[4] Il existe d'autres possibilités : une machine PLM, une machine abstraite de Vienne, une machine abstraite de Berkeley...

[5] Ou bytecode Java, code destinés à être exécuté par la machine virtuelle de Java, généralement contenu dans des fichiers '.class'.

Interopérabilité basée sur l'utilisation des méthodes natives

La machine virtuelle de Java implémente un mécanisme permettant d'appeler des méthodes *natives* depuis un programme écrit en Java [Eng99] [LY99] [Ven98].

Une méthode native est une méthode d'une bibliothèque ou d'un programme compilé en langage machine, c'est-à-dire en une suite de 1 et de 0 formant un programme exécutable directement par le processeur d'un ordinateur. Une telle méthode ou un tel programme ne pourra généralement être exécuté que sur une seule plate-forme[6] et requiérera, au mieux, une recompilation ou, au pire, une réécriture complète pour pouvoir être adapté à un autre environnement.

L'appel de méthodes natives en Java est facilité par l'utilisation de la "Java Native Interface" [Gor98] qui fournit des primitives et des outils permettant d'interfacer Java avec des programmes natifs écrits en C ou en C++ (voir figure 1.4). Ces primitives se chargent de la conversion de données, la gestion de l'environnement, la création/invocation d'une machine virtuelle Java depuis le programme natif...

Pour utiliser cette interface pour permettre l'interopérabilité de Java et de Prolog, il suffit de se procurer un interpréteur Prolog écrit en C, et fournissant une interface avec des programmes écrits dans ce langage[7]. Il suffira alors d'utiliser l'interface native Java pour, d'un côté, récupérer le résultat de l'interprétation du programme Prolog dans un programme Java, et, d'un autre côté, appeler des méthodes d'un programme Java[8] depuis le programme Prolog.

L'utilisation de méthodes natives possède quelques désavantages dont le plus important est la perte de la portabilité. Elle implique généralement de plus grandes difficultés d'implémentation et demande que le programmeur qui l'utilise soit familiarisé avec le langage dans lequel ont été écrites les méthodes natives.

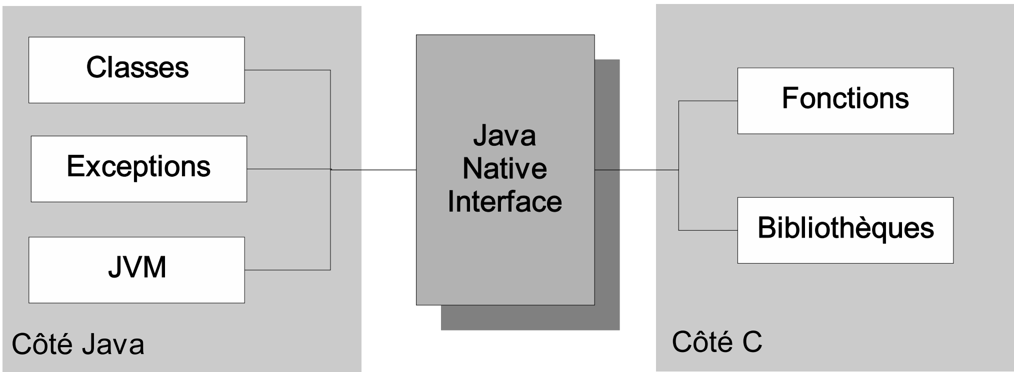

FIG. 1.4 – L'interface native Java

1.4.2 Autres mécanismes

Il existe d'autres mécanismes pour réaliser l'interopérabilité qui ne sont pas basés sur des fonctionnalités de la machine virtuelle de Java.

Redirection des entrées/sorties standards

Ce mécanisme est assez simple puisqu'il s'agit d'interroger un interpréteur Prolog en ligne de commande. Pour cela, les entrées et sorties standards devront être redirigées et le résultat fourni par l'interpréteur devra être analysé par le programme appelant. L'avantage de cette méthode

[6]celle pour laquelle il a été compilé

[7]De nombreux interpréteurs Prolog fournissent une telle interface, si aucune n'est présente, il est toujours possible d'en créer une "à la main".

[8]Par exemple des méthodes pour créer une interface graphique

est qu'elle peut être appliquée à n'importe quel interpréteur pouvant être interrogé en ligne de commande, quel que soit le langage dans lequel il est écrit. Le désavantage est qu'il s'agit d'une méthode assez lourde et peu élégante...

Mécanisme basé sur l'utilisation de XML

L'idée est d'échanger les informations entre un programme Prolog et une application Java en utilisant le langage XML développé par le W3C [EC00]. Cette méthode est déjà très largement utilisée pour l'interopérabilité entre systèmes d'information hétérogènes. Elle est assez facile à mettre en oeuvre puisqu'il suffit que chaque intervenant possède un parseur XML. Les requêtes à l'un seront dès lors "emballeés" dans un message XML qui sera transmis à l'autre. Ce dernier déballera la requête, l'évaluera et emballera la réponse dans un autre message XML qui sera envoyé au premier intervenant. L'interopérabilité se fait "par fichiers[9] interposés".

Mécanisme basé sur l'utilisation d'une architecture Client-Serveur

L'idée ici serait de créer un "serveur" Prolog qui recevra les requêtes du "client" Java et lui fournira les réponses. Pour réaliser l'interopérabilité dans les deux sens, il faudrait également posséder un "serveur" Java qui aurait comme "client" le "serveur" Prolog. Il faudra de plus créer le protocole qui permettra l'échange des requêtes et des réponses. L'interopérabilité sera ici basée sur l'utilisation des Sockets et le passage de messages entre le programme "client" et le "serveur" Prolog.

Les mécanismes suivants sont des cas particuliers de l'architecture client-serveur [OHE97] :

Mécanisme basé sur l'utilisation de XML-RPC ou SOAP :
Une autre solution, également basée sur du XML, serait l'utilisation d'un système d'appel de méthode à distance tel que XML-RPC ou SOAP [McL02]. Mais cela ne ferait que déplacer le problème puisqu'il nous faudrait concevoir une couche d'interfaçage entre l'interpréteur Prolog et la couche d'appel de méthodes à distance.

Mécanisme basé sur l'utilisation de CORBA :
L'idée ici est d'utiliser un interpréteur Prolog basé sur la technologie d'objets partagés CORBA ou d'implémenter une interface CORBA avec un interpréteur Prolog. Requêtes et réponses entre l'application Java et le programme Prolog seraient alors envoyées en utilisant le broker de CORBA. Le mécanisme d'interopérabilité entre les deux langages sera décrit explicitement au moyen d'un langage de description d'interface (ou IDL).

1.4.3 Quelle approche choisir ?

Je vais maintenant décrire l'approche que j'ai choisie pour réaliser l'interopérabilité entre Java et Prolog. L'objectif est de réaliser un mécanisme d'interopérabilité qui soit transparent et générique et basé sur la machine virtuelle de Java.

Pourquoi utiliser un mécanisme basé sur la machine virtuelle Java ?

La raison en est que nous voulons obtenir une granularité fine pour notre mécanisme. Ce que nous voulons, c'est faire des appels fréquents à de petites requêtes Prolog et ainsi fortement entrelacer programmes Java et programmes Prolog.
Cela élimine tant les mécanismes basés sur XML que sur l'architecture client-serveur dont la

[9] ou en tous cas par messages.

granularité est plus grosse. Ils sont en effet conçus pour réaliser des appels occasionnels au moteur Prolog et ne fournissent donc pas un entrelacement aussi fin que ceux basés sur la JVM.

L'utilisation d'un mécanisme basé sur la machine virtuelle de Java apporte d'autres avantages. Le premier est celui de la portabilité, qui constitue la philosophie même de la plate-forme Java. Ainsi un programme compilé pour la JVM pourra être exécuté sur n'importe quel système d'exploitation, pourvu qu'une implémentation de la machine virtuelle soit disponible[10].

Pourquoi vouloir un mécanisme générique et transparent ?

- Transparent car nous désirons cacher les détails de l'implémentation de Prolog sous-jacente;
- Générique car nous voulons un mécanisme qui soit identique quelle que soit l'implémentation de Prolog choisie (interpréteur natif, interpréteur écrit en Java, compilateur, architecture client serveur...).

Deux options sont possibles quant à l'implémentation de Prolog qui sera utilisée par le mécanisme d'interopérabilité. Nous pouvons :

1. réutiliser une implémentation de Prolog déjà existante, ce qui a pour avantage de ne pas devoir tout réimplémenter à partir de rien. Mais, cela va impliquer une augmentation de la difficulté d'implémentation du mécanisme d'interopérabilité, puisqu'il faudra adapter l'interpréteur Prolog au mécanisme d'interopérabilité;
2. implémenter un interpréteur Prolog expressément pour le mécanisme d'interopérabilité. Dans ce cas, l'interpréteur et le mécanisme d'interopérabilité seront implémentés en fonction l'un de l'autre, ce qui sera plus simple. Par contre, cela impliquera une charge de travail supplémentaire pour implémenter l'interpréteur.

Par générique, j'entends donc que le mécanisme d'interopérabilité que je propose pourra être implémenté à l'aide de n'importe quelle implémentation de Prolog, qu'elle soit implémentée pour l'occasion ou déjà existante.

Que choisir, un interpréteur conçu pour l'occasion ou déjà existant, un interpréteur natif ou implémenté en java ?

L'utilisation d'un interpréteur conçu pour l'occasion pourrait se justifier par le besoin d'implémenter des fonctionnalités bien particulières, par exemple un interpréteur Prolog minimal et très léger pour des applications embarquées, ou au contraire un interpréteur complet fournissant un grand nombre de prédicats prédéfinis pour des applications de plus haut niveau. L'utilisation d'un interpréteur Prolog déjà existant permet de garantir que les programmes déjà écrits fonctionneront sans problème avec le mécanisme d'interopérabilité, de respecter des contraintes de développement (par exemple un interpréteur imposé), d'utiliser des mécanismes propres à cet interpréteur...

Aucune raison ne requiert donc de choisir entre les trois mécanismes basés sur la machine virtuelle de Java que j'ai décrits plus haut, à savoir l'interpréteur natif ou l'interpréteur/compilateur Prolog écrit en Java.

L'utilisation d'un interpréteur écrit en Java ou d'un compilateur Prolog vers bytecode sont deux mécanismes très semblables, et le choix entre les deux dépendra surtout de la nécessité ou non d'avoir des programmes Java persistants.

A priori, l'utilisation d'un interpréteur natif sera plus efficace. Cela est dû à la nature même de la machine virtuelle de Java. Celle-ci introduit des mécanismes de sécurité et de gestion de mémoire qui diminuent les performances des programmes Java par rapport aux programmes natifs. Mais, en contrepartie, ces mécanismes facilitent l'usage de la mémoire[11] et augmentent la sécurité des applications[12].

[10] Il faut bien sûr tenir compte des problèmes de version qui font que certaines bibliothèques de classes Java pourraient ne pas être disponibles sur une version plus ancienne. C'est le cas, par exemple, du package java.nio qui n'est disponible qu'à partir de la version 1.4.

[11] L'absence de pointeurs en Java en est un.

[12] Par exemple pour les applets.

Cette tendance du langage Java à être moins performant que des langages compilés en code natif comme C ou C++ tend toutefois à disparaître au fur et à mesure que la JVM devient plus performante. Ce point alimente actuellement un débat assez houleux sur les newsgroups[13] de programmation Java et je renvoie le lecteur intéressé par ce sujet à la documentation sur les performances de la JVM, disponible en ligne sur la page officielle du langage Java[14].

Quelle est l'approche choisie dans ce mémoire ?

Pour toutes ces raisons, j'ai décidé de concevoir un mécanisme d'interopérabilité entre Java et Prolog qui permettra de choisir n'importe laquelle de ces options.

Dans ce mémoire, je vais donner une implémentation du mécanisme d'interopérabilité réalisée avec un interpréteur conçu pour l'occasion, basé sur le compilateur Prolog vers Jcode de Joshua Engel[15] [Eng99].

[13]Par exemple sur comp.lang.java.advocacy .

[14]http://java.sun.com .

[15]Ce choix de me baser sur un interpréteur déjà existant à été dicté par un soucis de facilité. Cela me permet, en effet, de ne pas avoir à réimplémenter tout un interpréteur Prolog par moi-même.

Chapitre 2

Une petite introduction au langage Prolog

Avant d'entamer la discussion sur l'interopérabilité entre Java et Prolog proprement dite, je vais introduire les concepts de base du langage Prolog qui seront utiles dans la suite. Une définition plus précise sort quelque peu du cadre de cette discussion, mais les personnes intéressées pourront trouver plus de renseignements sur le langage Prolog, par exemple, dans [SS97] [Lam95] [Dev90] et [Apt01].

2.1 Syntaxe

Avant d'aborder la sémantique (ou plutôt les sémantiques) du langage Prolog, je vais introduire la syntaxe de ce langage simple mais puissant.

2.1.1 Types de données

Dans le langage Prolog, il n'existe pas à proprement parler de données typées. Toutes les données sont des termes. Un terme peut-être :

- une constante : les constantes peuvent être soit numériques (entiers, floats...) ou non (dans ce cas on les appelle des *atomes*);
- une variable logique : les variables logiques sont des objets individuels non spécifiés. Elles sont un peu l'équivalent en programmation logique des manipulations de pointeurs. Les variables logiques Prolog ne peuvent être liées qu'une seule fois et une fois liées, leur valeur ne peut plus être modifiée;
- un terme composé : un terme composé, ou structure, est constitué d'un foncteur caractérisé par son nom (un atome) et son arité (le nombre de ses arguments), et d'une séquence de un ou plusieurs arguments;
- une liste : une liste, ou *cons*, est un cas particulier de terme composé. Une liste est constituée d'un argument appelé *tête* et d'une liste appelée *queue*. Une représentation possible[1] d'une liste en Prolog est *[Head | Tail]* ou *Head* est la tête de la liste et *Tail* sa queue.

Un terme est dit *clos* s'il ne contient pas de variable, dans le cas contraire il est dit *non clos*.

Un programme Prolog est formé par un ensemble de *clauses*. On distingue trois types de clauses :

[1] Il en existe d'autres.

- les faits : un fait exprime une relation entre deux objets. Un exemple de fait est *likes(frederic, apples).*;
- les requêtes : une requête est une clause permettant d'obtenir des informations d'un programme logique, une question. Un exemple de requête est *?-likes(frederic, apples).*;
- les règles : une règle est une clause de la forme *A :- B1,...,Bn*, où *A* est un *Goal*[2] appelé tête de la règle, alors que les *Goals B1,...,Bn* sont appelés corps de la règle. Un exemple de règle est *pere(X, Y):-male(X),parent(X, Y)*, où *pere(X, Y)* est la tête de la règle et *male(X), parent(X, Y)* son corps.

2.1.2 Généralité des structures en Prolog

Les structures Prolog sont très générales et permettent de représenter un grand nombre d'objets. Une structure peut représenter :

- un prédicat, par exemple *mange(frederic, pommes)*;
- une structure de données, un enregistrement[3], par exemple *personne(nom(jules, césar), habitat(rome))*;
- une liste, qui comme je l'ai déjà indiqué est un cas particulier des structures, par exemple .(a,.(b,.(c,[]))) que l'on écrit généralement sous la forme *[a, b, c]*, et qui possède le constructeur *cons*, noté '|' : |*(X, Y)* qui est équivalent à *[X|Y]*;
- un opérateur, par exemple +*(X, Y)* que l'on note *X + Y*;
- une clause, par exemple *:-(p, q)* qui est équivalent à *p:-q.*;
- un programme. En effet un programme Prolog est une liste de structures. Cela permet de faire en Prolog ce que l'on appelle de la méta-programmation, c'est-à-dire de la programmation sur des programmes;

2.2 Unification

Un élément important en Prolog est la règle d'unification que l'on peut énoncer de la manière suivante :

- une constante ne s'unifie qu'avec elle même;
- une variable peut s'unifier à n'importe quel terme, mais si l'on unifie deux variables liées, les objets auxquels elles sont liées doivent s'unifier;
- une structure s'unifie à une autre structure si :
  1. elles ont le même nom;
  2. elles ont le même nombre d'arguments (i.e. la même arité);
  3. leurs arguments s'unifient deux par deux;

Je vais maintenant définir la règle, ou algorithme, d'unification de manière plus formelle. Mais avant cela, voici d'abord quelques définitions.

On appelle *substitution* un ensemble fini, pouvant être vide, de paires de la forme $X_i = t_i$ où X_i est une variable et t_i un terme, et $X_i \neq X_j$ pour tout $i \neq j$. De plus, X_i ne peut apparaître dans t_j quels que soient i et j.

Soient A et B deux termes, on dira que A est une *instance* de B s'il existe une substitution θ telle que $A = B\theta$. Cela signifie que lorsque l'on remplace dans B les occurences de X_i, X_i étant une variable de θ, par la valeur t_i de ce X_i, donnée dans θ, on obtient A. Par exemple, si A est le terme $f(a,b)$ et B est le terme $f(X,Y)$, A est une instance de B par la substitution $X = a, Y = b$

[2]Un *Goal* est une entité qui n'est ni un fait, ni une requête.

[3]Ou *record*.

Soit t, t_1 et t_2 trois termes, t est une instance commune de t_1 et t_2 s'il existe deux substitutions θ_1 et θ_2 telles que $t = t_1\theta_1$ et $t = t_2\theta_2$.

Soit s et t deux termes, s est *plus général* que t s'il existe une substitution θ telle que $t = s\theta$ et qu'il n'existe aucune substitution θ' telle que $s = t\theta'$.

Soit s et t deux termes, on appelle *unificateur* une substitution θ telle que $t = s\theta$ et $s = t\theta$. Si deux termes possèdent un unificateur, on dit qu'ils *s'unifient*. Tout unificateur détermine une instance commune et inversement, toute instance commune détermine un unificateur.

On appelle *unificateur le plus général* (upg) de deux termes l'unificateur tel que l'instance commune associée est la plus générale. On peut montrer, mais je ne le ferai pas ici, que si deux termes s'unifient, alors ils ont le même *upg*.

Une version formelle de l'algorithme d'unification est donnée par la figure 2.1. Cet algorithme prend en entrée deux termes T_1 et T_2 et retourne l'unificateur le plus général de T_1 et T_2 si les deux termes s'unifient, ou échoue s'il ne s'unifient pas.

Input: t_1 et t_2 deux termes à unifier
Output: θ le plus grand unificateur de t_1 et t_2
Algorithm:

```
θ := φ ; où φ est l'ensemble vide
stack := "t1 = t2"
failure := false
while stack! = φ do
   stack.pop(X = Y)
   if Var(X) ∧ notOccursIn(X,Y) then
      stack.substitute(X,Y)
      θ.substitute(X,Y)
      θ = θ + "X = Y"
   else if Var(Y) ∧ notOccursIn(Y,X) then
      stack.substitute(Y,X)
      θ.substitute(Y,X)
      θ = θ + "Y = X"
   else if Var(X) ∧ Var(Y) ∧ X == Y then
      continue
   else if X = f(X1,...,Xn) ∧ Y = f(Y1,...,Yn) ∧ functor(f) ∧ n > 0 then
      stack.push("Xi = Yi"), i = 1..n
   else
      failure := true
   end if
end while
if failure = true then
   output failure
else
   output θ
end if
```

FIG. 2.1 – Algorithme d'unification

Remarque : de nombreuses implémentations de Prolog se passent de vérifier si un terme apparaît dans le terme auquel on veut tenter de l'unifier[4] pour des raisons de performances. Mais cela implique un risque de régression infinie lors de l'unification.

[4] C'est que l'on appelle l'*occur check*.

2.3 Backtracking

L'exécution d'un programme Prolog est, essentiellement, un parcours d'arbre de recherche ET-OU par réduction de problème en profondeur d'abord (stratégie de résolution SLD[5]).

En quelques mots[6], la résolution SLD consiste à choisir un objectif, ou *Goal*, parmi les clauses selon un ordre bien défini et à tenter de l'établir. On dira qu'un objectif est établi s'il s'unifie à un fait ou à une tête de règle dont tous les sous-objectifs sont établis.

Le backtracking ou rétro parcours consiste à retourner en arrière pour choisir un nouvel objectif à établir lorsque l'objectif courant échoue. Deux cas sont possibles :

- si l'objectif courant O est le fils d'un noeud-OU de l'arbre de recherche, on annule les substitutions opérées pour tenter d'établir O et on tente d'établir l'objectif suivant O';
- si l'objectif courant O est le fils d'un noeud-ET, on retourne (backtracking ou rétro parcours) au dernier objectif O', on annule les substitutions opérées pour tenter d'établir cet objectif Ob' et on tente d'établir Ob' avec un autre jeu de substitutions. Si plus aucun rétro parcours n'est possible, la tentative échoue.

Si la tentative d'établir Ob' échoue, on retourne au dernier Ob'' et ainsi de suite...

Ce retour en arrière est appelé rétro parcours ou backtracking.

2.4 Sémantiques de Prolog

Le langage Prolog possède deux sémantiques différentes mais complémentaires : une sémantique déclarative héritée de la logique du premier ordre et une sémantique procédurale.

2.4.1 Sémantique déclarative

La sémantique déclarative du programme Prolog est indépendante de son mode d'exécution. Dans cette sémantique, un programme est vu comme un ensemble d'axiomes. Ces axiomes se trouvent sous la forme $A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow C$, appelée clause de Horn et représentée en Prolog par C :- $A_1,...,A_n$. Un programme Prolog peut être vu comme une théorie en logique du premier ordre. La sémantique déclarative d'un programme Prolog sera un modèle de cette théorie, c'est-à-dire une interprétation, un ensemble de valeurs qui rend une clause vraie ou fausse, qui la vérifie. Cette définition est assez approximative et ne constitue qu'une intuition. Le lecteur intéressé pourra trouver plus de renseignements dans [Lam95] et [DL99].

2.4.2 Sémantique procédurale

La sémantique procédurale d'un programme Prolog décrit un mode d'exécution de ce programme et fait référence à l'ordre des clauses dans une procédure et à celui des littéraux dans une clause. On peut alors voir le programme Prolog comme un ensemble de procédures qui vont être exécutées dans un certain ordre.

Algorithme de résolution

L'algorithme de résolution est donné par la figure 2.2.

Une procédure à corps vide, c'est-à-dire un *fait*, se termine immédiatement et une *requête* $Q \equiv Q_{\wedge} ... \wedge Q_m$ sera exécutée comme un appel successif aux procédures Q_m.

[5]Pour "Selecting a literal, using a Linear strategy, limited to Definite clauses" aussi appelée résolution LUSH.

[6]De plus amples renseignements pourront être trouvés dans [Lam95] et [SS97]

Input: G, goal et P, programme
Output: une instance de G conséquence logique de P, 'non' sinon
Algorithm:
  $resolvent = G$
  while $resolvent \neq \phi$ **do**
    $select(A) : A \in resolvent$
    $select(A') : clause(A'), A' \leftarrow B_1 ... B_n \in P \wedge unify(A, A') \wedge mgu(A, A') = \theta$
    if $\nexists A, A'$ **then**
      break
    end if
    $resolvent = resolvent[A/B_1 ... B_n]$
    $resolvent = resolvent.\theta$
    $G = G.\theta$
  end while
  if $resolvent = \phi$ **then**
    output G
  else
    output 'non'
  end if

FIG. 2.2 – Algorithme de résolution de Prolog

2.4.3 Extensions extra-logiques

Le langage Prolog fournit souvent, selon les implémentations, un ensemble de primitives extra-logiques prédéfinies. Celles-ci permettent d'accroître l'efficacité du langage en fournissant des structures de contrôle (CUT[7], fail...) et des opérateurs à effets de bord (arithmétique, entrées-sorties, méta-prédicats...) qui ont pour effet de faire sortir le langage de la logique du premier ordre (perte de l'inversibilité, du raisonnement déclaratif...).

Ces extensions extra-logiques ne sont pas supportées par tous les interpréteurs Prolog.

[7] Le cut, '!', permet d'empêcher le backtracking dans le but d'éviter l'exploration de certaines parties de l'arbre de recherche, dont on sait qu'elle va mener à un échec ou que l'on ne veut pas explorer pour des raisons de performances. Il est, entre autres, utilisé avec *fail* pour implémenter le prédicat *not(P)* en Prolog.

Première partie

Discussions préliminaires

Chapitre 3

Machines virtuelles et modèles de mémoire

Dans ce chapitre, je vais introduire l'architecture et le modèle de mémoire[1] de la machine virtuelle Java, de la machine abstraite de Warren et d'un compilateur Prolog vers JCode. Il ne s'agit pas dans ce chapitre de décrire chacune de ces architectures en détails mais simplement d'introduire les bases qui seront utiles pour la suite de la discussion. A partir de celles-ci, je décrirai comment concilier les différentes architectures de mémoire et l'impact qu'elles vont avoir sur l'interopérabilité entre Java et Prolog.

3.1 La machine virtuelle Java

Cette section constitue une introduction à l'architecture de la machine virtuelle de Java. Les personnes intéressées par plus de détails sur l'architecture et le fonctionnement de la JVM peuvent consulter les livres [LY99] [Ven98] [Eng99].

Mais avant d'entamer la description de la machine virtuelle de Java, il est bon de faire une petite remarque sur ce que signifie ce terme. En réalité, il peut désigner trois choses différentes :

- la spécification décrite dans [LY99];
- une machine abstraite pour l'exécution de bytecode Java;
- une implémentation de cette machine abstraite;

Ce que nous allons décrire dans la suite est la machine abstraite pour l'exécution de bytecode Java.

3.1.1 Types de données de la machine virtuelle de Java

Les types de données de la machine virtuelle de Java se groupent en deux catégories : les types primitifs (numéraux, booléens et returnAddress) et les références.

La figure 3.1 donne les différents types de données. Ceux-ci sont décrits de manière plus précise dans la table 3.1.

Tous ces types de données correspondent à des types du langage Java sauf *returnAddress*. Remarquons qu'en pratique, les types byte, short, char et boolean sont représentés dans la machine virtuelle de Java par des entiers 32 bits[2].

[1] Par là, j'entends l'architecture et la gestion de la mémoire

[2] Il semble que ce ne soit plus le cas dans la version 1.4 du J2SE de Sun.

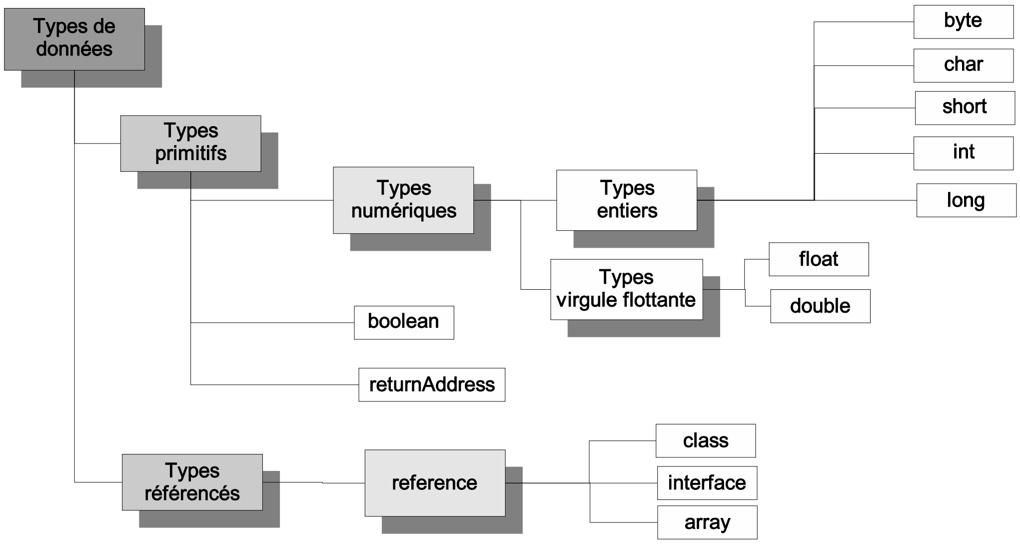

FIG. 3.1 – Types de données de la JVM

type	données représentées
byte	entiers 8 bits signés en complément à 2
short	entiers 16 bits signés en complément à 2
int	entiers 32 bits signés en complément à 2
long	entiers 64 bits signés en complément à 2
char	entiers 16 bits non signés pour représenter les caractères Unicode
float	nombres à virgule flottante 32 bits
double	nombres à virgule flottante 64 bits
boolean	valeur true ou false
reference	3 types différents : interface, classes ou tableaux
returnAddress	utilisé par la JVM comme pointeur pour les adresses de retour

TAB. 3.1 – Types de données de la JVM

3.1.2 Architecture de la machine virtuelle de Java

La machine virtuelle de Java est une machine abstraite capable d'exécuter du bytecode Java. Il s'agit, comme nous le verrons plus loin, d'une machine à piles. Le code du programme à exécuter, sous forme de JCode, se trouve, généralement, dans un fichier classe Java. Le format de ces fichiers est un format binaire indépendant du hardware ou du système d'exploitation.

Pour permettre cela, la JVM est composée de plusieurs éléments (voir figure 3.2) :
- un système de chargeurs de classes qui permet de charger le JCode d'une classe Java dans la machine virtuelle;
- une zone de mémoire contenant les données d'exécution;
- une moteur d'exécution, chargé d'exécuter les instructions d'un programme;
- une interface de méthodes natives permettant l'appel à des bibliothèques de fonctions natives;
- des bibliothèques de fonctions natives implémentant les primitives pour communiquer avec le système hôte;

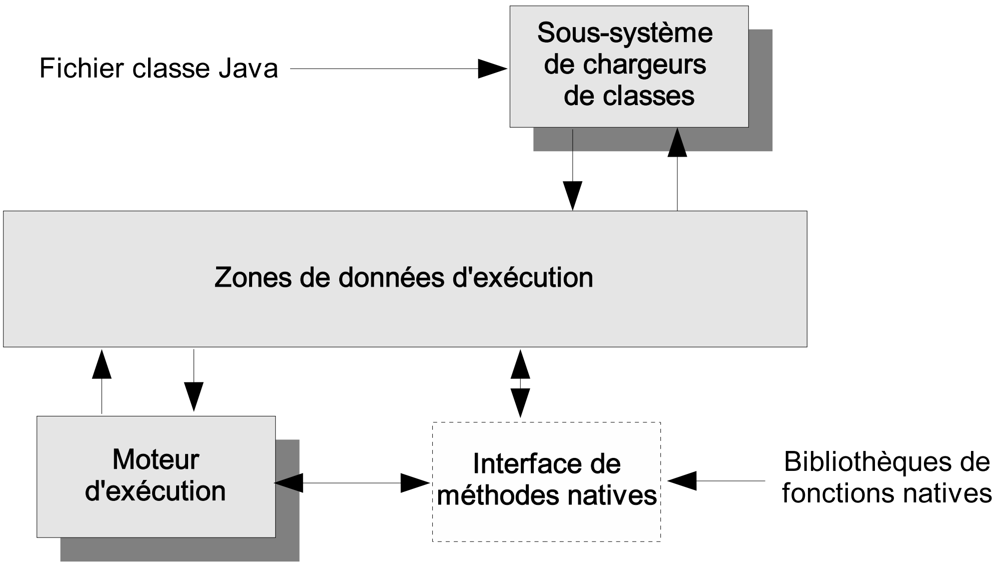

FIG. 3.2 – Architecture de la machine virtuelle de Java

Je n'entrerai pas dans les détails de cette architecture, une description complète pouvant être trouvée, comme indiqué précédemment, dans [LY99]. Par contre, les pages suivantes se penchent de manière plus précise sur l'organisation et la gestion de la mémoire dans la JVM.

3.1.3 Organisation et gestion de la mémoire dans la JVM

La mémoire de la machine virtuelle Java (voir figure 3.3) est composée de plusieurs éléments qui sont, pour certains, communs à l'ensemble de la JVM ou, pour d'autres, particuliers à un thread :
- le zone des méthodes[3];
- le tas[4];
- les piles Java;
- les registres du compteur de programme;
- les piles de méthodes natives;

Passons maintenant à la description de ces différents éléments.

[3] Ou *Method Area*.
[4] Ou *Heap*.

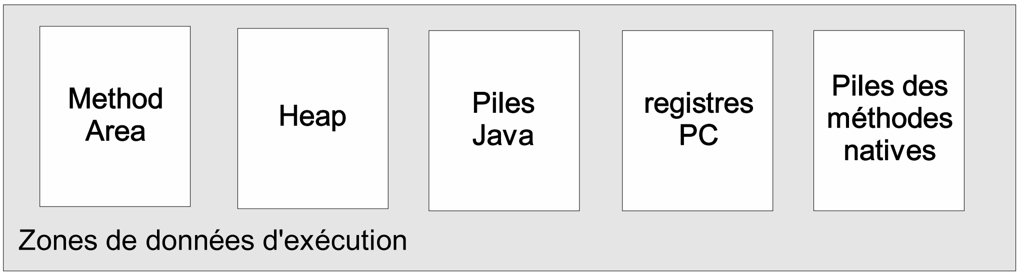

FIG. 3.3 – Organisation de la mémoire de la machine virtuelle de Java

La zone des méthodes

La *zone des méthodes* (voir figure 3.4) est un peu l'équivalent du segment TEXT d'un processus UNIX. Il contient les structures des classes chargées dans la JVM : les constantes (littéraux numériques et références des champs et méthodes), les champs, les données des méthodes, le code des méthodes, le code des constructeurs[5] et le code des instructions spéciales d'initialisation des classes et interfaces[6].

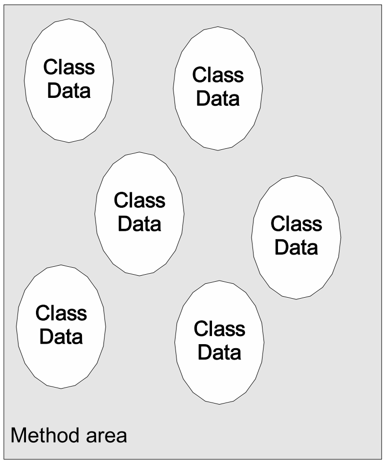

FIG. 3.4 – zone des méthodes

La *zone des méthodes* est partagée par tous les threads de la machine virtuelle Java et est créée à l'initialisation de celle-ci. L'ensemble des constantes est appelé *runtime constant pool* et est un peu l'équivalent de la table des symboles pour un programme UNIX. Il contient la correspondance entre le nom des constantes[7] d'une classe Java et leur valeur.

Le tas

Le *tas* (voir figure 3.5) contient les instances de classes et les tableaux. Les objets inutiles sont effacés par un mécanisme automatique de gestion de la mémoire appelé garbage collection. Comme la *zone des méthodes*, le *tas* est partagé entre tous les threads de la machine virtuelle.

[5]Les méthodes <init> au niveau de la machine virtuelle.

[6]Les instructions placées dans la partie *static{...}* de la classe et correspondant à la méthode <clinit> au niveau de la machine virtuelle.

[7]représentant des données, mais aussi des noms de méthodes, de classes, de champs...

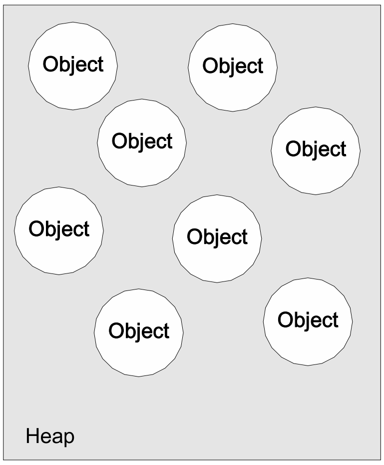

FIG. 3.5 – Le tas (Heap)

Les piles Java

A chaque thread de la machine virtuelle correspond une pile (voir figure 3.6). Les piles servent à stocker les variables locales, les résultats partiels et sont aussi utilisées lors de l'invocation ou le retour des méthodes.

Chaque pile est composée de *frames* utilisés pour :

- stocker les données;
- stocker les résultats partiels;
- l'édition dynamique de liens;
- retourner les valeurs des méthodes;
- dispatcher les exceptions;

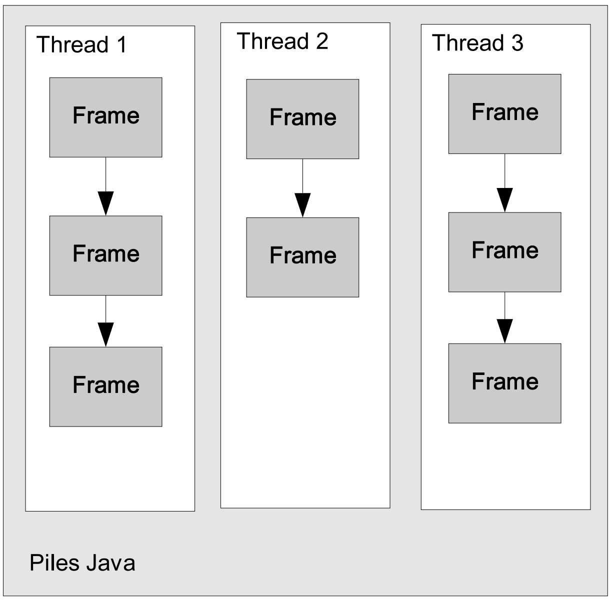

FIG. 3.6 – Les piles Java

A chaque invocation de méthode, un nouveau frame est créé et ajouté au sommet de la pile du thread responsable de l'invocation. Lorsqu'une méthode se termine, le frame correspondant est retiré de la pile du thread qui l'avait invoquée. On appelle *frame actif* (ou courant) le frame situé au sommet d'une pile, c'est-à-dire le frame correspondant à la méthode en cours d'exécution (voir figure 3.7). Un frame est composé d'une table des variables locales et d'une pile d'opérandes.

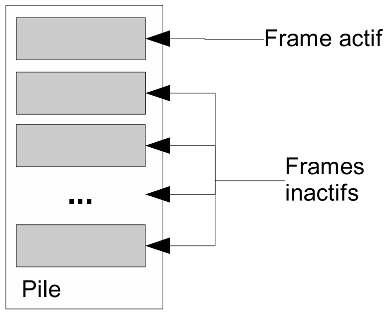

FIG. 3.7 – Organisation d'une pile Java

La table des variables locales est un tableau muni d'un adressage indexé. Son rôle est de permettre le passage de paramètres aux méthodes invoquées. La première variable, d'index 0, contient toujours la référence à l'objet courant.

Les variables de type int, short, char, byte, float, boolean, returnAddress et reference peuvent être stockées dans une seule case du tableau[8] alors que les variables de type long et double en nécessitent deux. Cela peut ne pas être le cas pour les tableaux de types primitifs et dépend, de plus, de l'implémentation de la machine virtuelle Java[9].

La pile des opérandes est une pile LIFO[10] servant à préparer les paramètres à passer à une méthode et à recevoir les résultats. C'est la pile de travail d'une méthode Java. C'est dans celle-ci qu'elle place et modifie les données qu'elle manipule et qu'elle placera sa valeur de retour.

Voilà qui clôt cette description sommaire de la machine virtuelle de Java.

3.2 La machine abstraite de Warren

Cette section constitue une introduction à l'architecture de la machine de Warren[11]. Cette machine n'est pas la seule possibilité pour la construction d'un interpréteur Prolog[12] mais elle constitue celle qui est la plus utilisée. Comme dans le cas de la machine virtuelle de Java, ce qui suit n'est qu'une introduction. Plus de renseignements sur les machines Prolog en général, et la machine de Warren en particulier, pourront être trouvés dans [AK91] [Hod90] [Kra] [Roy91].

3.2.1 Architecture générale de la machine abstraite de Warren

La machine abstraite de Warren (WAM) fut introduite pour la première fois en 1983 par David H.D. Warren. Comme PLM (Prolog Machine), la machine de Warren est une machine abstraite capable d'exécuter un programme écrit en langage Prolog. Pour ce faire, elle fournit une architecture de mémoire, constituée de piles et de registres, ainsi qu'un set d'instructions abstraites permettant la manipulation de la mémoire et l'exécution de programmes Prolog. Pour être interprété par une machine de Warren, un programme Prolog va être traduit en instructions abstraites.

[8]C'est-à-dire dans un seul mot.
[9]Ce n'est d'alleurs plus le cas dans la version 1.4 de la JVM de Sun Microsystems.
[10]LIFO = last in first out.
[11]La machine abstraite de Warren n'est pas la seule manière d'exécuter des programmes Prolog, [Roy93] retrace l'histoire des implémentations de Prolog de 1983 à 1993.
[12]Le Chapitre 2 donne une rapide introduction à la syntaxe et aux mécanismes du langages Prolog.

3.2.2 Organisation et gestion de la mémoire dans la WAM

La mémoire de la machine abstraite de Warren (voir figure 3.8) est constituée d'un certain nombre de registres et de zones de mémoire : le tas[13], la pile d'environnement, la pile de *choice point*[14] ou pile de bactracking, le *trail*, la pile *push-down* (PDL) ou pile d'unification et l'espace de code.
Dans la plupart des versions de la WAM, la pile d'environnement et la pile de *choice point* sont remplacées par une seule pile que l'on appelle tout simplement le Stack.

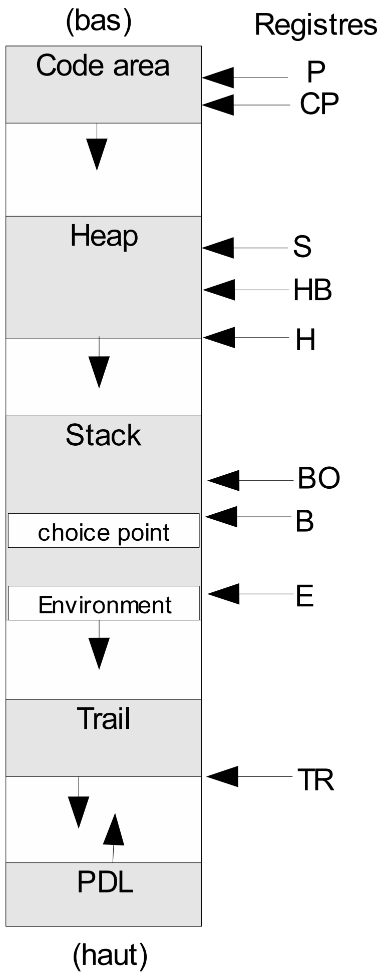

FIG. 3.8 – Organisation de la mémoire et registres de la machine abstraite de Warren

Zones de mémoire

Détaillons le rôle de chacune des zones de mémoire de la machine abstraite de Warren.

- Le tas, ou pile globale, est une pile qui permet de stocker les termes Prolog.
- La pile d'environnement contient des environnements qui encapsulent les variables locales à une clause. A cause du backtracking, elle ne suit pas réellement la politique de

[13]Ou Heap.
[14]Un choice point est une structure qui encapsule un état de backtracking. J'ai décidé de conserver le terme anglophone car je n'ai pas trouvé de traduction vraiment satisfaisante.

gestion d'une pile. Les structures de données stockées dans cette pile sont appelées *frames d'environnement*.

- La pile de *choice point*, ou pile de backtracking, contient des structures de données appelées *choice point frames* dont le rôle est d'encapsuler un état d'exécution[15] pour permettre le backtracking.
- Le Trail est une pile permettant de garder une trace des variables qui peuvent être "déliées" lors du backtracking.
- La pile push-down, ou pile d'unification, est utilisée lors de l'unification des termes composés.
- L'espace de code contient les instructions du langage abstrait résultant de la traduction du programme Prolog.

États d'exécution

L'état interne d'une machine abstraite de Warren est contenu dans un ensemble de registres donnés par la table 3.2

registre	description
P	compteur de programme, pointe vers la prochaine instruction à exécuter
CP	point de continuation, équivalent de l'adresse de retour
S	pointeur vers la structure courante dans le tas
HB	pointeur vers le sommet du tas lors de la création du choice point le plus récent
H	pointeur vers le sommet du tas
BO	pointeur vers un *cut*
B	pointeur vers le choice point le plus haut, c'est-à-dire le plus récent, dans le Stack ou la pile de backtracking
E	pointeur vers le frame de l'environnement courant dans le Stack ou la pile d'environnement
TR	pointeur vers le sommet du Trail
A_n	registres pour les arguments
Y_n	position des variables locales dans l'environnement

TAB. 3.2 – États d'exécution de la WAM

Frame d'environnement

Un frame d'environnement est une structure de données permettant d'encapsuler la valeur des variables locales d'une clause. La structure d'un frame d'environnement est donnée par la table 3.3 .

Choice point frame

Un *choice point* représente un point de l'exécution d'un programme où un backtracking est possible. Les valeurs des registres[16] au moment de la création d'un choice point sont stockées dans une structure de données appelée *choice point frame*. Le contenu d'un de ces frames est donné par la table 3.4.

[15] C'est-à-dire les valeurs des registres.
[16] Du moins ceux qui devront être restaurés au moment du backtracking.

registre	description
CE	pointeur vers le frame d'environnement de continuation
CP	point de continuation
Y_1	première variable locale
...	
Y_n	nième variable locale

TAB. 3.3 – Contenu du frame d'environnement

registre	description
n	arité
A_1	premier argument
...	
A_n	nième argument
CE	environnement de continuation
CP	point de continuation
B	pointeur vers le choice point précédent
BP	pointeur vers la prochaine clause
TR	pointeur du sommet du trail
H	pointer du tas
BO	pointer de *cut*

TAB. 3.4 – Contenu d'un choice point frame

3.2.3 Set d'instructions de la WAM

J'avais indiqué plus haut que la machine abstraite de Warren était constituée d'une architecture mémoire et d'un set d'instructions abstraites. Ce sont ces dernières dont je voudrais maintenant discuter. Il ne s'agit pas de donner une vue exhaustive mais simplement de faire une petite introduction sur le sujet. Les personnes désireuses d'obtenir plus de renseignements sur les instructions de la WAM en trouveront une description complète, y compris leur pseudo code, dans l'excellent tutoriel de Hassan Aït-Kaci [AK91].

Le set d'instructions abstraites de la WAM comprend :

- des instructions pour charger des arguments dans des registres (instructions *put_Xxxx*);
- des instructions pour l'unification (instructions *get_Xxxx* et *unify_Xxxx*);
- des instructions de contrôle procédural (instructions *call, execute, proceed, allocate* et *deallocate*);
- des instructions de sélection de clauses[17] (instructions *switch_on_term, switch_on_structure* et *switch_on_constant*);
- des instructions pour le backtracking et la gestion des *choice point* (instructions *try_me_else, return_me_else* et *trust_me_else fail*);

Un programme Prolog sera traduit en un ensemble d'instructions abstraites pour pouvoir être exécuté par la WAM. La manière dont cela est réalisé sort du cadre de ce mémoire. Le lecteur intéressé pourra toutefois trouver plus de détails dans [AK91].

Voilà qui clôt cette introduction à la machine abstraite de Warren.

3.3 Un compilateur Prolog vers JCode

Pour cette section, nous allons nous baser sur le compilateur fourni par Joshua Engel dans son livre [Eng99].

3.3.1 Architecture générale du compilateur de Engel

L'idée du compilateur de Joshua Engel est d'utiliser la machine virtuelle de Java comme une machine abstraite Prolog et le JCode comme langage abstrait. Cette idée est tout à fait réalisable. En effet, si l'on compare l'architecture de la machine virtuelle de Java et celle de la machine abstraite de Warren, on peut constater que leurs structures sont assez semblables. La table 3.5 compare les deux machines abstraites en mettant en évidence leurs ressemblance au point de vue de l'organisation de la zone de mémoire.

Zone de mémoire	JVM	WAM
Code du programme	contenu dans la zone des méthodes	contenu dans espace de code
Mémoire pour les objets	objets stockés dans le tas	structures stockées dans le tas
Ensemble de variables locales	table des variables locales	Pile d'environnement
Pile de travail	pile d'opérande	PDL

TAB. 3.5 – Ressemblances entre la JVM et la WAM

Pour pouvoir traduire un programme Prolog en bytecode Java et l'exécuter avec une machine virtuelle Java, il nous manque encore les éléments suivants :

[17] C'est-à-dire de branchement conditionnel.

- un mécanisme de backtracking;
- le Trail;

Un autre problème subsiste : les variables de la JVM sont très différentes des variables logiques de Prolog. La solution va être de créer une structure de données pour représenter les variables Prolog en Java. Il en faudra également d'autres pour représenter les structures et listes Prolog.

En ajoutant à tout cela les coroutines nécessaires pour la manipulation du Trail, des structures, des listes et des variables, il devient possible de compiler un programme Prolog en classes Java et de le faire exécuter par la JVM. C'est exactement ce que se propose de réaliser le compilateur de Joshua Engel.

3.3.2 Le compilateur

Le rôle du compilateur Prolog vers JCode de Engel va être de transformer un prédicat Prolog[18] en une classe Java. Je ne vais pas détailler ici la technique utilisée[19].

Une partie de l'algorithme de résolution va se trouver directement codée dans la classe Java, alors qu'une autre sera codée dans des coroutines fournies par un environnement d'exécution Prolog.

3.3.3 Environnement d'exécution Prolog et structures de données

L'idée de Engel, développée dans son livre [Eng99], est d'implémenter une classe Java qui aura le rôle d'environnement d'exécution Prolog et contiendra les éléments absents de la machine virtuelle de Java. C'est cette classe qui implémentera le Trail, les méthodes de gestion du Trail, l'algorithme d'unification, les méthodes de gestion des variables... Il implémentera également le moteur d'exécution de Prolog chargé d'évaluer les requêtes.

L'environnement d'exécution va manipuler des structures de données représentant les structures de données Prolog. Il s'agit des classes Structure, Cons et Var qui représentent respectivement les termes composés, les listes et les variables Prolog. Pour ce qui est des atomes, le choix de J. Engel s'est porté sur l'utilisation de la classe java.lang.String pour les représenter.

Une variable Prolog dans le compilateur est représentée par une structure de données contenant son nom[20] et l'objet qu'elle contient[21]. Une structure Prolog est représentée par une structure de données contenant son nom et un tableau contenant ses arguments. Une liste Prolog est représentée par une structure de données contenant sa tête et sa queue.

Le Trail (voir figure 3.9), pour sa part, est implémenté directement à l'intérieur de l'environnement d'exécution sous la forme d'une instance de java.util.Stack. Le rôle du Trail, comme indiqué plus haut, va être de garder une trace des variables qui peuvent changer durant le backtracking. Pour ajouter les variables d'un prédicat sur le Trail, les étapes à suivre sont :

1. placer un repère (ou tag) au sommet du Trail;
2. ajouter les variables une par une au sommet du Trail (push);

Les méthodes de gestion fournies par l'environnement d'exécution Prolog sont :

- *interpret*, qui permet d'interpréter un prédicat Prolog[22];
- *findVars*, qui retourne les arguments d'une structure ou d'une liste;
- *unify*, qui implémente l'algorithme d'unification;
- *deref*, qui déréférence le contenu d'une variable logique;

[18] Un ensemble de règles ayant le même nom et la même arité est appelé un prédicat, il s'agit, en gros, de l'équivalent en Prolog des procédure des langages impératifs.

[19] Les personnes intéressées pourront trouver tous les renseignements dans [Eng99].

[20] Si la variable est anonyme, un nom unique sera créé automatiquement.

[21] Appelé binding.

[22] Cette méthode se base sur les méthodes d'introspection fournies par la classe Java java.lang.Class ainsi que dans les classes du package java.lang.reflect.* .

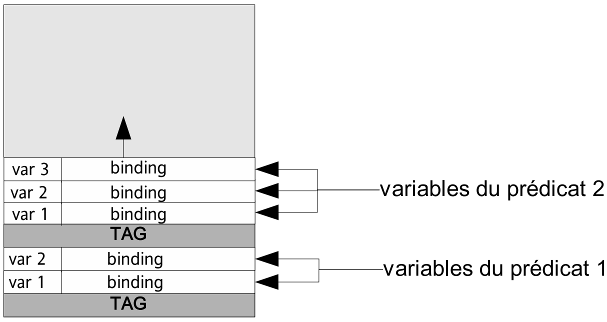

FIG. 3.9 – Structure du Trail du compilateur Prolog de J. Engel

- *setBinding*, qui crée une nouvelle variable logique contenant un objet donné en argument et l'ajoute au sommet du trail (push);
- *undoBindings*, qui enlève du trail les variables jusqu'à un certain repère (ou tag);
- *markTrail*, qui ajoute un repère au sommet du Trail et le retourne pour usage ultérieur;

Voilà qui termine cette description de l'architecture du compilateur Prolog vers JCode de Joshua Engel.

3.4 Problèmes de gestion de la mémoire entre les deux langages

Je vais maintenant me pencher sur un des problèmes qu'il va falloir résoudre pour pouvoir réaliser un mécanisme d'interopérabilité entre Java et Prolog : le partage de la mémoire entre un programme Java et un programme Prolog. L'objectif est de faire interagir deux machines abstraites différentes : la machine virtuelle de Java et l'interpréteur Prolog. Ces deux machines peuvent partager le même espace mémoire ou non. Dans cette section, je vais présenter les avantages et inconvénients de chacune de ces deux options du point de vue de l'efficacité, de la complexité et de la granularité du mécanisme d'interopérabilité. J'essaierai ensuite d'indiquer dans quelle catégorie se trouvent les mécanismes d'interopérabilité basés sur la machine virtuelle de Java que j'avais introduits au début de ce mémoire, à savoir l'utilisation d'un interpréteur natif, l'utilisation d'un interpréteur écrit en Java et l'utilisation d'un compilateur Prolog vers JCode[23].

3.4.1 Partage de l'espace mémoire

Quels sont les effets du partage ou non de l'espace mémoire entre la machine virtuelle de Java et l'interpréteur Prolog sur le mécanisme d'interopérabilité ? C'est la question que je me propose de discuter dans les paragraphes qui suivent. Commençons par le cas le plus simple : deux espaces mémoires séparés.

Espaces mémoire séparés

Si les deux espaces mémoires sont séparés, nous pouvons voir notre système comme deux processus communiquant entre eux à travers une interface[24]. Dans ce cas le mécanisme d'interopérabilité est simple, il s'agit d'un échange de données et d'appels de méthodes, entre deux

[23] Dans ce qui suit et pour ne pas surcharger le texte, j'utiliserai le terme *"machine Prolog"* pour désigner un interpréteur Prolog quelconque, qu'il soit natif, écrit en Java ou qu'il s'agisse du compilateur Prolog vers JCode.

[24] Dans notre cas l'Interface Native Java (JNI).

processus totalement indépendants ayant chacun leur vision de la mémoire. Un tel mécanisme est modélisé par la figure 3.10. La gestion de l'espace mémoire est plus simple du fait qu'aucun des deux processus ne peut accéder à la mémoire de l'autre et que le changement de contexte est géré par le système d'exploitation hôte.

Une autre question se pose : comment garantir la cohérence automatique entre la vision qu'a le programme Java de la mémoire de l'interpréteur Prolog et le véritable état de celle-ci ? On espère, et on est en droit de le faire, que la machine Prolog se chargera de garantir la cohérence, car comme le programme Java n'a aucun accès à la mémoire de Prolog, il ne peut donc ni vérifier ni modifier celle-ci. D'un autre côté, cela constitue également un avantage, dans le sens où le programme Java ne peut pas introduire d'incohérence en modifiant la mémoire de Prolog.

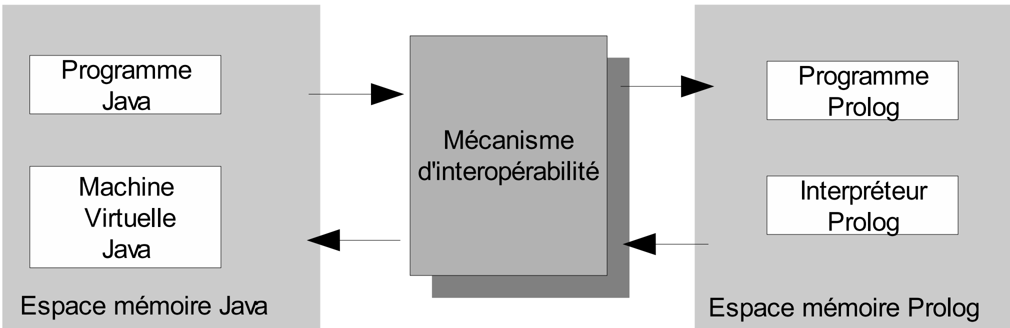

FIG. 3.10 – Modèle de mécanisme d'interopérabilité avec espaces mémoires séparés

Un espace mémoire partagé

L'idée ici est que les deux machines abstraites vont partager le même espace mémoire. Cela signifie que les structures de données de chacune d'entre elles pourront éventuellement être visibles pour l'autre. Le partage de l'espace mémoire n'est bien sûr possible que si la machine Prolog est implémentée en Java et qu'elle s'exécute dans la même machine virtuelle que le programme Java qui interagit avec elle.

Pour simplifier le problème, je ne vais considérer dans cette discussion que le cas où le programme Java et la machine Prolog s'exécutent dans le même Thread[25]. Cela correspond au cas où les appels à la machine Prolog sont satisfaits de manière synchrone[26].

Dans ce cas, programmes Prolog et programmes Java vont se partager le même espace mémoire, c'est à dire le même tas, la même zone des méthodes, le même compteur de programme et la même pile.

Mais le partage de l'espace mémoire mène à un problème. Si le programme Java a accès à la mémoire du programme Prolog, rien ne l'empêche, a priori, de modifier celle-ci et d'introduire des incohérences au sein de la machine Prolog. Par exemple, que se passerait-il si une méthode Java attribuait une valeur à une variable Prolog non liée ou si elle modifiait la valeur d'une variable logique déjà liée ? Il faudra donc fournir un mécanisme[27], que ce soit un mécanisme logiciel ou simplement des règles d'utilisation, pour gérer ces cas de figure.

Granularité et complexité du mécanisme d'interopérabilité

J'avais annoncé plus haut que nous allions aborder l'effet du partage ou non de la mémoire sur la granularité du mécanisme d'interopérabilité. J'avais éludé la question jusqu'ici parce celle-ci

[25] C'est dans cette optique que je vais spécifier et implémenter le mécanisme d'interopérabilité qui est au coeur de ce mémoire.

[26] Certains interpréteurs Prolog, comme le très bon mais propriétaire JIProlog, permettent l'évaluation de requêtes asynchrones, mais cette possibilité est absente de la plupart d'entre eux.

[27] Une sorte de mécanisme de protection de mémoire.

n'a de sens que si l'on peut comparer les deux possibilités et qu'il me fallait donc les décrire.

Le partage de mémoire va permettre de réduire la granularité du mécanisme d'interopérabilité. En effet, si les programmes Java et Prolog se partagent la mémoire, les instructions correspondant à l'exécution du code du programme Java, et celles correspondant à l'exécution du programme Prolog pourront être *entrelacées*. Il en résulte que le coût de l'interfaçage est réduit, tout le code s'exécutant au sein du même processus, et permet donc des interactions plus petites et plus fréquentes entre Java et Prolog. Cette entrelacement entre code Java et code Prolog sera limité. En effet, cet entrelacement ne cocernera pas les instructions individuelles, mais un entrelacement au niveau des méthodes[28] Java exécutées. La figure 3.11 illustre cette situation.

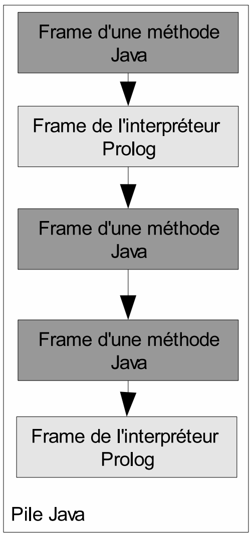

FIG. 3.11 – Entrelacement dans la pile Java

Dans le cas des espaces mémoires séparés, par contre, le coût des interactions entre Java et Prolog se trouvera augmenté par le coût des changements de contexte entre le processus de la machine virtuelle Java et celui de la machine Prolog.

D'un autre côté, la gestion de l'espace mémoire sera plus complexe si l'on utilise un espace partagé. Il faudra, comme je l'ai déjà dit plus haut, fournir un mécanisme pour garantir qu'un programme Java n'aura pas un comportement hostile envers un programme Prolog et n'introduira pas d'incohérences dans la mémoire de la machine Prolog. Les changements de contexte entre Prolog et Java devront, de plus, être gérés par le mécanisme d'interopérabilité, ce qui sera réalisé par de simples invocations de méthodes Java.

3.4.2 Mécanismes d'interopérabilité et gestion de la mémoire

J'ai préféré discuter des différents mécanismes d'interopérabilité que j'avais envisagés après avoir décrit les deux possibilités, pour le partage de la mémoire, pour la simple raison que cette discussion n'est pas aussi triviale qu'il n'y paraît à première vue.

Instinctivement, on aurait tendance à dire que l'utilisation d'un interpréteur Prolog natif appartient au premier cas, c'est-à-dire celui de deux processus aux espaces mémoire séparés, et que les deux mécanismes utilisant une machine Prolog implémentée en Java, qu'il s'agisse d'un interpréteur ou d'un compilateur, appartiennent au cas du partage de l'espace mémoire. C'est peut-être aller un peu vite. Étudions d'un peu plus près ces deux cas de figure.

[28]C'est-à-dire les méthodes Java correpondant à l'exécution du programme Java et celles correspondant aux appels à l'interpréteur Prolog.

Interpréteur natif et espaces mémoire séparés

Il est évident que le mécanisme d'interopérabilité basé sur l'utilisation d'un interpréteur natif correspond au cas des espaces mémoire séparés. Mais quels vont être les problèmes posés par cette situation ?

Le premier problème est qu'il va falloir un mécanisme permettant aux deux processus de communiquer entre eux. Un tel mécanisme devra faire appel à des primitives du système[29] d'exploitation sur lequel tournent les deux processus. Cela va résulter en une perte de portabilité de l'application. La machine virtuelle de Java offre une telle fonctionnalité à travers l'usage de mécanismes internes (les piles de méthodes natives), mais également grâce à des outils externes : l'interface native Java (JNI) [Gor98].

Le deuxième problème concerne les objets manipulés par le programme Java. L'interface JNI fournit des méthodes permettant d'accéder aux champs et méthodes d'objets Java. Elle permet même d'instancier un machine virtuelle Java depuis un programme natif et de lui faire exécuter des programmes, instancier et manipuler des objets... Parmi les possibilité de JNI, citons le passage de paramètres de Java vers un programme natif, l'appel de méthodes et l'accès aux champs des classes et objets depuis un programme natif... Bref, tout semble être disponible pour permettre de réaliser un mécanisme d'interopérabilité utilisant les méthodes natives. Le problème est que ce mécanisme est assez lourd et pourrait demander une modification en profondeur de l'interpréteur Prolog natif. Il n'existe de plus aucun mécanisme permettant une correspondance directe entre classes et objets Java et, par exemple, classes et objets C++.

Partage de l'espace mémoire pour les interpréteurs écrits en Java

Pour ce qui est des deux mécanismes basés sur une machine Prolog implémentée en Java, cela n'est pas aussi évident qu'il n'y paraît et mérite une petite discussion.

Une première chose à constater est que le programme Java et le programme Prolog qui interagissent vont s'exécuter dans le même thread de la même machine virtuelle Java. De ce point de vue, ils vont partager le même espace mémoire, c'est-à-dire la même zone des méthodes, le même tas, la même pile, le même compteur de programme, et, si nécessaire, la même pile des méthodes natives. La pile Java va donc contenir des morceaux de code Java mêlés à des morceaux de code Prolog[30].

Mais qu'en est-il des structures de données internes de la mémoire de la machine Prolog ? Si ces structures de données, ainsi que la politique de gestion de la mémoire, sont encapsulées dans la machine Prolog, celle-ci pourra être considérée, depuis un programme Java, comme une boîte noire possédant une interface à travers laquelle elle pourra être invoquée.

C'est généralement le cas pour les interpréteurs Prolog implémentés en Java. Les points d'accès à un programme Prolog depuis un programme Java se réduisent à des appels de méthodes à travers une interface, généralement fournie par le fabriquant de l'interpréteur[31].

Le cas du compilateur de Joshua Engel est quelque peu différent. Ici, le programme Prolog est transformé en classes Java, avec pour résultat de rendre toute la structure interne de la mémoire de la machine Prolog visible depuis un programme Java[32].

Le problème du partage de mémoire entre programme Java et programme Prolog ne se réduit donc pas à un problème de partage "physique" de la mémoire[33], mais concerne également la visibilité des structures internes de l'interpréteur Prolog, avec comme effet d'augmenter le nombre de niveaux d'indirection entre les deux.

[29] Ou "des systèmes", en effet, rien n'oblige les deux processus à être sur la même machine. L'interpréteur Prolog pourrait, par exemple, se trouver sur un serveur distant.

[30] J'entends par "code Java" l'ensemble des instructions appartenant au programme Java et par "code Prolog" les instructions correspondant à l'exécution du programme Prolog, ces deux codes étant exécutés par la JVM.

[31] C'est le cas, par exemple, de tuProlog, GNUProlog en Java, Moksa Prolog, JIProlog...

[32] C'est d'ailleurs sur cela que je vais baser l'implémentation du mécanisme d'interopérabilité entre Java et Prolog utilisant ce compilateur, voir Chapitre 11.

[33] C'est-à-dire au partage de la mémoire au niveau de la JVM .

3.4.3 En résumé

Les tables 3.6 et 3.7 résument cette section sur le partage de mémoire entre Java et Prolog.

La première table (3.6) compare les deux stratégies de gestion de la mémoire, à savoir mémoire partagée ou non, du point de vue de la complexité des mécanismes devant être impliqués (changement de contexte, protection de mémoire...) et de la granularité du mécanisme d'interopérabilité en fonction de la stratégie utilisée.

Stratégie	Gestion	Granularité
Espace mémoire partagé	complexe	plus petite
Espaces mémoire séparés	simple	plus grande

TAB. 3.6 – Comparaison entre les stratégies de gestion de l'espace mémoire

La seconde table 3.7 résume ce qui a été dit plus haut à propos des différences entre un interpréteur Prolog écrit en Java et un compilateur Prolog vers JCode, du point de vue du partage de l'espace mémoire (zone des méthodes, tas, piles et compteur de programme) et du partage des structures de données internes.

Machine Prolog	Espace mémoire	Structures de données
Interpréteur Prolog écit en Java	partagé	non partagées
Compilateur Prolog vers JCode	partagé	partagées

TAB. 3.7 – Comparaison entre l'interpréteur Prolog et le compilateur JCode

Chapitre 4

Conversion de données

Je vais maintenant aborder le problème de la conversion des données entre Java et Prolog. Dans un premier temps, je vais introduire le problème en rappelant quelles sont les grandes différences entre Java et Prolog au point de vue des types de données. Ensuite, je proposerai deux approches pour résoudre le problème. Je terminerai ce chapitre en indiquant quelle est l'approche que j'ai choisie et les raisons de ce choix.

4.1 Le problème

Comme je l'ai déjà indiqué, les langages Java et Prolog possèdent des manières différentes de voir leurs données. Le langage Java est un langage fortement typé. C'est un langage typé car, chaque variable y est d'un type particulier, un de ceux représentés sur la figure 4.1 et donnés par la table 4.1. De plus, comme Java possède un mécanisme de vérification des types de données, il est dit *fortement* typé. Le langage Java permet de plus de manipuler des objets.

type	données représentées
byte	entiers 8 bits signés en complément à 2
short	entiers 16 bits signés en complément à 2
int	entiers 32 bits signés en complément à 2
long	entiers 64 bits signés en complément à 2
char	entiers 16 bits non signés pour représenter les caractères Unicode
float	nombres à virgule flottante 32 bits
double	nombres à virgule flottante 64 bits
boolean	valeur true ou false

TAB. 4.1 – Types de données du langage Java

Le langage Prolog, par contre, est un langage non typé : les objets manipulés par Prolog sont tous du même "type", ce sont des *termes*[1]. De nombreux interpréteurs Prolog fournissent toutefois des méta-prédicats permettant d'obtenir de l'information sur le type d'un terme, des prédicats

[1] Voir Chapitre 2 pour plus de détails.

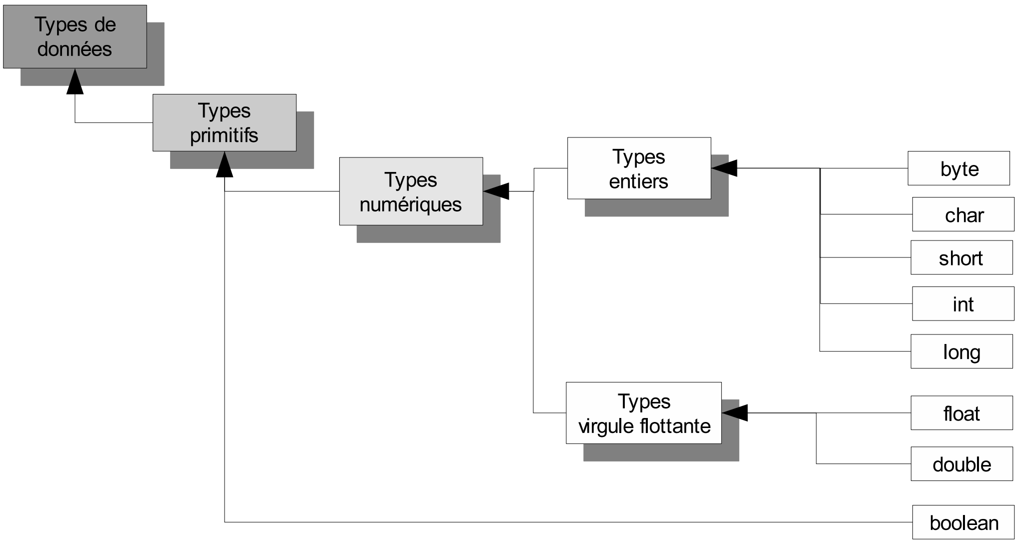

FIG. 4.1 – Types de données en Java

comme *integer/1*, *number/1*, *var/1*... C'est pour cela que l'on parle généralement de Prolog comme d'un langage *faiblement* typé[2]. La figure 4.2 donne les différents types de termes existants en Prolog.

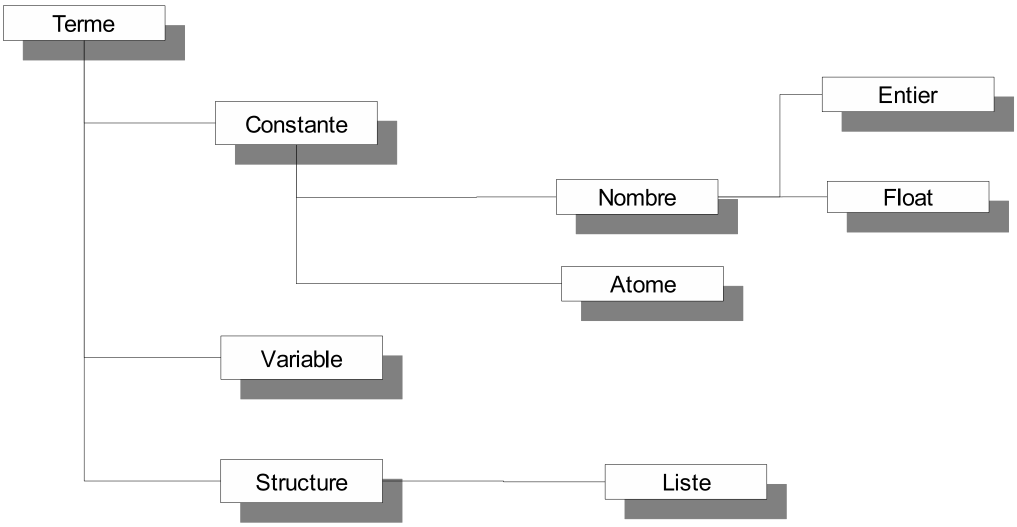

FIG. 4.2 – Structures de données en Prolog

L'un des problèmes à résoudre pour interfacer Java et Prolog va être de faire communiquer deux langages possédant des manières diamétralement opposées de voir les données qu'ils manipulent. Comment peut-on réaliser cela ?

4.2 Conversion de données entre Java et Prolog

La solution est assez simple : il va falloir effectuer une conversion entre les données de Prolog et les données de Java pour que, dans un sens, un programme Java puisse utiliser des données

[2]En prolog le typage est dynamique, alors qu'il est statique (effectué à la compilation) en Java.

générées par un programme Prolog et que, dans l'autre sens, un programme Prolog puisse utiliser les données qui lui sont fournies par un programme Java.

Les sous-sections suivantes décrivent deux approches pour régler le problème de la conversion de données entre Java et Prolog, et donnent les avantages et inconvénients de chacune d'entre elles.

4.2.1 Première approche : utilisation de classes Java existantes

La première approche est de convertir les structures *"natives"*[3] Prolog en classes des bibliothèques Java. Pour les structures de données qui n'ont pas d'équivalent Java, c'est-à-dire les variables logiques, les structures et les listes, il faudra implémenter des classes ad hoc. La table 4.2 donne l'équivalence entre structure de données Prolog et classes Java.

Structure de données Prolog	Classe Java
Atome	java.lang.String
Entier	java.lang.Integer
Float	java.lang.Float
Variable	PrologVar
Structure	PrologStructure
Liste	PrologList

TAB. 4.2 – Première approche : utilisation de classes Java existantes

Avantages

- l'utilisation de classes Java existantes facilite l'utilisation et l'apprentissage du mécanisme d'interopérabilité;
- il n'y a qu'un niveau de conversion, ce qui rend l'approche plus performante;
- cette approche est plus simple à utiliser côté Java puisqu'elle fait appel à des classes pour la plupart bien connues.

Inconvénients

- on perd de l'information sur les structures Prolog originales. Par exemple un entier, représenté par une instance de *java.lang.Integer*, n'est plus un terme;
- la conversion des types Java vers les types Prolog est moins aisée;
- le mécanisme de conversion de Prolog vers Java sera plus complexe à implémenter;
- cette approche n'est pas très élégante.

4.2.2 Seconde approche : créer des classes Java pour représenter les objets Prolog

La seconde approche consiste à implémenter une hiérarchie de classes (voir figure 4.3) Java pour représenter les structures de données Prolog. La conversion se déroulera alors en deux étapes :

1. le mécanisme d'interopérabilité entre Java et Prolog convertit les structures de données Prolog en classes manipulée par le mécanisme et vice versa;

[3] C'est-à-dire les différentes sortes de termes Prolog.

2. des méthodes de conversion permettent de transformer les classes du mécanisme d'interopérabilité en classes manipulables par un programme Java (Integer, Float, String...) et inversement.

De cette façon, on préserve les structures de données Prolog, tous les objets sont des termes, dans le programme Java, grâce aux propriétés d'héritage et de polymorphisme de la programmation orientée-objet. Dans un souci de transparence, seules les conversions entre types de données Java et termes Prolog ne seront visibles pour l'utilisateur, les conversions des structures de données de l'interpréteur Prolog vers les abstractions manipulées par le mécanisme d'interopérabilité seront encapsulées dans le mécanisme lui-même.

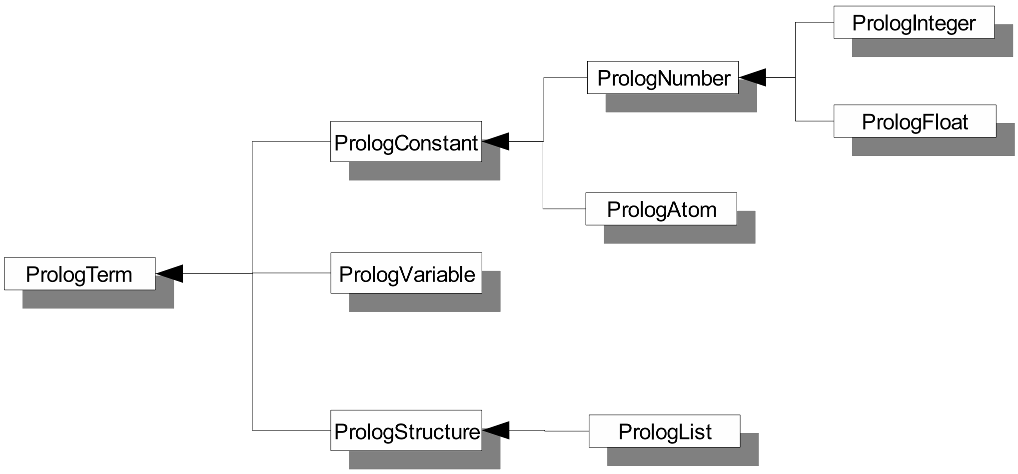

FIG. 4.3 – Représentation des données de Prolog en Java

Avantages

- préserve les structures de données Prolog;
- la conversion est plus aisée puisque Prolog et Java manipulent des objets semblables;
- plus élégante grâce à l'utilisation de la programmation orientée-objets;
- permet de manipuler les structures de données Prolog dans un programme Java;
- utilisation possible de l'héritage et du polymorphisme;
- possibilité de réduire les classes à des 'wrappers'[4] encapsulant des structures Prolog de l'interpréteur utilisé.

Inconvénients

- on a ajouté un niveau de conversion, ce qui résultera en une perte de performances;
- une nouvelle API devra être étudiée par l'utilisateur.

4.2.3 Solution choisie

En comparant les avantages et inconvénients de ces deux approches, ma préférence va à la seconde qui bien que moins performante, utilise réellement les possibilités de la programmation orientée-objet et, surtout, préserve les termes Prolog.

Du fait des classes Java qui seront utilisée pour représenter les constantes Prolog, les types Java qui pourront être échangés entre Prolog et Java et être manipulés par un programme Java seront limités à *int* (ou *java.lang.Integer*), *float* (ou java.lang.Float) et *java.lang.String*.

[4] C'est-à-dire des emballages.

Deuxième partie

Définition et spécification d'un mécanisme d'interopérabilité basé sur la machine virtuelle de Java

Chapitre 5

Démarche et remarques préliminaires

Dans ce premier chapitre, je vais introduire ma démarche et décrire l'organisation de la définition et de la spécification d'un mécanisme d'interopérabilité basé sur la machine virtuelle de Java.

Le mécanisme d'interopérabilité que je vais décrire et spécifier dans les chapitres suivants se divise en trois parties : un mécanisme permettant de représenter les termes du langage Prolog en Java et de récupérer les données qu'il contiennent, de manière à pouvoir les manipuler dans un programme Java, un mécanisme permettant à un programme Java d'appeler un programme Prolog, et, enfin, un mécanisme permettant à un programme Prolog d'invoquer des méthodes écrites en langage Java.

Les chapitres qui suivent présentent ces trois mécanismes sans entrer dans les détails d'implémentation qui seront abordés dans la partie III de ce mémoire. Dans ce qui suit, les classes sont décrites comme des classes au sens UML. Le fait qu'elles soient implémentées en Java par des interfaces, des classes abstraites ou des classes concrètes n'entre absolument pas en ligne de compte pour décrire et spécifier le mécanisme d'interopérabilité. Le Chapitre 6 décrit la représentation des termes Prolog en Java. Le Chapitre 7 s'intéresse au mécanisme d'interopérabilité de Java vers Prolog proprement dit. Enfin, le Chapitre 8 traite du mécanisme permettant à un programme Prolog d'invoquer des méthodes Java.

Mon objectif en procédant de cette manière est de rester dans l'optique de fournir un mécanisme d'interopérabilité générique et transparent. Tout ce qu'un programmeur a besoin de savoir pour l'utiliser se trouve dans les chapitres qui suivent. Les choix et détails d'implémentation seront discutés dans la partie III de ce mémoire.

Les chapitres qui suivent ne contiennent pas la spécification complète de chaque méthode, ce qui aurait fortement alourdi le texte. Celle-ci est donnée en annexe.

Chapitre 6

Spécification des structures de données représentant les termes Prolog

---

Dans ce chapitre, je vais décrire les classes Java représentant les termes Prolog et spécifier leurs méthodes. Il s'agit uniquement ici de donner les spécifications des structures de données sans s'intéresser aux choix d'implémentation. Ceux-ci seront abordés au Chapitre 10.

6.1 Description générale

La représentation des termes Prolog en Java, décrite au Chapitre 4 et donnée par la figure 6.1, forme une hiérarchie de classes dont la racine est *PrologTerm*.

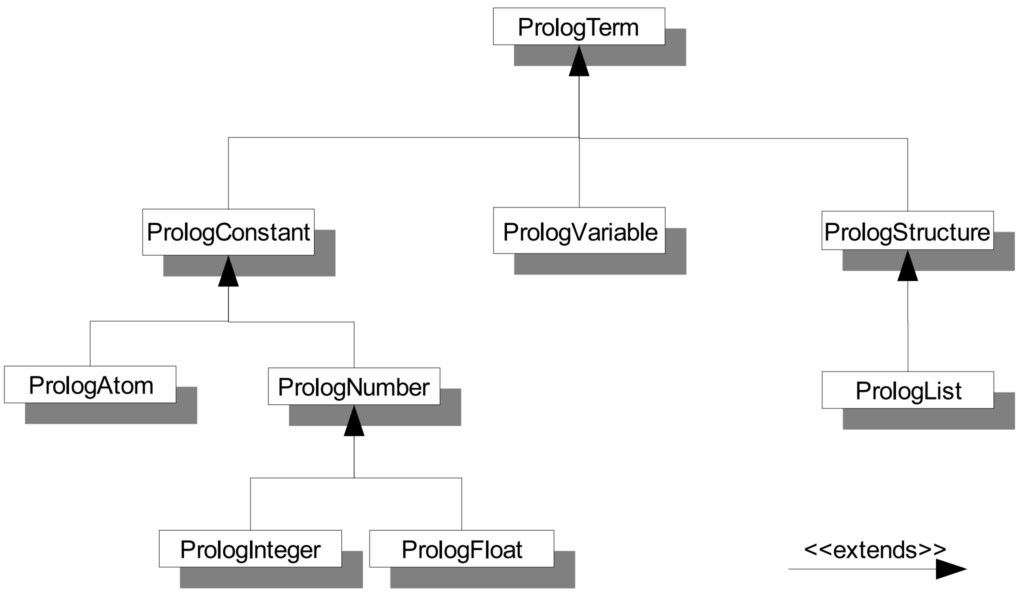

FIG. 6.1 – Diagramme des classes : structures de données Prolog

Ces classes fournissent différentes méthodes permettant la conversion de données, l'obtention d'informations sur le terme... Chacune de ces classes et méthodes va être décrite et spécifiée dans ce Chapitre.

Les seuls types Java qui pourront être échangés entre Java et Prolog seront *int* (ou *java.lang.Integer*), *float* (ou *java.lang.Float* et *java.lang.String*. La table 6.1 donne la conversion entre les classes du mécanisme d'interopérabilité et les types Java. Il faut également ajouter à cela la classe abstraite *java.lang.Number* qui correspond à la classe *PrologNumber*.

Terme	Type Java
PrologAtom	java.lang.String
PrologInteger	int ou java.lang.Integer
PrologFloat	float ou java.lang.Float

TAB. 6.1 – Conversion entre termes Prolog et types Java pour permettre la manipulation au sein d'un programme Java

Cette table ne signifie pas qu'un *PrologAtom* sera une chaîne de caractères Java ou qu'un *PrologInteger* sera un entier primitif 32 bits Java. Ce qu'elle signifie c'est que les constantes Prolog peuvent avoir à être manipulées dans des programmes Java. Par exemple, on peut vouloir additionner deux constantes numériques entière Prolog. Comme il serait assez long et peu intéressant d'implémenter des méthodes dans les classes représentant les constantes Prolog pour effectuer ces manipulations, la solution va être de convertir ces classes en types primitifs ou classes Java qui implémentent déjà ces fonctionnalités. Ainsi, pour reprendre l'exemple de l'addition des deux constantes numériques cité ci-dessus, on pourra les convertir soit en entier 32 bits primitifs, soit en instances de java.lang.Integer, qui pourront alors être manipulées par un programme Java.

6.2 Classes, méthodes et exceptions

Intéressons-nous maintenant à la description des différentes classes du diagramme des classes donné par la figure 6.1 et spécifions leurs méthodes.

6.2.1 Classes

Je vais maintenant présenter une par une les différentes classes que je vais utiliser pour représenter les structures de données Prolog en Java et spécifier leurs méthodes.

Toutes les classes représentant les structures de données Prolog commencent par le mot 'Prolog', et ce pour éviter des conflits de nom avec d'autres classes de l'API Java2.

La classe PrologTerm

La classe *PrologTerm* sert à représenter un terme Prolog en Java. Elle représente un terme générique et constitue la racine du diagramme des classes de la représentation des structures de données Prolog en Java. Elle possède trois sous-classes directes : *PrologConstant*, *PrologVariable* et *PrologStructure*.

La seule méthode déclarée par la classe PrologTerm est :
public java.lang.String getStringRepresentation() qui retourne la représentation du terme sous forme de chaîne de caractères.

La classe PrologConstant extends PrologTerm

La classe *PrologConstant* permet de représenter une constante Prolog. Elle étend la classe *PrologTerm* et possède deux sous-classes directes : *PrologNumber* et *PrologAtom*.

Les méthodes déclarées par la classe PrologConstant sont :

- *public java.lang.Object getValue()* qui retourne la valeur stockée dans la constante Prolog sous forme d'un objet java.lang.Object. Si la constante est un atome, cette objet sera une chaîne de caractère java.lang.String; si c'est une constante numérique, la méthode retournera un objet java.lang.Number; si c'est une constante entière ou de type nombre à virgule flottante, la méthode retournera, respectivement, un entier java.lang.Integer ou un nombre à virgule flottante java.lang.Float;
- *public java.lang.String getStringRepresentation()* méthode héritée de *PrologTerm*. La représentation sous forme de chaîne de caractères d'une constante Prolog est la représentation sous forme de chaîne de caractères de la valeur de cette constante.

La classe PrologAtom extends PrologConstant

La classe *PrologAtom* représente un atome (caractère et chaînes de caractères) Prolog en Java. Elle étend *PrologConstant*.

Les méthodes déclarées par la classe PrologAtom sont :

- *public java.lang.Object getValue()* héritée de *PrologConstant*;
- *public java.lang.String getString()* qui retourne le contenu de l'atome sous forme d'une chaîne de caractères Java;
- *public java.lang.String getStringRepresentation()* héritée de *PrologTerm*.

La classe PrologNumber extends PrologConstant

La classe *PrologNumber* représente les constantes numériques Prolog en Java. Elle étend PrologConstant et possède deux sous-classes directes : *PrologInteger* et *PrologFloat*.

Les méthodes déclarées par la classe PrologNumber sont :

- *public java.lang.Object getValue()*, heritée de *PrologConstant*;
- *public java.lang.Number getNumber()*, qui retourne le contenu de la constante numérique Prolog sous forme d'un nombre (java.lang.Number) Java. Ce nombre sera un entier java.lang.Integer si le *prologNumber* est une constante numérique entière, ou un nombre à virgule flottante java.lang.Float s'il s'agit d'une constante numérique de type virgule flottante;
- *public java.lang.String getStringRepresentation()*, héritée de *PrologTerm*.

La classe PrologInteger extends PrologNumber

La classe *PrologInteger* représente une constante numérique de type entière (entier signé en complément à deux 32 bits) Prolog en Java. Elle étend la classe *PrologNumber*.

Les méthodes déclarées par la classe PrologInteger sont :

- *public java.lang.Object getValue()*, héritée de *PrologConstant*;
- *public java.lang.Number getNumber()*, héritée de *PrologNumber*;
- *public java.lang.Integer getInteger()*, qui retourne le contenu de la constante numérique entière Prolog sous forme d'un java.lang.Integer Java;
- *public int getPrimitiveInt()*, qui retourne le contenu de la constante numérique entière Prolog sous forme d'un entier primitif 32 bits (int) Java;
- *public java.lang.String getStringRepresentation()*, héritée de *PrologTerm*.

La classe PrologFloat extends PrologNumber

La classe *PrologFloat* représente une constante numérique de type nombre à virgule flottante 32 bits Prolog en Java. Elle étend *PrologNumber*.

Les méthodes déclarées par la classe PrologFloat sont :

- *public java.lang.Object getValue()*, héritée de *PrologConstant*;
- *public java.lang.Number getNumber()*, héritée de *PrologNumber*;
- *public java.lang.Float getFloat()*, qui retourne le contenu de la constante numérique à virgule flottante Prolog sous forme d'un java.lang.Float;
- *public float getPrimitiveFloat()*, qui retourne le contenu de la constante numérique à virgule flottante Prolog sous forme d'un nombre à virgule flottante primitif 32 bits (float) Java;
- *public java.lang.String getStringRepresentation()*, héritée de *PrologTerm*.

La classe PrologVariable extends PrologTerm

La classe *PrologVariable* représente une variable logique, pouvant être liée une seule fois, en Java. Elle possède deux attributs, son nom et sa valeur, et étend *PrologTerm*.

Pour rendre compte de la propriété des variables logiques de ne pouvoir être liées qu'une seule fois, la méthode permettant de lier une *PrologVariable* à un terme Prolog, *PrologTerm*, devra signaler une violation de cette propriété. Pour réaliser cela, mon choix s'est porté sur la génération d'une exception d'un type particulier : *BindException*.

Les méthodes déclarées par la classe PrologVariable sont :

- *public java.lang.String getName()*, retourne le nom de la variable logique;
- *public boolean isBound()*, indique si la variable est liée ou non;
- *public void setBinding(PrologTerm) throws BindException*, tente de lier la variable logique avec le terme passé en argument. Lance une exception de type *BindException* si la variable était déjà liée;
- *public PrologTerm getBinding()*, retourne le terme lié à la variable ou 'null' si la variable est non liée;
- *public java.lang.String getStringRepresentation()*, héritée de *PrologTerm*. Le format de représentation d'une variable logique sous forme de chaîne de caractères est $Nom = Valeur$, par exemple $X = jean$.

La classe PrologStructure extends PrologTerm

La classe *PrologStructure* représente un terme composé, ou structure, Prolog en Java. Une structure est caractérisée par son nom, son arité et ses arguments. La classe*PrologStructure* étend *PrologTerm* et possède une sous-classe directe : *PrologList*.

Les méthodes déclarées par la classe PrologStructure sont :

- *public java.lang.String getName()*, retourne le nom du terme composé;
- *public int getArity()*, retourne l'arité (le nombre d'arguments) du terme composé;
- *public java.lang.String getFunctor()*, retourne le foncteur représentant le terme composé sous la forme *nom'/'arité*, par exemple *"father/2"*;
- *public PrologTerm[] getArguments()*, retourne un tableau contenant les arguments de la structure;
- *public java.lang.String getStringRepresentation()*, héritée de *PrologTerm*.

La classe PrologList extends PrologStructure

La classe *PrologList* représente une liste Prolog, ou cons, en Java. Il s'agit d'un cas particulier de la classe *PrologStructure*, qu'elle étend, dont le nom est '.' et l'arité toujours égale à 2. Une liste Prolog contient deux arguments, sa tête - un terme Prolog - et sa queue - une liste Prolog. La liste vide, '[]', sera représentée par une liste dont la tête et la queue seront égales à 'null'. On aurait également pu choisir de représenter la liste vide par 'null', mais cela n'aurait pas été très judicieux. En effet, dans ce cas, on ne peut pas distinguer une variable logique contenant la liste vide d'une variable logique non liée : dans les deux cas *getBinding()* retournera 'null'. De plus, l'utilisation d'une liste spéciale pour représenter '[]' est plus élégante et plus dans le style orienté-objets.

Les méthodes déclarées par la classe PrologList sont :

- *public java.lang.String getName()*, héritée de *PrologStructure*, retourne le nom du terme composé, c'est-à-dire '.' pour une liste;
- *public int getArity()*, héritée de *PrologStructure*, retourne l'arité de la liste, c'est-à-dire 2;
- *public java.lang.String getFunctor()*, héritée de *PrologStructure*, retourne le foncteur représentant la liste, c'est-à-dire *'./2'*;
- *public PrologTerm[] getArguments()*, héritée de *PrologStructure*, retourne un tableau contenant la tête et la queue de la liste;
- *public PrologTerm getHead()*, retourne la tête de la liste ou 'null' si la liste est vide;
- *public PrologTerm getTail()*, retourne la queue de la liste ou 'null' si la liste est vide;
- *public PrologTerm[] toArray()*, retourne le contenu de la liste sous forme d'un tableau de PrologTerm, ou un tableau de taille 0 si la liste est vide;
- *public int size()*, retourne le nombre de termes contenus dans la liste, 0 si la liste est vide;
- *public boolean isEmpty()*, indique si la liste est vide;
- *public boolean hasNext()*, indique s'il existe un terme suivant dans la liste;
- *public PrologTerm next() throws NoMoreTermException*, retourne le terme suivant dans la liste ou lance une exception de type *NoMoreTermException* s'il n'y a plus de terme suivant dans la liste;
- *public void add(PrologTerm)*, ajoute le terme passé en argument en tête de la liste;
- *public PrologTerm remove() throws NoMoreTermException*, retourne et supprime le terme situé en tête de la liste ou lance une *NoMoreTermException* si la liste est vide;
- *public java.lang.String getStringRepresentation()*, héritée de *PrologTerm*. Le format de représentation d'une liste sous forme de chaîne de caractère est [], pour une liste vide, ou *[*Arg_1*,...,*Arg_n*]* pour une liste contenant n termes, par exemple *[papa, maman, fils, fille]*.

6.2.2 Exceptions

BindException extends java.lang.Exception

Exception générée lorsque l'on essaie de lier une variable logique déjà liée à une valeur. J'ai décidé d'en faire une exception vérifiée (checked exception, étendant java.lang.Exception et pas java.lang.RuntimeException) pour une meilleure gestion au sein des programmes Java et pour une plus grande robustesse.

NoMoreTermException extends java.lang.Exception

Exception générée lorsque l'on tente d'accéder à, ou de supprimer, un terme dans une liste vide.

6.3 La fabrique de termes Prolog

Jusqu'ici, nous avons vu quelles sont les classes permettant de représenter les termes Prolog dans le mécanisme d'interopérabilité. Il nous faut également pouvoir les instancier. Comme l'implémentation de la classe *PrologTerm* et de ses sous-classes va dépendre de l'interpréteur Prolog qui sera utilisé pour l'implémentation d'un mécanisme d'interopérabilité, de simples constructeurs ne peuvent suffire. Il va falloir utiliser un mécanisme d'instanciation qui permettra de choisir l'implémentation des termes à utiliser à l'exécution.

Une idée simple et efficace pour réaliser cela est d'utiliser un motif de conception : le motif fabrique.

6.3.1 Le motif fabrique

L'idée du motif de conception[1] fabrique est de cacher l'instanciation d'un objet à l'application qui le crée. Ce pattern s'applique dans la situation où l'on ne contraîtra l'implémentation d'une classe qu'à l'exécution.

L'idée est alors d'utiliser une méthode de construction, une fabrique, qui pourra, selon les arguments qui lui seront passés, choisir entre différentes implémentation d'une classe. La figure 6.2 donne le diagramme des classes du motif fabrique.

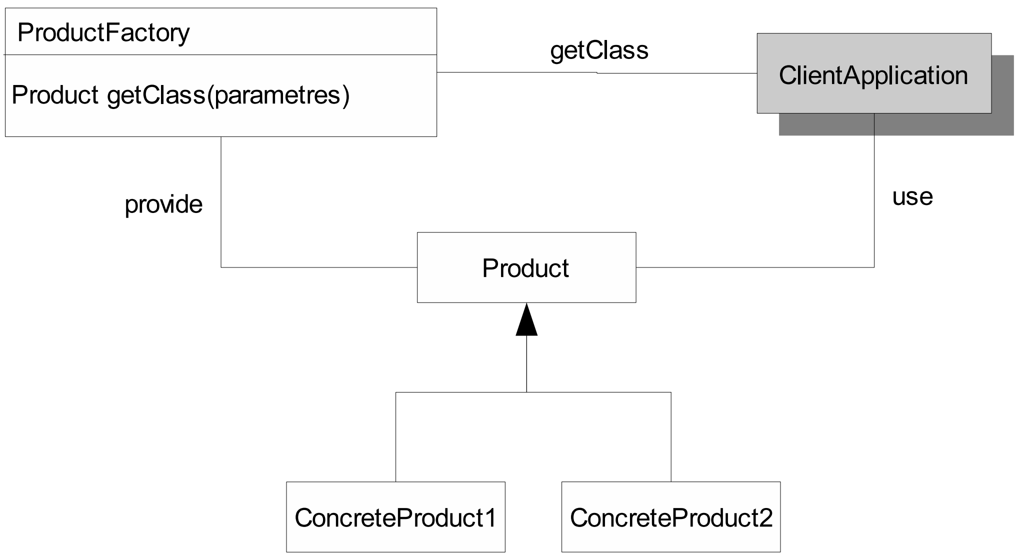

FIG. 6.2 – Motif fabrique

Les acteurs de ce diagramme sont :

- *Product* : un produit abstrait;
- *ConcreteProduct1* et *ConcreteProduct2* : deux implémentations du produit abstrait *Product*;
- *ProductFactory* : classe contenant la méthode fabrique *getClass()* qui permet, suivant les paramètres passés en argument, de créer une instance de *Product* correspondant à l'une des deux implémentations *ConcreteProduct1* et *ConcreteProduct2*;
- *ClientApplication* : une application qui demande à la fabrique une instanciation de la classe *Product*.

[1]Les lecteurs intéressés par plus de détails sur les motifs de conceptions sont renvoyés à [ea97].

6.3.2 Instanciation du motif pour l'instanciation des termes Prolog

Les méthodes fabrique de termes Prolog seront situées dans une classe *PrologTermFactory* qui devra être implémentée pour chaque interpréteur Prolog. Cette classe devra fournir des fabriques permettant d'instancier les sous-classes de *PrologTerm* à partir de l'implémentation des termes dans l'interpréteur Prolog utilisé, et des méthodes permettant de les instancier à partir de données Java.

Les méthodes fabriques fournies sont :

- *public PrologTerm convertTerm(java.lang.Object)*, permet de convertir un terme Prolog de l'interpréteur utilisé en un terme Prolog manipulé par le mécanisme d'interopérabilité;
- *public PrologConstant newConstant(java.lang.Object) throws InvalidArgumentException*, une fabrique de constantes Prolog. Si l'objet passé en argument est une chaîne de caractères, le terme retourné sera de type *PrologAtom*, s'il s'agit d'un entier java.lang.Integer, ce sera un *PrologInteger*, s'il s'agit d'un nombre à virgule flottante java.lang.Float, le terme retourné sera un *PrologFloat*;
- *public PrologInteger newPrologInteger(int)*, retourne une instance de *PrologInteger* à partir d'un entier primitif 32 bits Java;
- *public PrologFloat newPrologFloat(float)*, retourne une instance de *PrologFloat* à partir d'un nombre à virgule flottante primitif 32 bits Java;
- *public PrologStructure newStructure(String name, PrologTerm[] args)*, retourne une instance de structure Prolog *PrologStructure* à partir de son nom et de ses arguments;
- *public PrologList newList()*, retourne une liste Prolog vide. Cette liste pourra être remplie à l'aide de la méthode *add(PrologTerm)* de la classe *PrologList*.

Outre la méthode *convertTerm()* qui permet de convertir une variable logique utilisée par l'interpréteur Prolog en *PrologVariable*, la fabrique de termes ne fournit aucune autre méthode de création d'instance de *PrologVariable*, puisqu'un programme Java n'est pas censé créer de variable Prolog mais uniquement les manipuler.

6.3.3 Exceptions

InvalidArgumentException extends java.lang.Exception

Exception générée par la méthode *newConstant(java.lang.Object)* lorsque l'objet passé en argument ne peut pas être converti en une constante Prolog.

Chapitre 7

Mécanisme d'interopérabilité de Java vers Prolog

Dans ce chapitre, je vais spécifier le mécanisme permettant aux programmes Java de faire appel à des prédicats Prolog. Je vais dans un premier temps exposer l'idée générale du mécanisme d'interopérabilité. Je spécifierai ensuite ses différentes classes et méthodes. Il s'agit uniquement ici de donner une spécification sans s'intéresser aux choix d'implémentation. Ceux-ci seront abordés au Chapitre 11

7.1 Idée générale

L'idée de base du mécanisme d'interopérabilité de Java vers Prolog est de fournir une abstraction d'un interpréteur Prolog, qui sera la même quelle que soit l'implémentation sous-jacente de Prolog, et qui pourra être appelée depuis un programme Java. Cet interpréteur abstrait sera en réalité un emballage[1] pour un véritable interpréteur Prolog et fournira les méthodes nécessaires pour la gestion des théories, c'est-à-dire des programmes Prolog, et la résolution de requêtes. Il sera accompagné d'un ensemble de classes annexes permettant de représenter les clauses Prolog et les résultats de la résolution des requêtes.

Cette solution garantit la généricité et la transparence du mécanisme d'interopérabilité puisque quelle que soit l'implémentation de Prolog, l'interpréteur vu par le programme Java sera toujours le même. Par contre, cela va impliquer un plus gros effort de développement pour rendre compatibles l'interpréteur Prolog sous-jacent et les types de données qu'il fournit avec l'interpréteur abstrait. Il va également en résulter une perte de performances, puisqu'on introduit des niveaux d'indirection. Mais, c'est le prix à payer lorsque l'on veut obtenir un mécanisme générique et transparent.

7.2 Classes, méthodes et exceptions

Je vais maintenant décrire les différentes classes constituant le mécanisme d'interopérabilité de Java vers Prolog et spécifier leurs méthodes. Ici, de nouveau, je laisserai de côté les questions d'implémentation qui seront abordées au Chapitre 11.

[1] Ou *Wrapper*.

7.2.1 Classes utilitaires

Les classes utilitaires regroupent les classes qui vont permettre de représenter les clauses Prolog (règles, requête et faits) ainsi que celles qui stockent les résultats de la résolution des requêtes par l'interpréteur Prolog abstrait.

Le nom de chacune des classes utilitaires commence par le mot 'Prolog' pour éviter les conflits de nom avec d'autres classes Java, par exemple un conflit entre le result set utilisé par le mécanisme d'interopérabilité et celui de l'API JDBC (Java DataBase Connectivity).

Classes pour représenter les clauses

Ce qui suit constitue une description des classes représentant les clauses Prolog. Les critères de validité[2] des clauses ne sont pas décrits ici et font partie de choix d'implémentation du mécanisme d'interopérabilité. Le mécanisme de vérification de ces critères, s'il y en a un, sera souvent intimement lié à l'implémentation de Prolog sous-jacente au mécanisme. Les causes de non validité des clauses pourront être des erreurs de syntaxe, des problèmes d'accès à des fichiers...

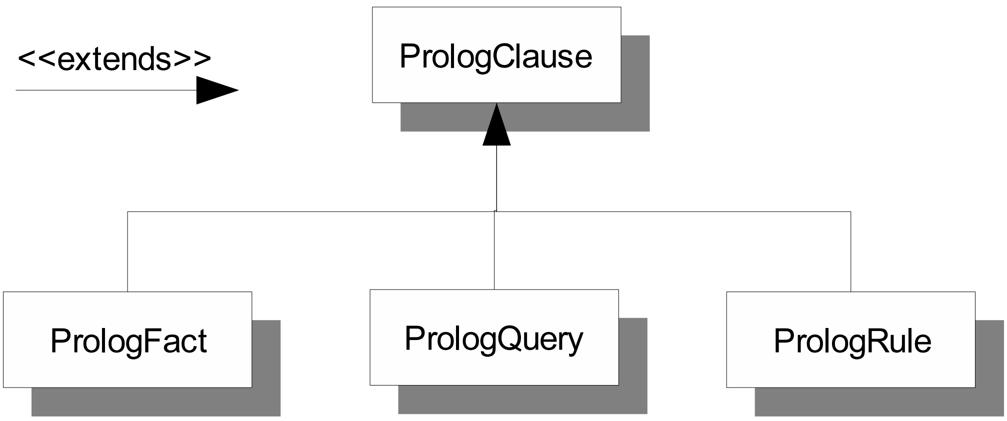

FIG. 7.1 – Diagramme des classes : Clauses Prolog

La raison pour laquelle j'ai décidé de représenter les clauses par des classes est de permettre d'utiliser les spécificités propres à leur implémentation pour chaque interpréteur Prolog.

Classes pour stocker les résultats des requêtes

J'ai décidé de définir deux classes pour stocker les résultats de l'évaluation des requêtes Prolog. Il s'agit de *PrologResult*, qui contient une solution de l'évaluation d'une requête, et *PrologResultSet*, qui contient toutes les solutions de l'évaluation d'une requête.

Mais qu'entend-on exactement par "résultat" ? Un résultat sera constitué par l'ensemble des variables non liées de la requête qui auront été liées lors de sa résolution, c'est ce que l'on appelle une substitution. Comme certaines requêtes, par exemple *frere(paul, jean).*, ne contiennent aucune variable non liée, il faudra également un mécanisme, tant pour *PrologResult* que pour *PrologResultSet*, permettant de savoir si la requête a échoué ou non. Cela est réalisé à travers deux méthodes, *isEmpty()* et *hasSolution()*, qui sont décrites dans ce qui suit.

[2] Il s'agit ici de validité au sens de savoir quelles sont les clauses qui vont être acceptées par le mécanisme d'interopérabilité et lesquelles seront refusées.

Classe PrologResult :
La classe *PrologResult* permet d'indiquer qu'une solution a été trouvée et, éventuellement, de stocker les variables qui ont été liées lors de l'évaluation d'une requête par l'interpréteur Prolog abstrait. Un résultat est nul s'il est vide[3] (i.e. ne contient pas de variables liées lors de l'évaluation de la requête) et que la requête qui l'a généré a échoué[4].

Methodes déclarées par la classe *PrologResult* :

- *public boolean isEmpty()*, indique si le résultat contient une substitution ou non;
- *public boolean hasSolution()*, indique si la résolution de la réquête ayant généré le résultat a réussi ou non;
- *public PrologVariable[] getSubstitution()*, retourne la susbtitution (les variables qui ont été liées lors de l'évaluation de la requête et la valeur à laquelle elles ont été liées) dans un tableau de *PrologVariable*, ou un tableau de taille 0 si la susbtitution est vide.

Un petit exemple d'utilisation d'un *PrologResult* est donné par la figure 7.2.

```
public static void printResult(PrologResult r)
{
  // si le résultat est vide
  if (r.isEmpty())
  {
    String hasresult;
    // si le résultat est une solution
    // afficher Yes, sinon, afficher No
    hasresult = r.hasSolution() ? "Yes" : "No";
    System.out.print(hasresult);
  }
  else
  {
    PrologVariable[] vars;
    // on extrait la substitution
    vars = r.getSubstitution();
    int n = vars.length;
    // une solution a été trouvée
    System.out.println("Yes");
    // on affiche la substitution
    for (int i = 0; i < n; ++i)
    {
      System.out.print(
        vars[i].getStringRepresentation());
      if (!(i == n - 1))
      {
        System.out.print(", ");
      }
    }
  }
}
```

FIG. 7.2 – Méthode affichant le contenu d'un PrologResult

Classe PrologResultSet :
La classe *PrologResultSet* permet d'indiquer qu'au moins une solution a été trouvée et, éven-

[3]Peut être vérifié par un appel à la méthode *isEmpty()*.
[4]Peut être vérifié par un appel à la méthode *hasSolution()*.

tuellement, de stocker toutes les solutions générées lors de l'évaluation d'une requête par l'interpréteur Prolog abstrait. Un result set est nul s'il est vide[5] (i.e. ne contient pas de résultat) et que la requête qui l'a généré a échoué[6].

Méthodes déclarées par la classe *PrologResultSet* :

- *public boolean isEmpty()*, indique si le *PrologResultSet* contient au moins un résultat ou non;
- *public boolean hasSolution()*, indique si la résolution de la réquête ayant généré le *PrologResultSet* a réussi ou non;
- *public int size()*, retourne la taille du *PrologResultSet*, c'est-à-dire le nombre de *PrologResult* qu'il contient;
- *public boolean hasNext()*, indique si le *PrologResultSet* contient un *PrologResult* suivant ou non;
- *public PrologResult next() throws NoMoreResultException*, retourne le *PrologResult* suivant contenu dans le *PrologResultSet* ou génère une exception de type NoMoreResultException si le *PrologResultSet* ne contient pas de résultat suivant;
- *public PrologResult[] toArray()*, retourne un tableau contenant tous les résultats du *PrologResultSet*, un tableau de taille 0 s'il est vide.

Un petit exemple d'utilisation d'un *PrologResultSet* est donné par la figure 7.3.

```
public static void printResultSet(PrologResultSet r)
  throws NoMoreResultException
{
  if (r.isEmpty())
  {
    String hasresult;
    // si le PrologResultSet est une solution
    // afficher Yes, sinon, afficher No
    hasresult = r.hasSolution() ? "Yes" : "No";
    System.out.println(hasresult);
  }
  else
  {
    // tant qu'il y a un résultat
    // dans le PrologResultSet
    while (r.hasNext())
    {
      // on extrait le résultat
      PrologResult re = r.next();
      // on affiche le résultat
      printResult(re);
      System.out.println();
    }
  }
}
```

FIG. 7.3 – Méthode affichant le contenu d'un PrologResultSet

7.2.2 L'interpréteur abstrait

L'interpréteur abstrait est, comme je l'ai déjà dit, une abstraction d'un interpréteur Prolog cachant l'implémentation de Prolog sous-jacente. Il fournit des méthodes de méta-programmation

[5] Peut être vérifié par un appel à la méthode *isEmpty()*.

[6] Peut être vérifié par un appel à la méthode *hasSolution()*.

et des méthodes de résolution de requêtes.

Méthodes de méta-programmation

Les méthodes de méta-programmation fournies par l'interpréteur abstrait permettent de charger un programme Prolog et de le modifier en lui ajoutant, ou en lui retirant, des règles et des faits.

Ces méthodes sont :

- *public void consult(PrologTheory) throws InvalidTheoryException*, charge[7] la théorie (le programme) passé en argument dans l'interpréteur;
- *public void assertz(PrologClause) throws InvalidClauseException*, ajoute la clause passée en argument à la théorie chargée dans l'interpréteur Prolog. Aucune vérification n'est garantie pour assurer la cohérence de la théorie après l'ajout de la clause. Cette méthode est suggérée mais pas obligatoire, elle n'est pas nécessairement supportée par tous les interpréteurs Prolog
- *public void retract(PrologClause) throws InvalidClauseException*, rétracte (supprime) la clause passée en argument de la théorie chargée dans l'interpréteur. Aucune vérification n'est garantie pour assurer la cohérence de la théorie après révocation de la clause. Cette méthode est suggérée mais pas obligatoire, elle n'est pas nécessairement supportée par tous les interpréteurs Prolog.

Méthodes de résolution de requêtes

Les méthodes de résolution de requête permettent, comme leur nom l'indique, de trouver les solutions à une requête donnée. Pour cela, l'interpréteur abstrait fournit deux types de mécanismes : le premier permettant d'obtenir les solutions d'une requête une par une, le second permettant de les obtenir toutes en une seule fois.

Pour plus de facilité, l'interpréteur abstrait ne comporte aucun mécanisme permettant de détecter si un prédicat Prolog possède 0, 1, plusieurs ou une infinité de solutions. Un tel mécanisme est en effet assez compliqué à mettre en oeuvre et sort du cadre de ce mémoire. Ce sera donc à l'utilisateur du mécanisme d'interopérabilité de prendre garde à ce problème.

Les méthodes de résolution de requêtes sont :

- *public PrologResult find(PrologQuery) throws InvalidQueryException*, retourne la première solution de l'évaluation de la requête passée en argument. Cette requête deviendra alors la requête courante pour l'interpréteur Prolog abstrait;
- *public PrologResult findNext() throws QueryNotInitializedException*, retourne la solution suivante de l'exécution de la requête courante. Si aucune requête n'a été préalablement évaluée par un appel à *find(Query)*, une exception de type QueryNotInitializedException est générée;
- *public PrologResult findNext(PrologQuery) throws InvalidQueryException*, retourne une solution de l'évaluation de la requête passée en argument. Cette méthode est fournie pour les interpréteurs permettant l'exécution de plusieurs requêtes simultanées;
- *public PrologResultSet findAll(PrologQuery) throws InvalidQueryException*, retourne un *PrologResultSet* contenant toutes les solutions de l'évaluation de la requête passée en argument, ou un *PrologResultSet* nul en cas d'échec de la requête.

7.2.3 Exceptions

Pour les mêmes raisons qu'au Chapitre 6, j'ai décidé d'en faire des exceptions vérifiées (checked exceptions, étendant java.lang.Exception et pas java.lang.RuntimeException).

[7] En pratique, il peut s'gir d'une compilation...

InvalidTheoryException extends java.lang.Exception

Cette exception est générée lorsque la théorie chargée dans l'interpréteur abstrait n'est pas valide.

InvalidClauseException extends java.lang.Exception

Cette exception est générée lorsque la clause passée à l'interpréteur abstrait est non valide.

InvalidRuleException extends InvalidClauseException

Spécialisation de InvalidClauseException, cette exception est générée lorsque la règle passée à l'interpréteur abstrait est non valide.

InvalidFactException extends InvalidClauseException

Spécialisation de InvalidClauseException, cette exception est générée lorsque le fait passé à l'interpréteur abstrait est non valide.

InvalidQueryException extends InvalidClauseException

Spécialisation de InvalidClauseException, cette exception est générée lorsque la requête passée à l'interpréteur abstrait est non valide.

NoMoreResultException extends java.lang.Exception

Exception générée lorsque l'on essaye d'accéder au resultat suivant d'un *PrologResultSet* qui n'en contient plus.

QueryNotInitializedException extends java.lang.exception

Cette exception est générée lorsque la méthode *findNext()* de l'interpréteur abstrait est invoquée sans invocation préalable de *find(Query q)*.

7.3 La fabrique d'objets Prolog

Après avoir décrit les différentes classes et méthodes utilisées par le mécanisme d'interopérabilité de Java vers Prolog, il nous faut maintenant discuter de la manière dont ils vont pouvoir être instanciés.

Le problème est ici quel que peu différent de celui de l'instanciation des termes Prolog, puisqu'ici c'est non seulement l'implémentation des objets qui devront être instanciés mais aussi celle de la fabrique qui va permettre de le faire qui ne seront connu qu'au moment de l'exécution.

Comment réaliser cela en pratique ? A l'aide du motif de fabrique abstraite.

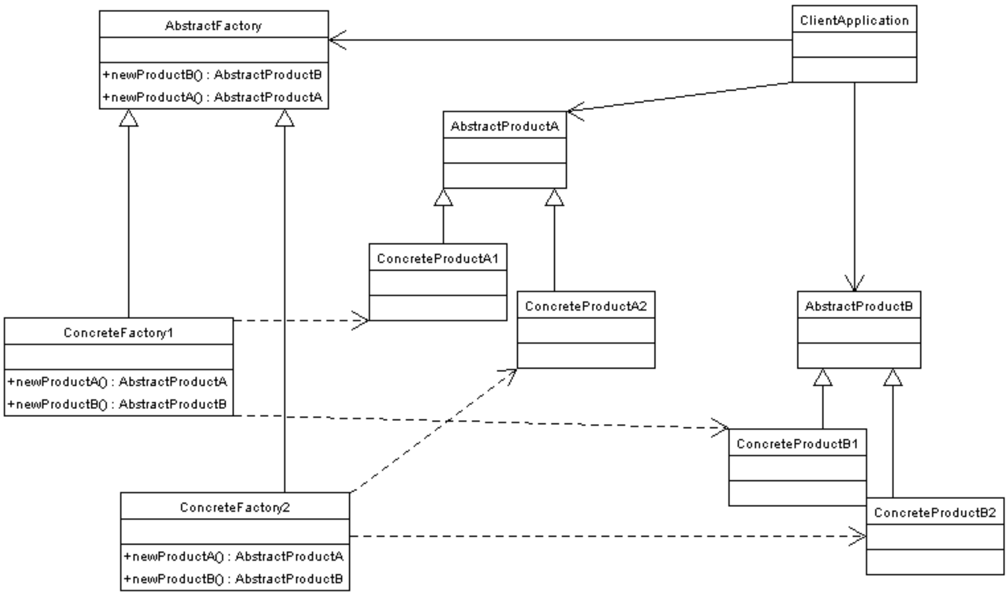

FIG. 7.4 – Le pattern Abstract Factory

7.3.1 Le motif de fabrique abstraite

Le motif de fabrique abstraite fournit un moyen de créer un objet obéissant à une interface donnée sans spécifier d'implémentation concrete pour la classe correspondante. Il fournit donc des fabriques d'objets qui ne peuvent être accédées que par leur interface. Cela permet de pouvoir manipuler dans un programmes des objets sans savoir, avant leur instanciation, quelle va être leur implémentation. Le diagramme des classes de ce motif est donné à la figure 7.4.

Les acteurs de ce motif sont :

- AbstractFactory : la fabrique abstraite fournit une interface constituée de méthodes fabriques de produits abstraits;
- AbstractProduct : le produit abstrait fourni par la fabrique abstraite;
- ConcreteFactory : la fabrique concrète implémente la fabrique abstraite et fournit des objets concrets, correspondants à une implémentation bien précise des produits abstraits;
- ConcreteProduct : l'implémentation du produit abstrait fournie par la fabrique concrète;
- Client : l'application cliente qui utilise la fabrique abstraite.

Instanciation du motif fabrique abstraite pour le mécanisme d'interopérabilité de Java vers Prolog

Le mécanisme d'interopérabilité de Java vers Prolog sera essentiellement une instanciation du motif de fabrique abstraite.

- la fabrique abstraite est implémentée par la classe *PrologProvider*;
- les produits abstraits sont les clauses Prolog (*PrologClause* et ses sous-classes *PrologRule*, *PrologQuery* et *PrologFact*), les théories (*PrologTheory*) et l'interpréteur Prolog abstrait (*PrologEngine*);
- les fabriques concrètes sont les implémentations de *PrologProvider* fournies pour chaque interpréteur Prolog;

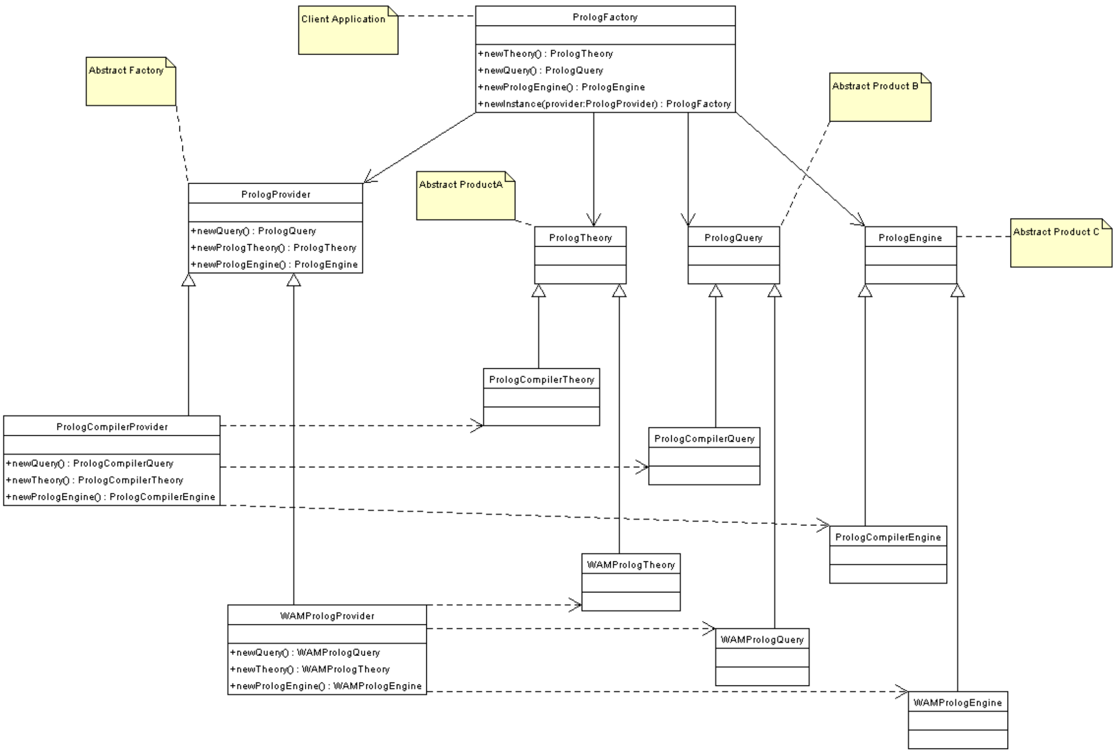

FIG. 7.5 – Instanciation du pattern Abstract Factory pour le mécanisme d'interopérabilité de Java vers Prolog

- les produits concrets sont les implémentations des produits abstraits pour chaque interpréteur Prolog;
- l'application cliente est une fabrique d'objets Prolog : *PrologFactory*.

7.3.2 Méthode fabrique de fabrique abstraite

L'instanciation de la fabrique abstraite *PrologPovider* se fera grâce à une méthode fabrique de la classe *PrologFactory*. Cette méthode prendra comme paramètre l'implémentation de la fabrique abstraite à utiliser.

L'instanciation se fera à l'aide de la méthode

- *public static PrologFactory newInstance(java.lang.String) throws UnknownProviderException* prenant comme argument le nom totalement qualifié de la fabrique concrète implémentant *PrologProvider* qui doit être utilisée, c'est-à-dire une nom du type *monpackage.MaClasse* où *monpackage* est le nom du package dans lequel se trouve la fabrique concrète à utiliser et *MaClasse* est le nom de cette fabrique.

Remarque : le chemin vers la classe *PrologProvider* passé en argument doit se trouver dans le chemin des classes (classpath) de la machine virtuelle Java.

L'exception UnknownProviderException

La fabrique de fabrique abstraite *PrologProvider* introduit donc un nouveau type d'exception, *UnknownProviderException*, spécialisant la classe *java.lang.exception*, qui indique que le *PrologProvider* correspondant au nom complètement qualifié passé en argument à la méthode *newInstance()* n'a pu être trouvé.

7.3.3 Méthodes fabriques d'objets Prolog

Les méthodes fabrique seront fournie par la fabrique abstraite *PrologProvider* et par ses implémentations. Elle permettront d'instancier les différentes classes que nous avons étudiées plus haut.

Fabriques de clauses

Pour plus de simplicité, j'ai décidé d'utiliser comme paramètre pour les fabriques de clauses une chaîne de caractères Java représentant cette clause. Cette solution est quelque peu limitative, mais elle a l'avantage d'être simple à implémenter et d'être adaptable à la majorité des interpréteurs Prolog, sinon tous.

Les méthodes fabrique à fournir pour les clauses sont :

- *public PrologRule newRule(java.lang.String) throws InvalidRuleException*, retourne un objet *PrologRule* créé à partir de la chaîne passée en argument. Celle-ci doit être de la forme $Head : -Body$ où $Head$, la tête de la règle, est une structure Prolog et *Body* une conjonction de structures Prolog constituant le corps de la règle. Un exemple de règle bien formatée est *father(X, Y):-male(X), parent(X,Y).*. Le point final peut-être omis[8];
- *public PrologFact newFact(java.lang.String) throws InvalidFactException*, retourne un objet *PrologFact* créé à partir de la chaîne passée en argument. Celle-ci doit être de la forme $name(argList).$, par exemple, *father(paul, jean).*. Le point final peut-être omis.;
- *public PrologQuery newQuery(java.lang.String) throws InvalidQueryException*, retourne un objet *PrologQuery* créé à partir de la chaîne passée en argument. Celle-ci doit être de la forme $? - Query$ ou *Query*, avec *Query* de la forme $nom(arglist).$. Par exemple, *?-frere(paul, jean).* est une requête bien formatée. Le point final peut-être omis;

[8]Il sera de toute façon ajouté ou supprimé selon l'implémentation de l'interpréteur Prolog qui sera utilisée par le mécanisme d'interopérabilité.

Fabriques de théories

Les théories Prolog pourront être construites à partir de différents objets : une chaîne de caractères (java.lang.String), un fichier java.io.File) ou un flux de données (java.io.InputStream). J'ai également décidé, pour plus de confort, de permettre de créer une théorie à partir d'un fichier en lui passant le nom du fichier sous forme de chaîne de caractères Java.

Les méthodes fabrique à fournir sont :

- *public PrologTheory newTheoryFromFile(java.lang.String) throws InvalidTheoryException*, retourne une instance de *PrologTheory* créée à partir du nom de fichier passé en argument;
- *public PrologTheory newTheoryFromFile(java.io.File) throws InvalidTheoryException*, retourne une instance de *PrologTheory* créée à partir du fichier passé en argument;
- *public PrologTheory newTheoryFromStream(java.io.InputStream) throws InvalidTheoryException*, retourne une instance de *PrologTheory* créée à partir du texte du programme passé en argument;
- *public PrologTheory newTheoryFromString(java.lang.String) throws InvalidTheoryException*, retourne une instance de *PrologTheory* créée à partir de la source de données (*InputStream*) passée en argument. Cette méthode est une généralisation de la méthode *newTheoryFromFile()* permettant d'utiliser n'importe quel flux d'entrée comme source d'un programme Prolog et, en particulier, l'entrée standard;

Fabrique de l'interpréteur abstrait

Puisqu'il n'y a qu'un seul interpréteur abstrait par implémentation de la fabrique abstraite, aucun paramètre n'est nécessaire à son instanciation.

La méthode fabrique de l'interpréteur abstrait est :

- *public PrologEngine newEngine()*, retourne une instance de l'interpréteur abstrait correspondant à l'implémentation de Prolog utilisée;

7.4 Un exemple d'utilisation

Un exemple d'utilisation est donné par la figure 7.6, dans lequel la gestion des exceptions a été omise pour ne pas le surcharger.

```
// instanciation de la fabrique d'objets Prolog
PrologFactory factory = PrologFactory.newInstance(
  "be.ac.ucl.info.blc.intero.compiler.PrologCompilerProvider");

// on crée une instance de l'interpréteur abstrait
PrologEngine interpret = factory.newEngine();

// on crée une théorie
PrologTheory theory = factory.newTheoryFromFile(
  "within.p");

// chargement de la théorie
interpret.consult(theory);

// on crée une instance de requête
PrologTheory query = factory.newQuery(
  "within(X, Y)");

// évaluation de la requête
// on veut les résultats un par un
PrologResult r = interpret.find(query);

doSomethingWith(r);

while (r.hasSolution())
{
    r = interpret.findNext();
    doSomethingWith(r);
}

// on crée une nouvelle instance de requête
query = factory.newQuery(
  "within(X, Y)");

// évaluation de la requête, cette fois on voudrait tous les
// résultats en une fois
PrologResultSet  rs = interpret.findAll(query);

while (rs.hasSolution() && rs.hasNext())
{
    doSomethingWith(rs.next());
}
```

FIG. 7.6 – Un exemple d'utilisation

Chapitre 8

Mécanisme d'interopérabilité de Prolog vers Java

Dans ce chapitre, je vais spécifier le mécanisme permettant aux programmes Prolog de faire appel à des méthodes Java. Je vais dans un premier temps exposer l'idée générale de ce mécanisme que les limitations que j'ai dû poser. Je donnerai ensuite quelques règles auxquelles les programmes Prolog utilisant le mécanisme d'interopérabilité pour invoquer des méthodes Java devront répondre. Je définirai ensuite le mécanisme proprement dit. Il s'agit uniquement ici de donner une description abstraite, sans s'intéresser aux choix d'implémentation. Ceux-ci seront abordés au Chapitre 12.

8.1 Idée générale

Je vais m'intéresser à la manière dont un programme Prolog va pouvoir invoquer une méthode écrite en Java, c'est-à-dire à la manière dont il peut importer un prédicat écrit en Java.

8.2 Règles et limitations

Les méthodes Java qui pourront être invoquées par un programme Prolog devront suivre des règles très strictes pour garantir le bon comportement du système.

La première limitation concerne le type de méthode pouvant être invoquée. On pourrait, et cela est réalisé par certaines interfaces entre Prolog et Java, autoriser un programme Prolog à invoquer n'importe quelle méthode Java, voire à créer et manipuler des objets Java. Ce ne sera pas le cas ici et je vais expliquer tout de suite pourquoi. Mais avant cela, voyons quels seraient les avantages de cette manière de procéder.

8.2.1 Arguments en faveur de l'invocation de n'importe quelle méthode Java depuis un programme Prolog

Le principal argument en faveur pour l'invocation de n'importe quelle méthode Java est le fait que l'on va pouvoir invoquer depuis un programme Prolog n'importe quelle fonction existante sans avoir à la réimplémenter. N'importe quel programme Java déjà existant pourra être appelé par n'importe quel programme Prolog. Si ce n'est pas le cas, il faudra emballer les programmes Java dans des méthodes pouvant être invoquées depuis un programme Prolog...

L'invocation de n'importe quelle méthode Java depuis un programme Prolog pourrait également permettre de créer des objets Java et de les manipuler dans des programmes Prolog.

8.2.2 Arguments contre l'invocation de n'importe quelle méthode Java depuis un programme Prolog

Pourquoi ne pas permettre d'invoquer n'importe quelle méthode ?

Si n'importe quelle méthode Java peut être invoquée depuis un programme Prolog, il est presque impossible de définir une manière standard pour effectuer le passage de paramètres et la récupération de la valeur de retour de cette méthode et cela demandera donc un gros travail de la part du mécanisme d'interopérabilité. Par exemple, certaines méthodes pourrait modifier l'état d'objets qui leur sont passés en argument, en plus de retourner une valeur.

Il faudra également trouver la classe contenant la méthode à invoquer, vérifier si cette méthode est statique ou non et, si besoin est, instancier la classe. Une solution pour réaliser cela est de fournir des prédicats prédéfinis permettant d'indiquer quelle méthode de quelle classe ou objet doit être invoquée et dans quelle variable placer le résultat, comme cela est fait pour certaines implémentations de Prolog[1].

Un autre problème concerne les types des arguments passés à une méthode Java et de sa valeur retournée. S'il s'agit d'un terme correspondant à un type primitif[2] Java ou à une chaîne de caractères, cela ne pose pas de problème. Mais comment faire lorsqu'il s'agit d'un tableau d'une classe ou d'un type défini par l'utilisateur ? Il faudra prévoir un mécanisme de conversion, a priori assez complexe, entre les termes Prolog et n'importe quelle classe ou type primitif Java ou interdire les invocations de méthodes prenant comme argument ou retournant un de ces types "non représentable trivialement" en Prolog.

Un autre problème est la gestion des exceptions, certaines méthodes génèrent des exceptions en cas d'erreur lors de leur invocation. Il faudrait donc définir un mécanisme pour gérer les erreurs des méthodes Java dans le programme Prolog.

Permettre à un programme Prolog d'invoquer n'importe quelle méthode Java pourrait violer la propriété de non mutabilité des variables logiques. En effet, si on peut interdire à un programme Java de modifier une variable logique déjà liée, comment être sûr qu'il ne va pas modifier l'état de cette variable en agissant sur son contenu. En fait, cela peut-être réalisé assez simplement en rendant la valeur stockée dans une variable logique finale. Ce problème ne se pose donc pas vraiment pour les invocations de méthodes. Par contre, il sera beaucoup plus important dans le cas de la création et la manipulation d'objets Java par un programme Prolog.

Les raisons pour interdire la création d'objets sont les suivantes :

1. la première raison est, tout simplement, que le langage Prolog ne connaît pas d'objets. Il ne connaît que des termes. En d'autres termes, Prolog correspond au paradigme de la programmation logique, alors que Java correspond à celui de la programmation orientée objets. Il pourrait être possible de représenter un objet Java par un terme Prolog, mais cela n'est pas trivial. Par exemple, comment gérer les problèmes d'héritage et de polymorphisme en Prolog ?
2. la deuxième raison est que les changements d'état des objets Java vont poser de nombreux problèmes dans les programmes Prolog. En effet, en Prolog, les variables son non-mutable : elles ne peuvent être liées qu'une seule fois et une fois liées leur valeur ne peut plus être modifiée. L'utilisation d'objets Java va à l'encontre de ce comportement attendu des variables Prolog puisque certains d'entre eux vont voir leur état changer lors de l'invocation de méthodes. On pourrait régler ce problème en exigeant que les prédicats Prolog modifiant les objets Java agissent sur des copies de ces objets et laissent sa version originale inchangée. Cela va résulter en une prolifération d'objets inutiles qui vont encombrer la mémoire de la JVM. Une autre solution est d'imposer que tous les objets Java manipulés par un programme Prolog soient non-mutables[3].

[1] Citons, par exemple, l'interface Jasper pour SICSTus Prolog, B-Prolog, Prolog Cafe ou JIPL.

[2] Ou d'un Wrapper pour un tel type (Float, Long, Int...).

[3] [Blo01] décrit plusieurs manières de forcer la non mutabilité des classes Java.

3. la troisième raison est une question de simplicité. En limitant les méthodes invocables, on facilite le fonctionnement du mécanisme d'interopérabilité, les appels aux méthodes de conversion...
4. enfin, cela sort quelque peu du cadre de ce mémoire et demanderait beaucoup plus de temps que ce que j'ai à ma disposition. Il ne s'agit plus ici de voir comment faire interopérer deux langages utilisant, l'un, le paradigme de la programmation orientée objets et, l'autre, le paradigme de la programmation logique, mais d'étudier la manière dont on pourrait réaliser un paradigme hybride regroupant les caractéristiques de la programmation logique et celles de la programmation orientée-objets.

8.2.3 Règles à suivre pour l'invocation de méthodes Java

Pour toutes ces raisons, je préfère limiter les méthodes pouvant être invoquées par un certain nombre de règles[4]. Les méthodes Java invocables depuis un programme Prolog devront impérativement suivre les règles suivantes et correspondre à la description qui sera donnée à la section 8.3.

Correspondance de signature

La correspondance entre signature Java et signature Prolog doit se conformer aux règles suivantes :

1. pour un prédicat Prolog de signature *nom(*arg_1*, ...,* arg_n*)*, la méthode Java invoquée aura pour signature *nomArité(PrologTerm[] args)*, où *args* est un tableau de termes de taille *n* contenant la représentation en Java des arguments du prédicat Prolog, placés dans le même ordre que ceux-ci. Par exemple, *fact(N, Z)* correspond à la signature Java *public boolean fact2(PrologTerm[] args)* où *args* contient deux *PrologTerm*, le premier est la représentation Java de *N*, le second celle de *Z*. La valeur booléenne retournée par la méthode indique si elle a réussi, auquel cas elle retournera 'true', ou échoué, auquel cas elle retournera false;
2. si la méthode doit retourner une valeur, alors cette valeur de retour sera stockée dans le **dernier argument** du prédicat Prolog correspondant, lequel devra être une variable logique non liée. Du côté Java, le dernier argument passé à la méthode sera une variable logique *PrologVariable* non liée qui sera liée à la valeur calculée. Par exemple, dans le prédicat *plus(N1, N2, N)*, *N* est une variable logique destinée à stocker le résultat de l'exécution de la méthode *public boolean plus3(PrologTerm[] args)*, où *args* sera de taille 3 et contiendra, dans l'ordre, deux *PrologInteger* et une *PrologVariable* non liée;
3. plus généralement, si plusieurs valeur doivent être retournée par la méthode, celles-ci seront stockées dans les derniers arguments du terme Prolog, et seront passé dans les arguments correspondant de la méthode Java sous la forme de variables logiques *PrologVariable* non liées;
4. si la méthode ne retourne aucune valeur, les arguments du prédicat correspondant seront, simplement, les arguments à passer à la méthode. Par exemple, *messagewindow(Message, Xpos, Ypos)*, un prédicat affichant une boîte de dialogue, invoquera la méthode Java *public boolean messagewindow3(PrologTerm[] args)* où *args* contiendra la représentation des arguments $Message$, $Xpos$ et $Ypos$.

J'ai défini ces règles pour garantir que le comportement attendu des méthodes Java invoquées sera respecté et pour simplifier la réalisation du mécanisme d'interopérabilité.

Pas de backtracking

Les méthodes Java appelée depuis Prolog doivent être déterministe. Ce choix est imposé dans le but de simplifier le mécanisme d'interopérabilité.

[4] Il ne s'agit pas ici de règles au sens Prolog, mais de règles d'implémentation.

Respect des types de données

Les arguments passés à une méthode Java par un programme Prolog doivent être d'un type compatible avec celui attendu par la méthode Java. De même les types retournés doivent correspondre. J'ai déjà discuté la correspondance entre termes Prolog et types/classes Java au Chapitre 6. Celle-ci est rappelée dans la table 6.1.

Par exemple, pour une méthode Java ayant comme signature *public boolean plus2(PrologTerm[] args)* où args attend comme données deux *PrologInteger* et une variable non liée dans laquelle stocker la valeur de retour de la méthode. Celle-ci correspond à la signature Prolog *plus(N1, N2, N)*, $N1$ et $N2$ seront des entiers Prolog - ou des variables liées contenant ces entiers Prolog, auquel cas seule la valeur liée à ces variables sera passée à la méthode Java - et N une variable logique non liée.

De plus les types de données pouvant être passé à une méthode Java invoquée par un programme Prolog sont limités à ceux représentables par les sous-classes de *PrologTerm* (voir Chapitre 6).

Respect de la non-mutabilité des variables logiques

La propriété de non mutabilité des variables logiques devra être respectées. Cela est imposé par les spécification de la classe *PrologVariable* décrite au Chapitre 6.

Pas de redéfinition des prédicats

Un prédicat Prolog correspondant à l'invocation d'une méthode Java ne pourra pas être redéfini dans le même programme Prolog et ce pour des raisons de cohérence. Par exemple, si la méthode *public void fact2(PrologTerm[] args)* est invoquée par le prédicat *fact(N, Z)* dans un programme Prolog, il est interdit de donner plus loin une nouvelle définition de ce prédicat *fact(N, Z)*, effectuant, par exemple, le calcul de la factorielle en utilisant des prédicats Prolog.

Pas d'instanciation ni de manipulation d'objets

Aucun objet Java ne pourra être créé ou manipulé directement par un programme Prolog pour les raisons que j'ai déjà exposées plus haut.

8.3 Description du mécanisme d'interopérabilité

Le mécanisme d'interopérabilité permettant à un programme Prolog d'invoquer des méthodes Java va être plus complexe à concevoir que le mécanisme inverse.

Bref la conception et, plus tard, la réalisation d'un mécanisme d'interopérabilité générique et transparent permettant à un programme Prolog d'invoquer des méthodes Java se trouve fortement compliquée. C'est pourtant ce que je vais tenter de faire ici.

L'idée que je vais développer est la suivante :

Tout d'abord, il faut un mécanisme permettant de signaler dans un programme Prolog qu'une méthode Java va être importée. Ceci peut être réalisé à l'aide d'une directive.
Ensuite, il nous faudra un moyen de détecter l'importation d'une méthode Java et de transformer cette méthode et le programme Prolog de manière à permettre l'invocation de la méthode. Ce rôle sera donné à un précompilateur.

Quant au problème des méthodes Java invocables, j'ai choisi d'obliger les méthodes Java appelées depuis Prolog à se trouver dans des classes bibliothèques qui devront étendre une ou plusieurs classes spéciales. Ces classes spéciales contiendront les éléments nécessaires à la conversion des données entre Prolog et Java.

Je ne vais pas décrire ici les détails du fonctionnement du mécanisme car ceux-ci vont être trop intimement liés à l'implémentation de l'interpréteur Prolog utilisé. Ils seront de ce fait décrits dans la Chapitre 12. Je vais décrire maintenant le mécanisme d'interopérabilité de Prolog vers Java en toute généralité, en insistant sur les éléments qui seront communs à toutes les implémentations.

8.3.1 Classes bibliothèques

Les méthodes Java pouvant être invoquées depuis un programme Prolog devront se trouver dans des classes bibliothèques étendant une classe *PrologLibrary*. Cette dernière contiendra tous les éléments nécessaires à la conversion des termes Prolog passés à la méthode par l'interpréteur Prolog en instances de la classe *PrologTerm* (voir Chapitre 6), à l'extraction des données contenues dans ces termes, à la création de nouveaux termes et à la conversion inverse des *PrologTerm* manipulée par la méthode Java vers les objets manipulés par l'interpréteur Prolog.

La conversion dans les deux sens des objets de l'interpréteur vers ceux manipulés par le mécanisme d'interopérabilité sera effectuée par la classe *PrologLibrary* elle-même. Ce que la (ou les) méthode(s) contenues dans cette classe va(vont) recevoir sera un tableau de *PrologTerm* prêt à être manipulé. Lors de cette conversion, les règles définies plus haut devront être respectées, en particulier l'ordre, la correspondance des types de données passés à la méthode Java et la conversion de signature entre Java et Prolog.

Pour permettre à la méthode invoquée de retourner la valeur calculée, la classe *PrologLibrary* contient un objet fournissant des méthodes pour convertir les types Java en instances de *PrologTerm*, nommons le $FACTORY$. Cet objet et ses méthodes seront décrits au Chapitre 10.

En ce qui concerne la récupération des données contenues dans les termes Prolog passés en argument à la méthode, elles pourront être récupérées grâce aux méthodes décrites au Chapitre 6.

Pour créer une nouvelle méthode Java pouvant être invoquée depuis un programme Prolog, il faut étendre la classe *PrologLibrary* et y placer le code méthode à appeler tout en respectant les règles vues précédemment. La figure 8.1 donne un exemple d'une telle bibliothèque.

La documentation de chaque méthode devra décrire ses conditions d'utilisation pour être sûr que les prédicats passés seront du bon type et respecteront bien l'usage attendu de la méthode.

8.3.2 Directive d'importation

L'importation d'une méthode Java dans un programme Prolog, laquelle méthode sera invoquée à travers un prédicat Prolog, sera signalée par une directive d'importation. Les directives déclarant les méthodes invoquées dans un programme Prolog seront situées, pour plus de facilité du point de vue des futures implémentations, dans un fichier séparé du code source.

Le rôle d'une directive d'importation est de faire le lien entre un prédicat invoquant une méthode Java utilisé dans un programme Prolog et la méthode Java invoquée. Cette directive sera de la forme *import(foncteur, bibliothèque)*, où $foncteur$ est le foncteur, au format *'nom/arité'*, identifiant le prédicat invoquant la méthode Java et *bibliothèque* le nom complètement qualifié de la classe Java contenant la fonction à invoquer. Le nom de la méthode à invoquer ne doit pas être indiqué dans cette directive puisqu'il peut être reconstruit à partir du foncteur du prédicat correspondant à la méthode.

Par exemple, la directive d'importation *import('fact/2', lib.Math)* indique que le prédicat de foncteur *'fact/2'* correspond à l'invocation une méthode Java $fact2$ située dans la bibliothèque *Math* du package *lib*.

```
package lib;

public class Math extends PrologLibrary
{
  /**
   * fact(N, Z)
   * N : entier positif
   * Z : variable
   * In : N - entier signé positif 32 bit
   * Out : Z - contient la valeur de N!
   */
  public boolean fact2(PrologTerm[] args)
  {
    // on extrait la valeur des arguments
    int n = ((PrologInteger) args[0]).getPrimitiveInt();

    // on calcule la valeur à retourner
    int z = fact(n);

    // on place la valeur de retour dans la variable
    // logique args[1]
    try
    {
      ((PrologVariable) args[1]).setBinding(
        FACTORY.newPrologInteger(1));
       return true;
    }
    catch(BindException e) // ne devrait pas se produire
    {
      // pour éviter de passer au programme Prolog
      // des valeurs erronées, on interrompt le programme
      // en cas d'erreur
      throw new RuntimeException(e.getMessage());
    }
  }

  /* autres méthodes... */

  // calcule la factorielle de l'entier passé en argument
  private static int fact(int n)
  {
    // pour éviter de passer au programme Prolog
    // des valeurs erronées, on interrompt le programme
    // en cas d'erreur
    if (n < 0)
     throw new RuntimeException(
       "entiers négatifs non autorisés dans la méthode fact");
    return (n == 0 || n == 1) ? 1 : (n * fact(n-1));
  }

  /* autres méthodes... */

}
```

FIG. 8.1 – Un exemple d'utilisation

8.3.3 Le précompilateur

Le précompilateur est un programme Java ayant pour rôle de transformer une source Prolog contenant des invocations de méthodes Java en un programme exécutable par un interpréteur Prolog donné, choisi pour l'implémentation du mécanisme d'interopérabilité.

Il prend en entrée les fichiers suivants :

- le fichier source du programme Prolog;
- le fichier de déclaration des directives d'importation de méthodes Java;
- les classes bibliothèques contenant les méthodes à invoquer, dont la liste sera déterminée à partir du texte des directives d'importation.

A partir de cela, il génère :

- un programme Prolog exécutable par l'interpréteur Prolog;
- des classes Java contenant des méthodes pouvant être invoquées depuis un programme Prolog par cet interpréteur.

Il s'agit donc d'un traducteur qui transforme les programmes Prolog contenant des appels relatifs aux directives d'importation en programmes Prolog exécutables par l'interpréteur, et les bibliothèques de méthodes dans le format du mécanisme d'interopérabilité en bibliothèques de méthodes utilisables par le compilateur Prolog.

8.3.4 Gestion des exceptions

La gestions des exceptions qui se produisent lors de l'invocation d'une méthode Java depuis Prolog doit être prise en charge par la méthode invoquée elle-même. J'ai fait ce choix par souci de simplicité et par manque de temps pour réaliser un mécanisme plus complexe.

Actuellement, si une erreur se produit dans une méthode Java invoquée par un programme Prolog, cela se traduira par la génération d'une *RuntimeException*, qui devra être interceptées sous peine de faire se terminer brusquement le programme.

Ce système n'est pas très évolué et un peu abrupt, mais le développement d'un système plus évolué n'aurait pas été possible dans le temps imparti pour ce mémoire.

Troisième partie

Implémentation d'un mécanisme d'interopérabilité basé sur la machine virtuelle de Java

Chapitre 9

Quelques remarques générales sur l'implémentation

Dans ce chapitre, je voudrais introduire quelques concepts communs aux différents problèmes d'implémentation qui se poserons dans la suite.

9.1 Choix et limitations

Dans ce qui suit, je vais présenter mes idées et choix d'implémentation pour les trois parties du mécanisme d'intropérabilité spécifié aux Chapitres 6, 7 et 8. Ensuite, je donnerai quelques détails à propos de l'implémentation du mécanisme avec le compilateur Prolog de Joshua Engel.

Pour ce qui est des limitations du mécanisme, j'ai décidé de ne considérer que des programmes Prolog du premier ordre avec utilisation de listes. Le mécanisme d'interopérabilité pourra toutefois être étendu par la suite pour inclure des fonctionnalités d'ordre supérieur. Ce choix est dû au fait que le compilateur Prolog de J. Engel est limité au premier ordre avec listes, et à un désir de simplifier l'implémentation et la conception du mécanisme d'interopérabilité proprement dit.

Les classes implémentant le mécanisme spécifié dans la partie II devront, de plus, respecter les règles et limitations qui y ont été définies précédemment comme le respect de la non mutabilité des variables logiques, les conventions de nom, la correspondance entre types de données et termes...

Pour des questions de sécurité et de robustesse, les constructeurs, ainsi que certaines méthodes, des classes implémentant le mécanicanisme d'interopérabilité devront avoir un accès privé ou limité au package dans lequel ces classes se trouvent. L'instanciation de ces classes doit alors obligatoirement passer par la fabrique de termes Prolog, *PrologTermFactory*, ou par la fabrique d'objets Prolog, *PrologFatory*.

9.2 Pourquoi utiliser le compilateur Prolog vers JCode

Le choix d'utiliser le compilateur Prolog vers JCode de Joshua Engel comme implémentation de Prolog, bien qu'il soit incomplet[1], est dicté par le fait que, de toutes les implémentations à ma disposition, c'est celle-là qui tire le mieux parti des possibilités de la machine virtuelle de Java.

[1] Il n'a d'ailleurs jamais été dans sa vocation de fournir un interpréteur Prolog complet, mais uniquement un exemple de langage compilé pour la machine virtuelle de Java.

Tout d'abord, la compilation des prédicats Prolog en fichiers classes Java permet d'obtenir des programmes Prolog persistant et ne nécessitant plus d'être chargé à chaque utilisation dans l'interpréteur, à moins de les avoir modifiés.

Un autre avantage provient du choix d'utiliser un assembleur de JCode plutôt que la transformation des prédicats en langage Java. En effet, dans le compilateur de J. Engel, les prédicats Prolog sont directement transformés en JCode, sans passer par le langage Java. Cela permet de simplifier fortement leur implémentation, en particulier le backtracking, en utilisant des instructions disponibles au niveau de la JVM et n'existant pas dans le langage Java. Ces instructions sont le *goto* et les autres instructions de saut. Je ne vais pas décrire ici la manière dont cela est implémenté. Le lecteur intéressé en trouvera une description précise au Chapitre 13 du livre de Joshua Engel, [Eng99].

Une autre raison, que j'ai déjà citée pour utiliser le compilateur de Joshua Engel, est que les structures internes de cet interpréteur sont accessibles depuis un programme Java. Cela simplifiera considérablement l'implémentation, puisque le mécanisme d'interopérabilité pourra les utiliser directement.

Enfin, ce compilateur est distribué sous licence GPL, ce qui me permet de pouvoir accéder à son code source pour le modifier.

Chapitre 10

Structures de données et mécanisme de conversion

Dans ce chapitre, je voudrais discuter de mes choix d'implémentation des structures de données Java représentant les termes Prolog telles qu'elles ont été spécifiées et décrites au Chapitre 6. Il s'agit ici de décrire mes propres choix et de donner des guides d'implémentation pour ceux qui voudraient implémenter le mécanisme d'interopérabilité avec un autre interpréteur Prolog que celui utilisé dans le cadre de ce mémoire.

10.1 Choix d'implémentation

Pour l'implémentation des classes représentant les structures de données Prolog en Java, trois solutions sont possibles. On peut utiliser des interfaces, des classes abstraites ou des classes concrètes. Pour pouvoir déterminer quelle(s) solution(s) choisir, je vais dans un premier temps m'intéresser aux avantages et inconvénients de chacune d'entre elles.

L'utilisation d'interface ou de classes abstraites permet une grande flexibilité. Elles permettent de tirer parti des spécificités des structures de données de chaque interpréteur Prolog et seront, de ce fait, plus simple à implémenter.
Ces interfaces(ou les classes abstraites) devront être implémentées pour chaque interpréteur Prolog. Il pourra le plus souvent s'agir de classes emballant les structures de données propres à chaque interpréteur. Des interfaces seront toujours préférées à des classes abstraites, à moins d'avoir à implémenter certaines méthodes pour garantir le comportement du mécanisme d'interopérabilité. Cette préférence donnée aux interface a pour but de permettre l'héritage multiple pour les classes implémentantes, ce qui, en Java, n'est possible qu'à travers des interfaces.

Le choix d'une implémentation concrète particulière possède plusieurs avantages. De telles classes sont plus simples à utiliser et seront plus performantes, puisqu'elles auront moins de degrés d'indirection. De telles classes permettront de plus de garantir le même comportement du mécanisme d'interopérabilité quelle que soit l'implémentation de Prolog sous-jacente.
Par contre, l'utilisation d'une implémentation donnée fait perdre les avantages que l'on pourrait tirer des spécificités de chaque interpréteur. De plus, il ne sera pas nécessairement possible de transformer les termes de l'interpréteur dans ces classes et il faudra également fournir, pour chaque implémentation de Prolog, des méthodes de conversion entre structures de données du mécanisme d'interopérabilité et structures de données de l'interpréteur utilisé. Un autre désavantage est la disparition de la possibilité d'utiliser l'héritage multiple.
Le choix de cette implémentation concrète particulière devra être bien médité de peur de créer des classes qui seront incompatibles avec certains interpréteurs Prolog.

J'ai décidé d'utiliser une solution mêlant interfaces et classes abstraites. La classe *PrologTerm* et les classes héritées de *PrologConstant* seront représentées par des interfaces pour permettre une plus grande liberté du point de vue de leur implémentation. Par exemple, certains interpréteurs[1] représentent toutes les constantes Prolog sous forme de chaînes de caractères. Dans ce cas, une seule classe concrète sera nécessaire pour représenter tous les types de constantes. Des interfaces sont dès lors nécessaires puisque le modèle de l'héritage en Java ne permet l'héritage multiple qu'à travers des interfaces.

Pour les autres structures de données, j'ai opté pour l'utilisation de classes abstraites car elles vont me permettre de définir certaines méthodes génériques (comme equals, hashcode...) et ainsi garantir un comportement identique pour chaque interprétation de Prolog.

10.2 Implémentations

Dans cette section, je voudrais discuter de la manière dont les différentes classes représentant les termes Java dans le mécanisme d'interopérabilité que j'ai défini dans la Partie II.
Pour cela, je vais détailler mon implémentation utilisant le compilateur Prolog vers JCode de Joshua Engel. Je donnerai ensuite quelques guides d'implémentation pour d'autres interpréteurs Prolog.

Les classes représentant les termes Prolog dans le mécanisme d'interopérabilité décrit dans ce mémoire se trouvent dans le package *be.ac.ucl.info.blc.interop.struct*.

10.2.1 Implémentation avec le compilateur de Joshua Engel

Les classes correspondant à l'implémentation avec le compilateur de Joshua Engel se trouvent dans le package *be.ac.ucl.info.blc.interop.compiler*. Les classes du compilateur lui-même se trouve dans *com.sootNsmoke.prolog*. Quant aux exceptions utilisées par le mécanisme d'interopérabilité, elles sont situées dans le package *be.ac.ucl.info.blc.interop.exception*.

Pour l'implémentation le compilateur de Joshua Engel, *PrologConstant* et toutes ses sous-classes sont implémentées par une seule et même classe avec le compilateur de Joshua Engel. Ce choix est dicté par le fait que ce compilateur représente toutes les constantes Prolog sous forme de chaînes de caractères. La centralisation de toutes les méthodes facilite grandement l'implémentation puisqu'il s'agira ici d'analyser une chaîne de caractères pour la convertir soit en un nombre, soit en un atome.

Les classes du mécanisme de conversion implémenté avec le compilateur Prolog de Engel sont, dans tous les cas où cela s'est révélé possible, des emballages pour les structures de données manipulées par le compilateur. Le seul cas où cela n'a pas été réalisable est celui des variables logiques. En effet, si le compilateur fournit bien une classe pour représenter les variables logiques, son utilisation au sein de la classe *PrologCompilerVariable* s'est révélée impossible à cause d'un choix d'implémentation du compilateur lui-même dans lequel les mêmes variables logique sont réutilisée à chaque calcul d'une solution. Ainsi, ma première version de l'implémentation, qui utilisait un emballage des variables du compilateur, retournait la même substitution pour toutes les solutions obtenues par l'appel à la méthode *findAll(PrologQuery)*. Pour règlé se problème, deux solution étaient envisageables : changer l'implémentation de la classe *PrologCompilerVariable* ou réimplémenter la partie du compilateur Prolog qui posait problème. Par manque de temps et par facilité, j'ai préféré changer l'implémentation de la variable logique qui contient maintenant une copie du nom et de la valeur de la variable du compilateur à partir de laquelle elle a été construite.

Une seule méthode de la fabrique de termes Prolog n'a pas été implémentée avec le compilateur de Joshua Engel. Il s'agit de la méthode *newStructure()*. La raison à cela est que le compilateur de Engel ne permet pas la manipulation des structures par les prédicats Prolog. Cette méthode va générer une exception de type *UnsupportedOperationException* si l'on tente de l'invoquer.

[1] C'est le cas du compilateur de Joshua Engel.

10.2.2 Guides d'implémentation

Dans la plupart des cas, les classes du mécanisme de conversion pourront être implémentées comme des emballages pour les structures de données utilisées par l'interpréteur Prolog lui-même. Cette façon de procédé simplifie l'implémentation et permet de tirer parti des méthodes fournies par ces structures de données.

Comme pour l'implémentation des constantes Prolog avec le compilateur de J. Engel, chaque classe du mécanisme de conversion de données ne doit pas nécessairement correspondre à une seule classe dans une implémentation particulière. Le choix sera directement dicté par les classes et structures de données effectivement disponibles pour l'interpréteur Prolog utilisé.

Si une méthode du mécanisme d'interopérabilité ne peut pas être implémentée avec l'interpréteur Prolog choisi, la meilleure solution est de fournir une méthode dont le corps se réduira à la génération d'une *UnsupportedOperationException*. Cela permet de ne pas renvoyer un résultat arbitraire[2] et qui pourrait créer des erreurs[3] dans la suite du programme. Cette exception étant une exception non vérifiée lancée à l'exécution[4], le fait qu'elle soit générée par une méthode **doit être documenté**.

[2]Par exemple 'null'.
[3]Par exemple, dans le cas de 'null', une NullPointerException.
[4]Elle étend java.lang.RuntimeException.

Chapitre 11

Mécanisme d'interopérabilité de Java vers Prolog

Je vais maintenant discuter mes choix d'implémentation pour le mécanisme d'interopérabilité de Java vers Prolog telle qu'il a été décrit au Chapitre 7. Je terminerai cette discussion par une description plus précise de mon implémentation utilisant le compilateur Prolog vers JCode de Joshua Engel, avant de donner des guides pour l'implémentation du mécanisme avec d'autres interpréteurs Prolog que les deux proposés ici.

11.1 Choix d'implémentation

Je vais décrire dans ce qui suit mes choix d'implémentation pour le mécanisme d'interopérabilité permettant à un programme Java de faire appel à un programme Prolog, et ce au travers de l'interpréteur abstrait décrit au Chapitre 7.

11.1.1 Clauses Prolog

J'ai décidé de représenter les clauses Prolog par des interfaces vides. Le choix de l'utilisation d'interfaces s'imposait de lui-même pour fournir une plus grande flexibilité d'implémentation. J'ai choisi d'utiliser des interfaces vides, c'est-à-dire sans méthodes, car aucune méthode n'est nécessaire pour ces interfaces qui n'ont de raison d'exister que pour de la vérification de type (type checking).

11.1.2 Théories

Le choix d'interfaces vides pour la classe *PrologTheory* a été dicté par les mêmes considérations que les clauses Prolog.

11.1.3 Résultats des requêtes

Les classes *PrologResult* et *PrologResultSet* sont réalisées par des interfaces contenant les méthodes décrites au Chapitre 7. Ces interfaces devront être implémentées pour chaque interpréteur Prolog.

11.1.4 L'interpréteur abstrait

Pour l'interpréteur abstrait, mon choix s'est également porté vers une interface pour pouvoir, comme précédemment, profiter d'une plus grande liberté d'implémentation.

11.1.5 La fabrique abstraite

L'interface *PrologProvider* fournit toutes les méthodes de fabrique pour les différentes classes décrites au Chapitre 7. Elle devra être implémentée en fonction de chaque interpréteur Prolog pour lequel on veut implémenter le mécanisme d'interopérabilité. On ne peut accéder directement aux méthodes de cette interface. Cela ne peut se faire qu'au travers de la classe *PrologFactory*. Cela permet de garantir que seule les méthodes décrites dans l'interface seront accessibles et, ainsi, garantir que le comportement de la fabrique abstraite sera le même pour tous les interpréteurs Prolog.

11.1.6 La fabrique d'objets Prolog

La fabrique d'objets Prolog *PrologFactory* est la seule classe concrète du mécanisme d'interopérabilité de Java vers Prolog. Elle ne peut pas être instanciée directement, tous ses constructeurs étant privés, mais peut être accédée à travers la méthode fabriques statique *newInstance(java.lang.String)* décrite au Chapitre 7.

La méthode *newInstance(java.lang.String)* utilise l'API d'introspection de Java pour trouver et instancier le *PrologProvider* devant être utilisé par la fabrique d'objets Prolog. Il s'agit de l'utilisation des méthodes *forName(java.lang.String)* et *newInstance()* de la classe *java.lang.Class* dans le code suivant :

```
PrologProvider p = (PrologProvider)
    Class.forName(provider_fully_qualified_name).newInstance();
```

11.2 Implémentations

Le but de cette section est de discuter de la manière dont les interfaces du mécanisme d'interopérabilité permettant à un programme Java d'appeler un programme Prolog peuvent être implémentées pour les différentes interpréteurs Prolog disponibles.
Pour cela, je vais détailler mon implémentation utilisant le compilateur Prolog de Joshua Engel avant de donner des guides permettant l'implémentation du mécanisme d'interopérabilité en utilisant d'autres interpréteurs Prolog.

Les classes de l'interpréteur abstrait, de la fabrique abstraite, de la fabrique d'objets Prolog, des clauses et des solutions des requêtes se trouvent dans le package *be.ac.ucl.info.blc.interop.engine*. Quant aux exceptions utilisées par le mécanisme d'interopérabilité, elles sont situées dans le package *be.ac.ucl.info.blc.interop.exception*.

11.2.1 Implémentation avec le compilateur Prolog de Engel

Le compilateur Prolog de Joshua Engel n'offre aucun mécanisme pour permettre l'interopérabilité avec Java. Ici, il m'a fallu modifier le code source même du compilateur ainsi que la visibilité de certaines méthodes et classes. J'en ai profité pour supprimer certains appels à des méthodes dépréciées et les remplacer. Enfin, j'ai ajouté quelques méthodes en vue simplifier la réalisation du mécanisme d'interopérabilité.

Un premier choix d'implémentation concerne les conventions de nommages. Toutes les classes implémentant le mécanisme d'interopérabilité de Java vers Prolog avec le compilateur de J.

Engel ont un nom commençant par 'PrologCompiler'. Par exemple, la fabrique concrete pour ce compilateur a pour nom *PrologCompilerProvider*.

Un autre choix d'implémentation, dans le cas du compilateur Prolog vers JCode, a été d'encapsulé le code permettant la résolution des requêtes et le chargement des théories dans les classes correspondantes, c'est-à-dire, respectivement, *PrologCompilerQuery* et *PrologCompilerTheory*. Ce choix a simplement été fait par soucis de simplicité d'implémentation.

Les seules méthodes qui n'ont pas été implémentées avec le compilateur de Joshua Engel sont les méthodes *asserts(PrologClause clause)* et *retract(PrologClause clause)*. Celles-ci nécessiteraient l'implémentation d'un mécanisme très compliqué du fait de l'implémentation du compilateur. En effet, celui-ci transforme les prédicats Prolog en fichiers classes Java. Pour modifier les programmes Prolog, il faudrait donc pouvoir accéder aux règles stockées dans les classes, ajouter ou supprimer le code correspondant à la clause que l'on veut ajouter ou rétracter, modifier la table des constantes... Une autre solution serait de faire les modifications sur le fichier source du programme Prolog lui-même et de le recompiler à chaque insertion ou suppression d'une clause. Comme ces deux méthodes ne sont pas essentielles, j'ai préféré ne pas me pencher sur leur implémentation, pour pouvoir m'intéresser à des questions plus importantes.

Les classes correspondant à l'implémentation du mécanisme d'interopérabilité de java vers Prolog basée sur le compilateur Prolog vers Jcode de J. Engel se trouvent dans le package *be.ac.ucl.info.blc.interop.compiler*. Le code source du compilateur lui-même se trouve dans *com.sootNsmoke.prolog*.

11.2.2 Guides d'implémentation

Dans ce qui suit, je vais donner quelques petites règles et conseils que devraient suivre les implémentations du mécanisme d'interopérabilité de Java vers Prolog décrit au Chapitre 7.

Premièrement, les méthodes des classes du mécanisme d'interopérabilité qui ne peuvent pas être implémentées avec l'interpréteur Prolog choisi. Dans ce cas, la meilleure solution est de fournir une méthode dont le corps se réduira à la génération d'une *UnsupportedOperationException*. Cette exception étant une exception non vérifiée lancée à l'exécution[1], le fait qu'elle est lancée par une méthode **doit être documenté**. Il en va d'ailleurs de même pour toutes les exceptions de ce type. C'est le cas par exemple de la méthode *findNext(PrologQuery)* de la classe *PrologEngine* tant pour l'implémentation avec le compilateur de Engel que pour celle utilisant tuProlog.

Ensuite, mais il s'agit ici plus d'un conseil général, l'utilisation de motifs de conception ou d'idiomes peut souvent résoudre les problèmes que l'on rencontre au cours de la conception d'un programme. Jeter un coup d'oeil à la documentation, en particulier [ea97], s'avère souvent payant.

[1]Elle étend java.lang.RuntimeException.

Chapitre 12

Mécanisme d'interopérabilité de Prolog vers Java

Je vais maintenant discuter mes choix d'implémentation et l'architecture du mécanisme d'interopérabilité de Prolog vers Java. Je terminerai cette discussion par une description plus précise de mon implémentation utilisant le compilateur de Joshua Engel.

12.1 Idées générales

Le mécanisme d'interopérabilité permettant à un programme Prolog d'invoquer des méthodes Java est plus complexe à concevoir que le mécanisme inverse.

La première raison à cela est que tous les interpréteurs Prolog ne fournissent pas de mécanismes permettant de le faire. Il faudra donc, dans certains cas, créer de toutes pièces un tel mécanisme et modifier certaines parties de l'interpréteur Prolog. Il se peut même que cela soit totalement impossible à réaliser avec certains interpréteurs.

Une autre raison est que chaque interpréteur fournissant un mécanisme permettant l'appel de méthodes Java depuis Prolog a une manière différente de procéder. Il ya des interpréteurs qui ajoutent des prédicats prédéfinis pour l'appel de fonctions externes (par exemple SISCTus Prolog), d'autres qui fournissent des classes java spéciales représentant des bibliothèques de fonctions invocables depuis Prolog (c'est le cas de tuProlog).

Bref la réalisation d'un mécanisme d'interopérabilité générique et transparent permettant à un programme Prolog d'invoquer des méthodes Java s'en trouve fortement compliquée.

12.2 Implémentation avec le compilateur Prolog vers JCode de Joshua Engel

Le temps m'a manqué pour implémenter le mécanisme d'interopérabilité de Prolog vers Java. J'ai, toutefois, une idée assez précise de la manière dont cela pourrait être réalisé. C'est cette idée que je vais décrire dans ce qui suit.

Le compilateur de Joshua Engel ne fournit aucun mécanisme tout fait permettant de créer des prédicats écrits en langage Java. Par contre, la manière dont il compile les programmes Prolog en fichiers classes Java va permettre de créer un tel mécanisme.

Premièrement, chaque clause constituant un prédicat[1] d'un programme Prolog sera compilé en une seule classe Java qui contiendra l'ensemble des règles et des faits de ce prédicat. Le nom d'une telle classe Java est toujours de la forme *nom_arité*, où *nom* est le nom du prédicat et *arité* son arité.

Ensuite, le point d'accès à cette classe est constitué par une méthode *call(Object[] args)* qui prend en argument les arguments à passer au prédicat et qui se charge de faire appel à un environnement d'exécution Prolog pour l'évaluation du prédicat.

Enfin, lorsque le compilateur de Engel compile un prédicat Prolog en fichier classe Java, il ne vérifie pas que les prédicats éventuellement appelés existent déjà[2].

Tout cela va nous pemettre d'envisager la réalisation d'un mécanisme permettant à un programme Prolog compilé pour le compilateur d'appeler des prédicats écrits en Java. Je n'ai pas eu le temps d'implémenter ce mécanisme, mais je vais tout de même vous exposer la manière dont je le conçois.

Pour cela, un tel prédicat devra être implémenter de la manière décrite au Chapitre 8. Il se trouvera donc dans une classe étendant une bibliothèque *PrologLibrary*. L'importation du prédicat sera déclarée dans un fichier séparé du code du programme.

L'idée est alors d'implémenter un précompilateur qui, à partir des directives d'importation fournies, va créer une nouvelle classe Java, nommée en suivant les conventions du compilateur de Joshua Engel. Cette méthode contiendra, dans sa méthode *call*, le code nécessaire à la conversion des termes du compilateur, pour leur utilisation par la méthode à appeler, et faisant appel à la méthode contenue dans la bibliothèque. Elle contiendra aussi le code permettant la conversion du ou des valeurs retournées par la méthodes Java en termes du compilateur.

Il nous faudra également quelques méthodes annexes :

-
- public int convert(Object[] args, PrologTerm[] term) , converti les termes du compiléteur en termes utilisables par le mécanisme d'interopérabilité et retourne le nombre de variables non liées se trouvant parmi ces termes[3];
-
- public void bound(Object[] args, PrologTerm[] term, int n), lie les n variables non liées de *args* aux n valeurs correspondantes de *term*.

Le précompilateur pourrait avoir un modèle de classe Java pour les prédicats ecrits en Java qu'il pourra compléter avec les instructions propres à l'invocation de chaque méthode. Il pourra alors compilé la classe complétée en faisant appel au compilateur Java de Sun Microsystems, situé dans le package *com.sun.tools.javac*.

Pour clarifier un peu les choses, la figure 12.1 donne le pseudo code d'une telle classe pour l'importation d'une méthode calculant la factorielle d'une entier[4].

[1]C'est-à-dire un ensemble de règles et de faits Prolog ayant le même nom et la même arité.
[2]C'est-à-dire qu'il ne vérifie pas s'ils sont déjà compilé.
[3]C'est-à-dire le nombre de valeur qui devront être retournées par la méthode Java.
[4]La bibliothèque correspondante a été décrite au Chapitre 8.

```
// importation de la bibliothèque contenant
// la méthode à invoquer depuis Prolog
import lib.Math;

public class fact_2
{
  /* constructeur omis */

/* méthode call */
public boolean call(Object[] args)
{
  // arguments à passer à la méthode
PrologTerm[] term;

// on convertit les termes et on obtient
// le nombre de valeurs de retour attendues
  int unboundVars = convert(args, term);

  /* code spécifique à l'importation */

  // instanciation de la bibliothèque
  Math lib = new Math();
  // appel de la méthode importée
  boolean success = lib.fact2(term)

  /* fin du code spécifique */

  // récupération du résultat
  bound(args, term, unboundVars);

  // le prédicat a réussi
  return success;
}
}
```

FIG. 12.1 – Pseudo code d'un prédicat écrit Java pour le compilateur de Joshua Engel

Quatrième partie

Exemples

Chapitre 13

Scénarios d'utilisation du mécanisme d'interopérabilité entre Java et Prolog

Ce chapitre décrit quelques cas d'utilisation typique du mécanisme d'interopérabilité. A partir de là, je tenterai d'indiquer les situations où l'application du mécanisme de Java vers Prolog ou de celui de Prolog vers Java sera la plus adaptée.

13.1 Ajout d'une interface graphique à un système expert

Le premier exemple que nous allons étudier concerne l'ajout d'une interface graphique à un système expert implémenté en Prolog. Pour résoudre ce problème, deux approches sont envisageables :

1. ajouter une boîte à outils graphiques au langage Prolog, et implémenter l'interface utilisateur en utilisant les prédicats de cette boîte à outils;
2. créer l'interface graphique dans un programme Java qui invoquera le système expert.

Pour la première approche, il faudra utiliser le mécanisme d'interopérabilité de Prolog vers Java. Mais, on peut se demander si cette approche est la plus indiquée pour le problème qui nous intéresse ici. En effet, il ne s'agit pas de permettre à un programme Prolog de créer une interface graphique, mais de créer un "front-end" graphique pour un programme Prolog, ce qui est assez différent. Il n'est pas nécessairement utile d'implémenter une boîte à outils graphiques dans une telle situation et cela demanderait un gros travail de développement.

La seconde approche consiste à utiliser les boîtes à outils graphiques Java que sont AWT, Swing ou SWT pour créer une interface graphique, et, ensuite, à invoquer le système expert à travers le mécanisme d'interopérabilité de Java vers Prolog à partir de cette interface graphique. Cette seconde approche est beaucoup plus simple et beaucoup plus indiquée dans l'exemple qui nous intéresse ici. Voyons, en quelques mots, comment va se dérouler un tel scénario d'interopérabilité.

Tout d'abord, dressons la liste des acteurs du scénario. Nous avons :

- une interface graphique écrite en Java;
- la fabrique d'objets Prolog, *PrologFactory*;
- l'interpréteur abstrait du mécanisme d'interopérabilité, *PrologEngine*;

- un interpréteur Prolog concret exécutant le système expert.

Le scénario va se dérouler en deux étapes :

La première étape est l'étape d'initialisation. Dans cette étape, l'interface graphique doit :

1. créer une instance de la fabrique d'objets;
2. demander à la fabrique d'objets une nouvelle instance de l'interpréteur abstrait, ce qui provoquera la création d'une nouvelle instance de l'interpréteur concret;
3. demander une nouvelle instance de théorie à la fabrique d'objets;
4. passer la théorie à l'interpréteur abstrait qui chargera celle-ci dans l'interpréteur concret.

Après cette étape, l'interpréteur abstrait est prêt à répondre aux requêtes de l'interface graphique.

La figure 13.1 illustre cette première étape.

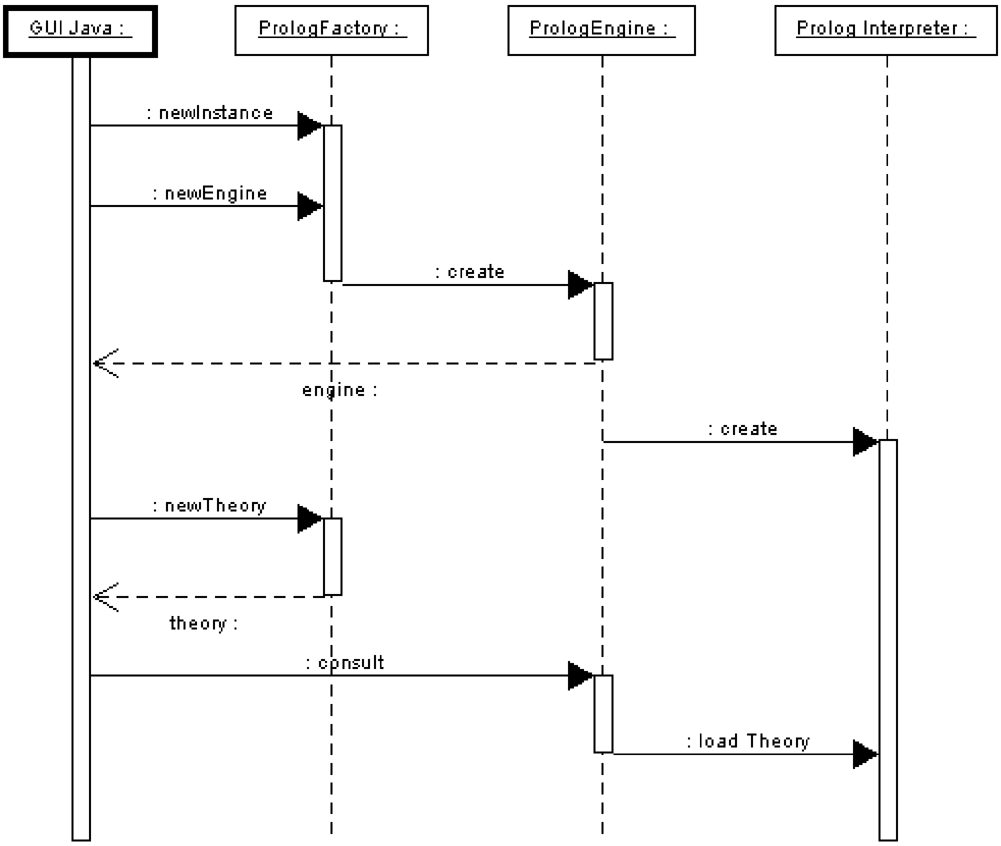

FIG. 13.1 – Étape d'initialisation

Dans la seconde étape, le programme de l'interface graphique va pouvoir interroger le système expert à travers le mécanisme d'interopérabilité et représenter le résultat pour l'utilisateur. C'est l'étape d'interrogation. Pour cela, le programme Java va demander l'évaluation d'une requête au mécanisme d'interopérabilité, qui va transmettre cette demande à l'interpréteur Prolog. Celui-ci va évaluer la requête et retourner la, ou les solution(s) selon le cas, au mécanisme d'interopérabilité. Ce dernier va alors transformer les résultats et les passer à l'interface graphique où ils pourront être exploités.

La figure 13.2 illustre cette seconde étape.

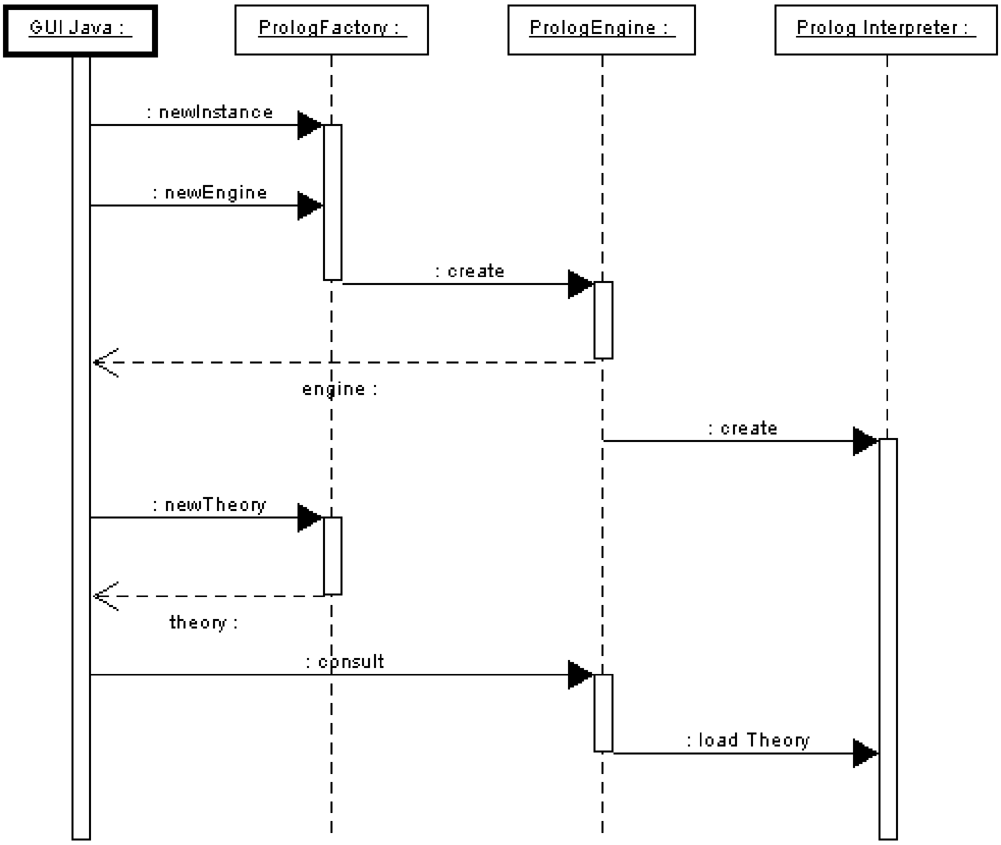

FIG. 13.2 – Étape d'interrogation

13.2 Ajout de prédicats d'interrogation de bases de données relationnelles à un programme Prolog

Ce second cas d'utilisation typique consiste à ajouter des fonctionnalités au langage Prolog, sous forme de prédicats prédéfinis. Supposons, par exemple, que l'on veuille ajouter des prédicats d'interrogation de bases de données relationnelles à Prolog.

Ici, l'objectif est d'ajouter des fonctionnalités au langage Prolog. Le mécanisme d'interopérabilité de Prolog vers Java s'impose donc de lui-même. Les prédicats pourront utiliser l'API JDBC pour accéder à la base de données.

Les acteurs de ce second scénario vont être :

- un programme Prolog;
- le mécanisme d'interopérabilité entre Prolog et Java;
- un prédicat d'interrogation de base de données relationnelles écrit en Java;
- une base de données.

Le scénario se déroule de la manière suivante :

Le programme Prolog fait appel à un prédicat prédéfini à travers le mécanisme d'interopérabilité de Prolog vers Java. Le mécanisme convertit les données qui lui sont passées pour qu'elles puissent être utilisées par le prédicat écrit en Java et les passe à ce dernier. Le prédicat va alors interroger la base de données, calculer un résultat à partir des données reçues en retour, et passer le résultat de ce calcul au mécanisme d'interopérabilité qui va le convertir, avant de le retourner au programme Prolog.

La figure 13.3 illustre ce scénario.

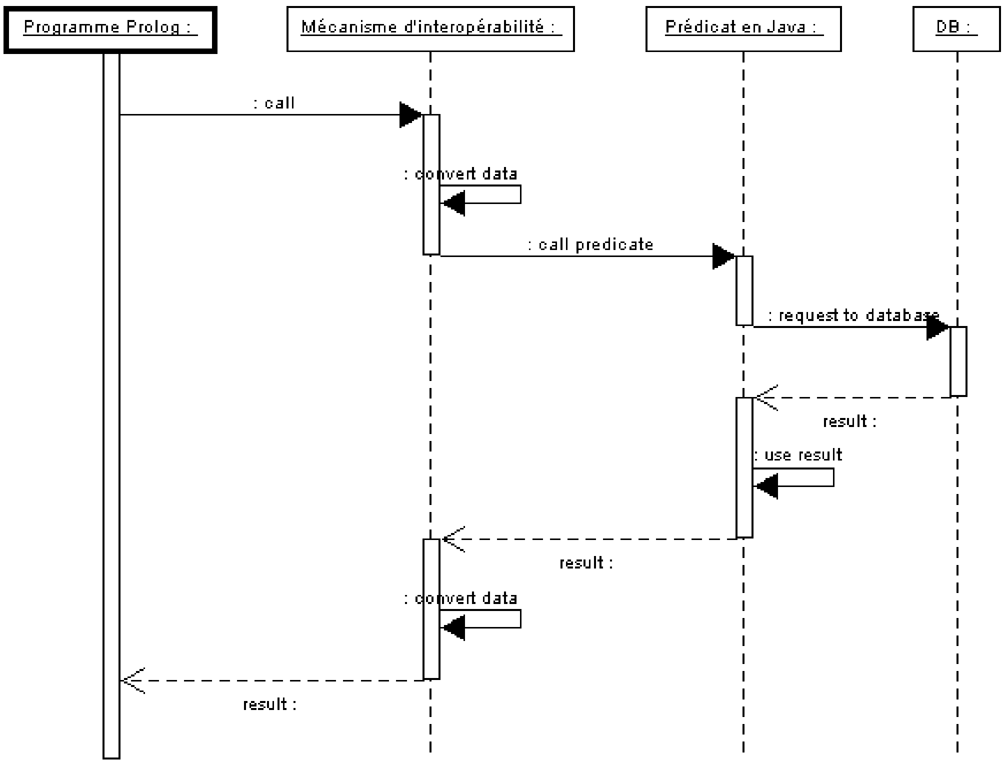

FIG. 13.3 – Scénario d'interopérabilité de Prolog vers Java

13.3 Choix du sens du mécanisme d'interopérabilité

Après avoir étudié les deux exemples typiques précédents, résumons les conditions dans lesquelles l'utilisation de l'un ou l'autre sens du mécanisme d'interopérabilité entre Java et Prolog[1] est plus indiqué.

L'utilisation du mécanisme d'interopérabilité de Java vers Prolog sera le plus indiqué lorsque l'on veut ajouter une couche de présentation (par exemple une interface graphique) par-dessus un programme Prolog déjà existant, ou lorsque l'on désire ajouter de l'intelligence artificielle dans un programme Java.

L'utilisation du mécanisme d'interopérabilité de Prolog vers Java est plutôt réservé aux cas où l'on désire ajouter des fonctionnalités au langage Prolog sous forme de prédicats prédéfinis.

[1] C'est-à-dire le sens de Java vers Prolog ou le sens de Prolog vers Java.

Chapitre 14

Un interpréteur Prolog

Dans ce chapitre, je vais décrire une application complète utilisant le mécanisme d'interopérabilité de Java vers Prolog, implémenté à l'aide du compilateur Prolog vers JCode de Joshua Engel.

14.1 Description du problème

Il s'agit ici d'implémenter un interpréteur Prolog élémentaire capable de charger une théorie et de trouver la solution à des requêtes.

Les fonctionnalités implémentées sont :

- le chargement d'une théorie;
- obtenir toutes les solutions d'une requête en une fois;
- compter les solutions d'une requête;
- obtenir les solutions d'une requête une à une.

14.2 Implémentation

Je ne vais décrire que les fragments de code illustrant la manière dont le mécanisme d'interopérabilité est utilisé. Le code source complet du programme est fourni en annexe.

14.2.1 Instanciation de la fabrique d'objets Prolog et de l'interpréteur

La première étape à suivre est d'instancier la fabrique d'objets Prolog *PrologFactory* et l'interpréteur abstrait *PrologEngine*. La manière dont cela est réalisé est illustrée par la figure 14.1.

14.2.2 Méthode de chargement d'une théorie

Une méthode est chargée dans l'interpréteur abstrait à travers la méthode *consult* de notre exemple. La figure 14.2 donne le code de cette méthode.

La première étape consiste à créer une instance de théorie *PrologTheory* en utilisant la fabrique d'objets Prolog *PrologFactory*. Cette théorie est ensuite passée à la méthode *consult* de l'interpréteur abstrait *engine*.

```
private static String provider =
    "be.ac.ucl.info.blc.interop.compiler.PrologCompilerProvider";

/*...*/

try
{
  // instanciation de la fabrique d'objets Prolog
  factory = PrologFactory.newInstance(provider);
}
catch (UnknownProviderException e1)
{
System.err.println(e1.getMessage());
System.exit(-1);
}
// instanciationde l'interpréteur
engine = factory.newPrologEngine();
```

FIG. 14.1 – Instanciation de la fabrique d'objets et de l'interpréteur

```
private static void consult(String theory) throws InvalidTheoryException
{
  System.out.println("loading " + theory);

  // création d'une instance de théorie
  PrologTheory t = factory.newTheoryFromFile(theory);

  // chargement de la théorie
  engine.consult(t);

  // chargement de la théorie réussi
  System.out.println("Yes");
}
```

FIG. 14.2 – Chargement d'une théorie

14.2.3 Méthode pour trouver toutes les solutions à une requête

Notre interpréteur élémentaire peut trouver et afficher toutes les solutions d'une requête grâce à la méthode *findAll* dont le code est donné à la figure 14.3.

Pour cela, on commence par créer une instance de requête *PrologQuery* en utilisant la méthode *newQuery* de la fabrique d'objets Prolog. Cette requête est ensuite passée à la méthode *findAll()* de l'interpréteur Prolog abstrait. Les solutions sont enfin affichées par un appel à la méthode *printResultSet* décrite à la figure 7.3, donnée en exemple au Chapitre 7 et décrite en détails plus loin.

```
private static void findAll(String query) throws InvalidQueryException,
  NoMoreResultException
{
  // créer une nouvelle requête
  PrologQuery q = factory.newQuery(query);

  // trouver toutes les solutions de la requête
  PrologResultSet rs = engine.findAll(q);

  // afficher les solutions
  if (rs.hasSolution())
 System.out.println(rs.size() + " answers :");
  printResultSet(rs);
}
```

FIG. 14.3 – Obtenir toutes les solutions à une requête

14.2.4 Méthode comptant le nombre de solutions à une requête

La méthode comptant le nombre de solutions à une requête est très semblable à la méthode *findAll* décrite ci-dessus. Son code est donné à la figure 14.4.

```
private static void count(String query) throws InvalidQueryException
{
  // créer une nouvelle instance de requête
  PrologQuery q = factory.newQuery(query);

  // obtenir toutes les solutions de la requête
  PrologResultSet rs = engine.findAll(q);

  // afficher le nombre de solutions trouvées
  System.out.println("Yes");
  System.out.println("count=" + rs.size());
}
```

FIG. 14.4 – Méthode comptant le nombre de solutions à une requête

14.2.5 Méthode pour obtenir les solutions d'une requête une par une

La méthode suivante permet d'obtenir les solutions d'une requête une à une, à la demande. Pour cela, elle utilise les méthodes *find(Query)* et *findNext()* de l'interpréteur abstrait *PrologEngine*.

La première étape consiste à créer une instance de requête à l'aide de la fabrique d'objets Prolog. Cette requête est ensuite passée à la méthode *find* de *engine*. Le résultat retourné est affiché par la méthode *printResult*, décrite au Chapitre 7, figure 7.2, et qui sera reprise en détails plus loin.

La méthode demande alors à l'utilisateur si une autre solution est désirée, auquel cas un appel est fait à la méthode *findNext* de *engine* et le résultat affiché par la méthode *printResult*. Si l'utilisateur ne désire pas d'autre solution, la méthode se termine.

```
private static void find(String query) throws InvalidQueryException,
  QueryNotInitializedException, IOException
{
  // créer une nouvelle requête
  PrologQuery current = factory.newQuery(query);

  // trouver la première solution
  PrologResult r = engine.find(current);

  /* code omis : affichage d'un message d'aide */

  // affichage de la première solution
  printResult(r);

for(;;)
{
    /* code omis :  on demande à l'utilisateur
     s'il désir une autre solution */

  // si l'utilisateur veut une autre solution
  if (nxt.toString().equals(NEXT))
  {
      // trouver la solution suivante
      PrologResult r2 = engine.findNext();

      // afficher la solution suivante
      printResult(r2);

      // s'il n'y a plus de solution on sort de la boucle
      if (!more) { break; }
    }

    // si l'utilisateur ne veut pas d'autre solution
    // on sort de la boucle for(;;)
    else
    {
      more = false;
      break;
    } //else
  } //for
}
```

FIG. 14.5 – Méthode donnant les solutions à une requête une par une à la demande

14.2.6 Méthodes annexes

Plusieurs méthodes annexes sont également déclarées par notre interpréteur Prolog élémentaire. Il s'agit d'une méthode affichant un écran d'aide, d'une méthode affichant des informations sur l'interpréteur, et de deux méthodes pour afficher les solutions des requêtes. Je ne vais pas détailler les deux premières, mais je vais m'intéresser aux deux autres.

La méthode *printResult()* (voir figure 14.6) permet d'afficher le contenu d'un résultat de requête. Pour cela, il lui faut découvrir si le résultat contient une solution, et si cette solution a donné lieu à une substitution. Cela peut être fait en utilisant, respectivement, la méthode *isEmpty()* et la méthode *hasSolution()* de la classe *PrologResult*.

Si le résultat est vide et que la requête a échoué (si *isEmpty()* et *hasSolution()* retournent toutes les deux 'false'), la méthode affiche 'No', avant de se terminer.
Si le résultat est vide mais que la requête a réussi, c'est-à-dire que la résolution de la requête a réussi mais n'a pas engendré de substitution, la méthode affiche 'Yes' et se termine.
Si une substitution a, de plus, été créée, la méthode affiche cette substitution. Pour cela, elle extrait d'abord la substitution à l'aide de la méthode *getSubstitution()* de *PrologResult*. Ensuite, elle extrait chaque variable logique de la substitution et invoque sur chacune la méthode *getStringRepresentation* pour afficher son contenu.

```
public static void printResult(PrologResult r)
{
  // si le résultat est vide
  if (r.isEmpty())
  {
    String hasresult;
    // si le résultat est une solution
    // afficher Yes, sinon, afficher No
    hasresult = r.hasSolution() ? "Yes" : "No";
    System.out.print(hasresult);
  }
  else
  {
    PrologVariable[] vars;
    // on extrait la substitution
    vars = r.getSubstitution();
    int n = vars.length;
    // une solution a été trouvée
    System.out.println("Yes");
    // on affiche la substitution
    for (int i = 0; i < n; ++i)
    {
      System.out.print(
        vars[i].getStringRepresentation());
      if (!(i == n - 1))
      {
        System.out.print(", ");
      }
    }
  }
}
```

FIG. 14.6 – Méthode affichant une solution

La méthode *printResultSet()* est utilisée pour afficher toutes les solutions d'une requête. Si la requête a échoué, la méthode affiche 'No' et se termine. Si la requête a réussi mais n'a donné lieu à aucune substitution, la méthode affiche 'Yes' et se termine. Si la requête a réussi et a engendré une ou plusieurs susbtitutions, la méthode parcourt le *PrologResultSet* passé en argument à l'aide des méthodes *hasNext()* et *next()* pour obtenir les solutions et fait appel à *printResult()* pour les afficher.

```
public static void printResultSet(PrologResultSet r)
  throws NoMoreResultException
{
  if (r.isEmpty())
  {
    String hasresult;
    // si le PrologResultSet est une solution
    // afficher Yes, sinon, afficher No
    hasresult = r.hasSolution() ? "Yes" : "No";
    System.out.println(hasresult);
  }
  else
  {
    // tant qu'il y a un résultat
    // dans le PrologResultSet
    while (r.hasNext())
    {
      // on extrait le résultat
      PrologResult re = r.next();
      // on affiche le résultat
      printResult(re);
      System.out.println();
    }
  }
}
```

FIG. 14.7 – Méthode affichant toutes les solutions à une requête

14.2.7 La boucle principale

La boucle principale de l'interpréteur lit les commandes entrées par l'utilisateur et invoque les éventuelles méthodes correspondantes.

14.3 Utilisation de l'interpréteur

Pour lancer l'interpréteur, il suffit d'exécuter la commande `java PrologInterpreter` dans une console. L'interpréteur se lancera et proposera une ligne de commandes où les requêtes pourront être entrées.

La ligne de commandes de l'interpréteur commence par le symbole "?-", après lequel doivent être entrées les requêtes. Les requêtes doivent obligatoirement se terminer par un ".", sous peine de provoquer une erreur.

Outre les requêtes se référant à un programme Prolog chargé, l'interpréteur inclut un certain nombre de requêtes spéciales :

- *consult*, permet de charger une théorie. Par exemple, *consult(within.p).*;
- *findall*, affiche toutes les solutions d'une requête. Par exemple, *findall(within(X, usa)).*.
- *count*, affiche le nombre de solutions d'une requête. Par exemple, *count(within(X, californie)).*;

- *help*, permet d'obtenir un écran d'aide. Utilisation : *help.*;
- *about*, affiche des informations sur l'interpréteur. Utilisation : *about.*;
- *bye*, permet de quitter l'interpréteur. Utilisation *bye.*.

La figure 14.8 donne une capture d'écran de l'interpréteur.

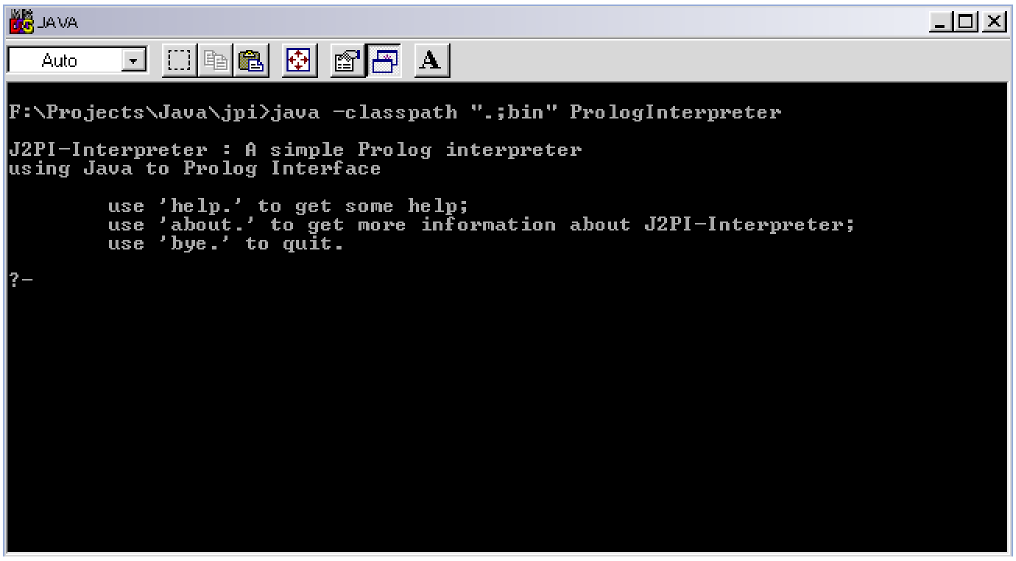

FIG. 14.8 – L'interface utilisateur de l'interpréteur Prolog

Cinquième partie

Conclusion et perspectives

Chapitre 15

Conclusion

Me voici arriver au terme de ce mémoire. Il est maintenant temps de conclure et de passer en revue ce que nous avons vu au cours de cette petite dicussion.

L'objectif de ce mémoire était d'étudier l'interopérabilité de deux langages de programmation basée sur la machine virtuelle de Java, dans le cas particulier de l'interaction entre les langages Java et Prolog.

Pour cela, j'ai tout d'abord décrit les problèmes de gestion de la mémoire et de conversion de données, impliqués par la réalisation d'un mécanisme d'interopérabilité entre deux langages aussi différents que Java et Prolog. J'ai ensuite spécifié et, en partie implémenté, un mécanisme d'interopérabilité transparent et générique entre Java et Prolog, basé sur la machine virtuelle de Java. J'ai également cherché à tirer le meilleur parti possible de l'utilisation de la machine virtuelle de Java en basant mon implémentation sur le compilateur Prolog vers JCode, développé par Joshua Engel dans le cadre de son livre [Eng99].

Le mécanisme décrit dans ce mémoire devrait assez facilement être adapté à l'interopérabilité entre Java et d'autres langages interprétés, tels que Lisp, Scheme[1]...

Il était impossible, dans le temps imparti, d'étudier en profondeur un sujet aussi vaste que celui abordé par ce mémoire, et, surtout, à le condenser en un simple mémoire. J'aurais bien aimé pouvoir pousser plus loin mon étude et mettre sur papier les nombreuses idées que je n'ai pu aborder, faute de temps et de place. J'aurais, en particulier, voulu passer plus de temps sur l'implémentation du mécanisme d'interopérabilité et ainsi fournir une implémentation focntionnelle de la partie permettant à un programme Prolog d'invoquer des méthodes Java. J'aurais également aimé réaliser une implémentation avec un second interpréteur Prolog écrit en Java[2]. J'aurai aussi voulu corriger les quelques bugs du compilateur de Joshua Engel et compléter ses fonctionnalités. Une dernière chose qui m'aurait plu, aurait été de discuter plus en profondeur des différences entre paradigme de la programmation orientée-objets et paradigme de la programmation logique. Je n'ai pas eu non plus le temps d'implémenter un exemple plus conséquent pour illustrer l'utilisation de mon mécanisme d'interopérabilité.

Malgré tous ces aspects que je n'ai pu étudier, ce mémoire m'a appris énormément de choses sur les langages Prolog et Java, et en particulier sur la manière dont ils sont implémentés. J'ai découvert que ces deux langages sont beaucoup plus subtils qu'il n'y paraît au premier abord et j'aurais désiré pousser plus moins mon étude à leur sujet. J'ai également appris qu'il était difficile d'abandonner certains aspects intéressants d'un problème, mais que cet abandon était souvent nécessaire à l'aboutissement d'une étude ou d'un projet.

Ce mémoire, malgré les difficultés inévitables que j'ai rencontrées lors de sa réalisation, a été pour moi une expérience enrichissante et que j'aurais aimé pousser plus loin.

[1] Une implémentation pourrait d'ailleurs être réalisée à l'aide du compilateur Scheme vers JCode, développé, de nouveau, par Joshua Engel dans le livre précité.

[2] Implémentation que j'avais commencée, mais que j'ai préféré abandonner pour terminer ce mémoire dans les temps.

Chapitre 16

Perspectives

Pour clore ce mémoire, j'aimerais aborder quelques idées qui pourraient être exploitées dans l'avenir dans le cadre de l'interopérabilité entre Prolog et Java.

Une manière d'améliorer le mécanisme d'interopérabilité de Java vers Prolog serait de construire un mécanisme dans lequel un programme Java pourrait directement appler un programme Prolog, un peu de la manière dont les méthodes natives sont appelées depuis Java. Ainsi, au lieu de devoir instancier un interpréteur Prolog et lui demander la résolution de requêtes, cette résolution pourra être cachée dans l'invocation d'une méthode Java. Une manière de réaliser cela pourrait être d'étendre le langage Java avec un nouveau mot clé, *prolog*, indiquant que la méthode invoquée est en réalité la résolution d'une requête Prolog. Par exemple, la méthode *public prolog PrologList cons(PrologTerm X, PrologList L1);* pourrait directement être traduite en la résolution de la requête *cons(X, Y,* $[X|Y]$*)*.

En ce qui concerne le mécanisme d'interopérabilité de Prolog vers Java, une amélioration serait de simplifier les signatures des méthodes Java invoquables. Ainsi, une méthode permettant d'effectuer le calcul de la factorielle d'un nombre entier positif pourrait avoir comme signature *public int fact(int n)*, ce qui est plus instinctif que l'actuel *public boolean fact2(PrologTerm[] args)*. Dans ce cas, le mécanisme d'interopérabilité devra se charger lui-même de la vérification des types de données, de leur conversion, de la gestion des exceptions...

Java propose un mécanisme de gestion des exceptions simple et performant. Certaines implémentations de Prolog fournissent également de tels mécanismes. Il pourrait donc être possible de passer à un programme Java des exceptions générées par un programme Prolog et inversement. Un tel mécanisme pourrait être créé en ajoutant à Prolog des prédicats prédéfinis permettant de passer des exceptions générées dans un programme Prolog à un programme Java, et, inversement, de récupérer depuis Prolog des exceptions générées dans un programme Java. Ce mécanisme pourrait, par exemple, faire appel à une pile pour stocker les exceptions, et à des envois de notification au programme auquel elles sont destinées.

L'étape suivante dans l'amélioration du mécanisme d'interopérabilité sera de concevoir un langage hybride entre Java et Prolog, qui contiendra du code des deux langages. Mais ceci est une autre histoire...

Bibliographie

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine : a tutorial reconstruction*. The MIT Press, 1991.

[Apt01] Krzysztof R. Apt. The logic programming paradigm and prolog. 2001. Disponible en ligne à l'adresse http://arxiv.org/abs/cs.PL/0107013.

[BBE98] Athman Bouguettaya, Boualem Benatallah, and Ahmed Elmagarmid, editors. *Interconnecting Heterogeneous Information Systems*. Kluwer Academic Publisher, 1998.

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[Dev90] Yves Deville. *Logic Programming : Systematic Program Development*. Addison-Wesley, 1990.

[DL99] Phillipe Delsarte and Axel Van Lamsweerd. Ingi 2341 mathématiques discrètes : Bases logiques de l'informatique, 1999. Notes de cours du cours Ingi 2341 - Université Catholique de Louvain - Faculté des Sciences Appliquées - Département d'Ingénierie Informatique.

[ea97] Frank Buschmann et al. *Pattern-oriented Software Architecture*. John Wiley and Sons, 1997.

[EC00] Robert Eckstein and Michel Casabianca. *XML précis et concis*. O'Reilly, 2000.

[Eng99] Joshua Engel. *Programming for the Java Virtual Machine*. Addison-Wesley, 1999.

[Gor98] Rob Gordon. *JNI : the Java Native Interface*. Prentice Hall, 1998.

[Hod90] Joshua S. Hodas. Compiling prolog, from the plm to the wam and beyond. 1990. Disponible en ligne à l'adresse http://www.cs.hmc.edu/ hodas/papers/wpe.pdf.

[Hov95] David Hovel. Using prolog in windows NT network configuration. In *Proceedings of the Practical Applications in Prolog Conference*. Decision Theory and Adaptive Systems Group - Microsoft Research, April 1995. disponible en ligne à l'adresse http://research.microsoft.com/dtas/davidhov/pap.htm.

[Kra] Andreas Krall. Implementation techniques for prolog. Disponible en ligne à l'adresse http://www.complang.tuwien.ac.at/andi/wlp_94.ps.gz.

[Lam95] Axel Van Lamsweerde. Ingi 2635 concepts et méthodes de l'intelligence artificielle, 1995. notes de cours du cours Ingi 2635 - Université Catholique de Louvain - Faculté des Science Appliquées - Département d'Ingénirie Informatique.

[LY99] Tim Lindholm and Franck Yellin. *The Java Virtual Michine Specification (2nd Edition)*. Addison-Wesley, 1999.

[McL02] Brett McLaughlin. *Java et XML*. O'Reilly, 2002.

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards, editors. *Client/Server - Guide de Survie*. John Wiley and Son, 1997.

[Roy91] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of Southern California, 1991. Disponible en ligne à l'adresse http://www.info.ucl.ac.be/people/PVR/Peter.thesis/Peter.thesis.html.

[Roy93] Peter Van Roy. The wonder years of sequential prolog implementation. Technical report, Digital Research Laboratory - Paris, 1993.

[SS97] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1997.

[TD01] André Thayse and Philippe Delsarte. *Logique pour le traitement de la langue naturelle*. Hermes Science, 2001.

[Tha91] André Thayse, editor. *From natural language processing to logic for expert systems*. John Wiley and Son, 1991.

[Ven98] Bill Venners. *Inside the Java Virtual Machine*. MacGraw Hill, 1998.

Sixième partie

Annexes

Annexe A

Spécification complète des méthodes du mécanisme d'interopérabilité

Dans cette annexe se trouve la spécification complète des méthodes du mécanisme d'interopérabilité. Leur présentation suit l'ordre dans lequel ces méthodes ont été présentées dans la partie II de ce mémoire.

A.1 Représentation des données et méthodes de conversion

A.1.1 Méthodes de la classe PrologTerm

- public java.lang.String getStringRepresentation(); :
Retourne une représentation de la classe sous forme de chaîne de caractères Java.
In : void;
Out : java.lang.String - représentation du terme sous la forme d'une chaîne de caractères;
Pre : -;
Post : la représentation sous forme de chaîne de caractères du terme a été retournée;

A.1.2 Méthodes de la classe PrologConstant

- public java.lang.Object getValue(); :
Retourne la valeur stockée dans la constante Prolog sous forme d'un objet java.lang.Object. Si la constante est un atome, cette objet sera une chaîne de caractère java.lang.String; si c'est une constante numérique, la méthode retournera un objet java.lang.Number; si c'est une constante entière ou de type nombre à virgule flottante, la méthode retournera, respectivement, un entier java.lang.Integer ou un nombre à virgule flottant java.lang.Float.
In : void;
Out : java.lang.Object - la valeur de la constante;
Pre : la constante a été initialisée (i.e. elle contient une valeur);
Post : la valeur de la constante a été retournée;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java. La représentation sous forme de chaîne de caractères d'une constante Prolog est la représentation sous forme de chaîne de caractère de la valeur de cette constante.
(Voir PrologTerm).

A.1.3 Méthodes de la classes PrologAtom

- public java.lang.Object getValue(); héritée de PrologConstant :
Retourne la valeur stockée dans l'atome Prolog sous forme d'un Object Java.
(Voir PrologConstant).

- public java.lang.String getString(); :
Retourne le contenu de l'atome sous forme d'une chaîne de caractères Java.
In : void;
Out : java.lang.String - la valeur de l'atome;
Pre : l'atome a été initialisé (i.e. contient une valeur);
Post : la valeur de l'atome a été retournée;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java.
(Voir PrologTerm).

A.1.4 Méthodes de la classe PrologNumber

- public java.lang.Object getValue(); héritée de PrologConstant :
Retourne la valeur stockée dans le nombre Prolog sous forme d'un Object Java.
(Voir PrologConstant).

- public java.lang.Number getNumber(); :
Retourne le contenu de la constante numérique Prolog sous forme d'un nombre (java.lang.Number) Java. Ce nombre sera un entier java.lang.Integer si le *prologNumber* est une constante entière, ou un nombre à virgule flottante java.lang.Float s'il s'agit d'une constante numérique de type virgule flottante.
In : void;
Out : java.lang.Number - la valeur de la constante numérique;
Pre : la constante numérique a été initialisée;
Post : la valeur de la constante numérique a été retournée;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java.
(Voir PrologTerm).

A.1.5 Méthodes de la classe PrologInteger

- public java.lang.Object getValue(); héritée de PrologConstant :
Retourne la valeur stockée dans l'entier Prolog sous forme d'un Object Java.
(Voir PrologConstant).

- public java.lang.Number getNumber(); héritée de PrologNumber :
Retourne le contenu de l'entier Prolog sous forme d'un nombre (java.lang.Number) Java.
(Voir PrologNumber).

- public java.lang.Integer getInteger(); :
Retourne le contenu de l'entier Prolog sous forme d'un Integer (java.lang.Integer) Java.
In : void;
Out : java.lang.Integer - la valeur de l'entier 32bits;
Pre : l'entier a été initialisé;
Post : la valeur de l'entier a été retournée;

- public int getPrimitiveInt(); :
Retourne le contenu de l'entier Prolog sous forme d'un entier primitif 32 bits (int) Java.
In : void;
Out : int - la valeur de l'entier 32bits;
Pre : l'entier a été initialisé;
Post : la valeur de l'entier a été retournée;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java.
(Voir PrologTerm).

A.1.6 Méthodes déclarées par la classe PrologFloat

- public java.lang.Object getValue(); héritée de PrologConstant :
Retourne la valeur stockée dans le nombre à virgule flottante Prolog sous forme d'un Object Java.
(Voir PrologConstant).

- public java.lang.Number getNumber(); héritée de PrologNumber :
Retourne le contenu du nombre à virgule flottante Prolog sous forme d'un nombre (java.lang.Number) Java.
(Voir PrologNumber).

- public java.lang.Float getFloat(); :
Retourne le contenu du nombre à virgule flottante Prolog sous forme d'un Float (java.lang.Float) Java.
In : void;
Out : java.lang.Float - la valeur du nombre à virgule flottante 32bits;
Pre : la constante (nombre à virgule flottante) a été initialisée;
Post : la valeur du nombre à virgule flottante a été retournée;

- public float getPrimitiveFloat(); :
Retourne le contenu du nombre à virgule flottante Prolog sous forme d'un nombre virgule flottante primitif 32 bits (float) Java.
In : void;
Out : float - la valeur du nombre à virgule flottante 32bits;
Pre : la constante (nombre à virgule flottante) a été initialisée;
Post : la valeur du nombre à virgule flottante a été retournée;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java.
(Voir PrologTerm).

A.1.7 Méthodes de la classe PrologVariable

- public java.lang.String getName(); :
Retourne le nom de la variable logique **In :** void;
Out : java.lang.String - le nom de la variable;
Pre : la variable a été initialisée;
Post : le nom de la variable a été retourné;

- public boolean isBound(); :
Retourne *true* si la variable est liée à une valeur [1], *false* dans le cas contraire.
In : void;
Out : boolean - 'true' si la variable et liée, 'false' sinon;
Pre : -;
Post : 'true' a été retourné si la variable est liée, sinon 'false' a été retourné;

- public void setBinding(PrologTerm) throws BindException; :
Lie la variable au terme Prolog passé en argument. Un exception de type BindException est générée si le variable est déjà liée.
In : PrologTerm - la valeur à lier à la variable;
Out : void;
Pre : la variable est non liée;
Post : la variable est liée au terme passé en argument;
Exception : BindException - levée si la variable était déjà liée;

- public PrologTerm getBinding(); :
Retourne la valeur liée à la variable [2] ou *null* si la variable est non liée.
In : void;
Out : PrologTerm - le terme lié à la variable, ou 'null' si la variable est non liée;
Pre : -;
Post : le terme lié à la variable a été retourné, ou 'null' si la variable n'était pas liée;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java (Voir PrologTerm). Le format de représentation d'une variable logique sous forme de chaîne de caractères est $Nom = Valeur$, par exemple $X = jean$.

A.1.8 Méthodes de la classe PrologStructure

- public java.lang.String getName(); :
Retourne le nom de la structure.
In : void;
Out : java.lang.String - le nom du terme composé;
Pre : le terme composé a été initialisé;
Post : le nom du terme composé a été retourné;

- public int getArity(); :
Retourne l'arité de la structure, c'est-à-dire le nombre de ses arguments.
In : void;
Out : int - l'arité du terme composé (i.e. son nombre d'arguments);
Pre : le terme composé a été initialisé;
Post : l'arité du terme composé a été retournée;

[1] Un déréférencement peut être nécessaire selon l'implémentation...
[2] Un déréférencement peut être nécessaire selon l'implémentation...

- public PrologTerm[] getArguments(); :
Retourne un tableau de PrologTerm contenant les arguments de la structure.
In : void;
Out : PrologTerm[] - tableau des arguments du terme composé, un tableau de taille 0 si la structure ne contient aucun argument;
Pre : le terme composé a été initialisé;
Post : les arguments du terme composé ont été retournés ou un tableau de taille 0 si la structure ne contient aucun argument;

- public java.lang.String getFunctor(); :
Retourne le foncteur représentant la structure sous la forme *nom'/'arité*, par exemple *"father/2"*.
In : void;
Out : java.lang.String - le foncteur du terme composé;
Pre : le terme composé a été initialisé;
Post : le foncteur du terme composé a été retourné;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java. Le format de représentation d'une structure Prolog sous forme de chaîne de caractère est *Nom(ListeArguments)*, par exemple *frere(paul,jean)*.
(Voir PrologTerm).

A.1.9 Méthodes de la classe PrologList

- public java.lang.String getName(); héritée de PrologStructure :
Retourne le nom de la structure, c'est-à-dire *"."*.
In : void;
Out : java.lang.String - le nom du terme composé a été retourné, c'est-à-dire '.';
Pre : la liste a été initialisée;
Post : le nom du foncteur de la liste, '.', a été retourné;

- public int getArity(); héritée de PrologStructure :
Retourne l'arité de la structure, c'est-à-dire le nombre de ses arguments, toujours égal à 2 pour une liste.
In : void;
Out : int - l'arité de la liste, 2;
Pre : la liste a été initialisée;
Post : l'arité de la liste, 2, a été retournée;

- public PrologTerm[] getArguments(); héritée de PrologStructure :
Retourne un tableau de PrologTerm contenant les arguments de la structure, c'est-à-dire un tableau contenant comme premier élément la tête de la liste (de type PrologTerm) et comme second élément la queue de la liste (de type PrologList).
In : void;
Out : PrologTerm[] - un tableau de termes content la tête et la queue de la liste;
Pre : la liste a été initialisée;
Post : les arguments de la liste, sa tête et sa queue, ont été retournés;

- public PrologTerm getHead(); :
Retourne la tête de la liste, 'null' si la liste est vide.
In : void;
Out : PrologTerm - la tête de la liste, 'null' si la liste est vide;
Pre : la liste a été initialisée;
Post : la tête de la liste a été retournée, 'null' si la liste est vide;

- public PrologList getTail(); :
Retourne la queue de la liste, 'null' si la liste est vide (c'est-à-dire '[]').
In : void;
Out : PrologList - la queue de la liste, null si la liste est vide;
Pre : la liste a été initialisée;
Post : la queue de la liste a été retournée, ou null si la liste est vide;

- public PrologTerm[] toArray(); :
Retourne le contenu de la liste sous forme d'un tableau de PrologTerm, ou un tableau de taille 0 si la liste est vide.
In : void;
Out : PrologTerm[] - les termes contenus dans la liste sous forme de tableau, un tableau de taille 0 si la liste est vide;
Pre : la liste a été initialisée;
Post : les termes contenus dans la liste ont été retournés ou un tableau de taille 0 a été retourné si la liste était vide;

- public int size(); :
Retourne le nombre de termes contenu dans la liste, 0 si la liste est vide.
In : void;
Out : int, le nombre de termes contenu dans la liste, 0 si la liste est vide;
Pre : la liste a été initialisée;
Post : la taille de la liste a été retournée;

- public boolean isEmpty(); :
Retourne 'true' si la liste est vide, c'est-à-dire '[]'.
In : void;
Out : boolean - 'true' si la liste est vide, 'false' sinon;
Pre : la liste a été initialisée;
Post : a retourné 'true' si la liste est vide, 'false' sinon;

- public boolean hasNext(); :
Retourne 'true' si la liste contient un terme suivant.
In : void;
Out : boolean - 'true' si la liste contient un terme suivant, 'false' sinon;
Pre : la liste a été initialisée;
Post : a retourné 'true' si la liste contient un terme suivant, 'false' sinon;

- public PrologTerm next() throws NoMoreTermException; :
Retourne le terme suivant dans la liste.
In : void;
Out : PrologTerm - le terme suivant dans la liste;
Pre : la liste contient encore au moins un terme;
Post : le terme suivant dans la liste a été retourné;
Exception : NoMoreTermException - lancée si la liste est vide;

- public void add(PrologTerm); :
ajoute le terme passé en argument en tête de la liste.
In : PrologTerm - le terme à ajouter en tête de la liste;
Out : void;
Pre : -;
Post : le terme passé en argument a été ajouté en tête de la liste;

- public PrologTerm remove() throws NoMoreTermException; :
Retourne et supprime le terme situé en tête de la liste.
In : void;
Out : PrologTerm - le terme en tête de la liste la liste;
Pre : la liste n'est pas vide;
Post : le terme situé en tête de la liste a été supprimé et retourné;
Exception : NoMoreTermException - lancée si la liste est vide;

- public java.lang.String getStringRepresentation(); héritée de PrologTerm :
Retourne une représentation de la classe sous forme de chaîne de caractères Java. Le format de représentation d'une liste sous forme de chaîne de caractère est [], pour une liste vide, ou *[*Arg_1*,...,*Arg_n*]*, par exemple *[papa, maman, fils, fille]*.
(Voir PrologTerm).

A.1.10 Méthodes de la fabrique de termes

- public PrologTerm convertTerm(java.lang.Object); :
Retourne une instance de *PrologTerm* construite à partir d'un terme de l'interpréteur Prolog utilisé pour l'implémentation du mécanisme d'interopérabilité.
In : java.lang.Object - terme de l'interpréteur Prolog sous-jacent;
Out : PrologTerm - terme du mécanisme d'interopérabilité;
Pre : -;
Post : la conversion du terme passé en argument en terme du mécanisme d'interopérabilité a été retournée;

- public PrologConstant newConstant(java.lang.Object) throws InvalidArgumentException; :
Transforme un objet Java en constante Prolog. Si l'objet passé en argument est de type java.lang.String, la constante retournée sera de type *PrologAtom*; s'il s'agit d'un java.lang.Integer, la constante retournée sera un *PrologInteger*; s'il s'agit d'un java.lang.Float, la constante retournée sera de type *PrologFloat*. Si l'objet passé en argument ne correspond à aucun de ces trois cas, une exception de type InvalidArgumentException est générée.
In : java.lang.Object - objet à partir duquel la constante Prolog doit être construite;
Out : PrologConstant - la constante construite à partir de l'objet passé en argument;
Pre : l'objet passé en argument est un type valide pour la construction d'une constante Prolog;
Post : la constante construite à partir de l'objet passé en argument a été retournée;
Exception : InvalidArgumentException - lancée l'objet passé en argument ne peut être utilisé pour construire une constante Prolog;

- public PrologInteger newPrologInteger(int); :
Retourne une constante numérique entière Prolog construite à partir de l'entier primitif 32 bits passé en argument.
In : int - valeur de la constante;
Out : PrologInteger - constante numérique entière construite à partir de l'entier 32 bits primitif passé en argument;
Pre : -;
Post : la constante numérique entière construite à partir de l'entier 32 bits primitif passé en argument a été retournée;

- public PrologFloat newPrologFloat(int); :
Retourne une constante numérique à virgule flottante Prolog construite à partir du nombre à

virgule flottante primitif 32 bits passé en argument.
In : float - valeur de la constante;
Out : PrologFloat - constante numérique à virgule flottante construite à partir du nombre à virgule flottante 32 bits primitif passé en argument;
Pre : -;
Post : la constante numérique à virgule flottante construite à partir du nombre à virgule flottante 32 bits primitif passé en argument a été retournée;

- public PrologStructure newStructure(java.lang.String, PrologTerm[]); :
Retourne une instance de structure Prolog *PrologStructure* à partir de son nom et de ses arguments.
In : java.lang.String - le nom de la structure;
In : PrologTerm[] - les arguments de la structure;
Out : PrologTerm - terme ayant comme nom et arguments ceux passé en paramètres à la méthode;
Pre : -;
Post : le terme ayant comme nom et arguments ceux passé en paramètres à la méthode a été retourné;

- public PrologList newList(); :
Retourne une liste Prolog vide, c'est-à-dire '[]'. Celle-ci pourra être remplie à l'aide de l'opération *add(PrologTerm)* de la classe *PrologList*.
In : void;
Out : PrologList - une liste vide;
Pre : -;
Post : une liste Prolog vide a été retournée;

A.2 Mécanisme d'interopérabilité de Java vers Prolog

A.2.1 Méthodes de la classe représentant les solutions des requêtes Prolog

Méthodes de la classe PrologResult

- public boolean isEmpty(); :
Retourne 'true' si le résultat est vide, c'est-à-dire qu'il ne contient pas de variables ayant été liées lors de la résolution de la requête, 'false' sinon. Cette méthode doit être utilisée avec *hasSolution()* pour savoir si l'évaluation de la requête a ou non généré une solution. En effet, ce n'est pas parce que *isEmpty()* retourne 'false' qu'aucune solution n'a été trouvée. Par exemple, l'évaluation de *frere(paul, jean).* peut avoir une solution, mais le résultat retourné sera vide.
In : void;
Out : boolean - 'true' si le résultat est vide, c'est-à-dire ne contient pas de variables ayant été liées lors de la résolution de la requête, 'false' sinon;
Pre : le résultat a été initialisé par l'exécution d'une requête;
Post : 'true' a été retourné si le résultat est vide, 'false' sinon;

- public boolean hasSolution(); :
Retourne 'true' si une solution a été trouvée lors de l'évaluation de la requête qui a créé le résultat, 'false' sinon. Cette méthode est différente de la méthode *isEmpty()*, qui indique si le résultat contient des variables liées lors de la résolution de la requête, car ce n'est pas parce qu'aucune variable n'a été liée lors de la résolution d'une requête que celle-ci a échoué.

In : void;
Out : boolean - 'true' si une solution a été trouvée à la requête ayant généré le résultat, 'false' sinon;
Pre : le résultat a été initialisé par l'exécution d'une requête;
Post : 'true' a été retourné si une solution a été trouvée à la requête ayant généré le résultat, 'false' sinon;

- public PrologVariable[] getSubstitution(); :
Retourne la substitution vérifiant la requête sous la forme d'un tableau contenant les variables non-liées ayant été liées lors de l'exécution de la requête qui a généré le résultat, un tableau de taille 0 si aucune variable non-liée n'a été liée.
In : void;
Out : PrologVariable[] - un tableau contenant les variables non-liées ayant été liées lors de l'exécution de la requête qui a généré le résultat, un tableau de taille 0 si aucune variable non-liée n'a été liée;
Pre : le résultat a été initialisé par l'exécution d'une requête;
Post : les variables non-liées ayant été liées par la résolution de la requête et contenues dans le résultat ont été retournées ou un tableau de taille 0 si aucune variable n'a été liée lors de l'évaluation de la requête.;

Méthodes de la classe PrologResultSet

- public boolean isEmpty(); :
Retourne 'false' si le result set contient des résultats, c'est-à-dire si la résolution de la requête a généré une ou plusieurs substitutions, 'true' si aucun résultat n'a été généré. Comme pour la classe *PrologResult*, un result set vide ne signifie pas que la requête a échoué. Il faut avoir recours à la méthode *hasSolution()* pour le savoir.
In : void;
Out : boolean - 'true' si le result set ne contient aucun résultat, 'false' sinon;
Pre : le result set a été initialisé par la résolution d'une requête;
Post : 'true' a été retourné si le result set ne contient aucun résultat, 'false' sinon;

- public boolean hasSolution(); :
Retourne 'true' si la requête dont l'évaluation a créé le result set avait une solution. Remarque, si, pour un *PrologResultSet*, *isEmpty()* et *hasSolution()* retournent toutes les deux 'true', cela signifie que la requête avait exactement une solution et que son évaluation n'a lié aucune variable. C'est ce qui se produira lors de la résolution de requête contenant uniquement des variables liées comme, par exemple, *frere(paul, jean).*.
In : void;
Out : boolean - 'true' si une solution a été trouvée à la requête ayant généré le result set, 'false' sinon;
Pre : le result set a été initialisé par l'exécution d'une requête;
Post : 'true' a été retourné si une solution a été trouvée à la requête ayant généré le result set, 'false' sinon;

- public int size(); :
Retourne la taille du result set, c'est-à-dire le nombre de résultats qu'il contient.
In : void;
Out : int - le nombre de résultats contenu dans le result set, 0 si le result set est vide;
Pre : le result set a été initialisé par l'exécution d'une requête;
Post : le nombre de résultats dans le result set a été retourné, 0 si le result set est vide;

- public boolean hasNext(); :
Retourne 'true' si le result set contient un autre résultat, 'false' sinon. Cette méthode permet, en conjonction avec *next()* d'utiliser un *PrologResultSet* comme un itérateur des résultats qu'il contient.
In : void;
Out : boolean - 'true' si le result set contient un autre résultat, 'false' sinon;
Pre : le result set a été initialisé par l'exécution d'une requête;
Post : 'true' a été retourné si le result set contient un autre résultat, 'false' sinon;

- public PrologResult next() throws NoMoreResultException; :
Retourne le résultat suivant dans le result set. Cette méthode permet, en conjonction avec *next()* d'utiliser un *PrologResultSet* comme un itérateur des résultats qu'il contient.
In : void;
Out : PrologResult - le résultat suivant dans le result set;
Pre : le result set a été initialisé par l'exécution d'une requête et contient encore au moins un résultat;
Post : le résultat suivant dans le result set a été retourné;
Exception : NoMoreResultException - lancée si le result set ne contient pas de résultat suivant;

- public PrologResult[] toArray(); :
Retourne un tableau contenant tous les résultats du result set, un tableau de taille 0 si le result set est vide.
In : void;
Out : PrologResult[] - un tableau contenant tous les résultats du result set, un tableau de taille 0 si le result set est vide;
Pre : le result set a été initialisé par l'exécution d'une requête;
Post : tableau contenant tous les résultats du result set, un tableau de taille 0 si le result set est vide, a été retourné;

A.2.2 Méthodes de l'interpréteur abstrait

Méthodes de méta-programmation

- public void consult(PrologTheory) throws InvalidTheoryException; :
Charge la théorie passée en argument dans l'interpréteur Prolog.
In : PrologTheory - la théorie (i.e. le programme Prolog) à charger dans l'interpréteur abstrait;
Out : void;
Pre : la théorie est valide;
Post : le théorie passée en argument a été chargée dans l'interpréteur Prolog;
Exception : InvalidTheoryException - si la théorie n'est pas valide;

- public void assertz(PrologClause) throws InvalidClauseException; [3] :
Ajoute la clause passée en argument à la théorie chargée dans l'interpréteur Prolog. Aucune vérification n'est garantie pour assurer la cohérence de la théorie après l'ajout de la clause. Cette méthode est suggérée mais pas obligatoire, elle n'est pas nécessairement supportée par tous les interpréteurs Prolog.
In : PrologClause - la clause à ajouter à la théorie courante, cette clause doit être un fait *PrologFact* ou une règle *PrologRule*;
. **Out :** void;
Pre : la clause à ajouter n'existe pas dans la théorie courante;
Post : la clause donnée en argument a été ajoutée à la théorie courante;
Exception : InvalidClauseException - lancée si la clause est une requête ou s'il s'agit d'un fait ou d'une règle non valide;

[3] Au départ, cette méthode était baptisée *assert* mais ce terme est un mot clé de Java dans la version 1.4.

- public void retract(PrologClause) throws InvalidClauseException :
Rétracte la clause passée en argument de la théorie chargée dans l'interpréteur. Aucune vérification n'est garantie pour assurer la cohérence de la théorie après révocation de la clause. Cette méthode est suggérée mais pas obligatoire, elle n'est pas nécessairement supportée par tous les interpréteurs Prolog.
In : PrologClause - la clause à rétracter;
Out : void;
Pre : la clause à rétracter se trouve dans la théorie courante;
Post : ;
Exception : InvalidClauseException - lancée si la clause est une requête ou s'il s'agit d'un fait ou d'une règle non valide, ou si la clause n'a pas pu être trouvée dans la théorie;

Méthodes de résolution de requêtes

- public PrologResult find(PrologQuery) throws InvalidQueryException; :
Retourne la première solution de l'évaluation de la requête passée en argument. Cette requête deviendra alors la requête courante pour l'interpréteur Prolog abstrait.
In : PrologQuery - la requête à évaluer;
Out : PrologResult - le premier résultat de l'évaluation de la requête;
Pre : la requête est valide ;
Post : la requête est devenue la requête courante de l'interpréteur abstrait et le premier résultat de l'évaluation de la requête a été retourné;
Exception : InvalidQueryException - si la requête n'est pas valide (erreur de syntaxe...);

- public PrologResult findNext() throws QueryNotInitializedException; :
Retourne la solution suivante de l'exécution de la requête courante. Si aucune requête n'a été préalablement évaluée par un appel à *find(Query)*, une exception de type QueryNotInitializedException est générée. Chaque appel à *findNext()* portera sur la requête courante et celle-ci ne peut être modifiée qu'à travers l'invocation de *find()*.
In : void;
Out : PrologResult - le résultat suivant de la requête courante, un résultat nul [4] si aucune solution n'a pu être trouvée;
Pre : la requête courante de l'interpréteur a été initialisée;
Post : le résultat suivant de la requête courante, un résultat nul est retourné si aucune solution n'a pu être trouvée;

- public PrologResult findNext(PrologQuery) throws InvalidQueryException; :
Retourne une solution de l'évaluation de la requête passée en argument. Cette méthode est fournie pour les interpréteurs permettant l'exécution de plusieurs requêtes simultanées.
Cette méthode aura pour effet de créer plusieurs machines virtuelles Prolog, une par requête [5], à moins que l'implémentation de l'interpréteur ne fournisse un mécanisme pour la résolution en parallèle de plusieurs requêtes.
Cette méthode n'est que suggérée et aucunement obligatoire, elle n'est pas nécessairement supportée par tous les interpréteurs Prolog.
In : PrologQuery - la requête à évaluer;
Out : PrologResult - le résultat suivant de l'évaluation de la requête, un résultat nul si la requête n'a pas de solution;
Pre : la requête est valide;
Post : le résultat suivant de l'évaluation de la requête, ou un résultat nul si la requête n'a pas de solution, est retourné;
Exception : InvalidQueryException - lancée si la requête n'est pas valide;

[4] C'est-à-dire vide (isEmpty() retourne 'true') et sans solution (hasSolution() retourne 'false').

[5] Le même résultat peut être obtenu en créant, dans le programme Java, un interpréteur abstrait pour chaque requête et en chargeant dans chacun d'entre eux le même programme Prolog.

- public PrologResultSet findAll(PrologQuery) throws InvalidQueryException; :
Retourne un *PrologResultSet* contenant tous les résultats générés lors de l'évaluation de la requête passée en argument, ou un result set nul en cas d'échec de la requête. Si la requête à résoudre possède une infinité de solutions, l'invocation de la méthode va conduire à une boucle infinie.
In : PrologQuery - la requête à évaluer;
Out : PrologResultSet - l'ensemble des résultats générés par l'évaluation de la requête ou un result set nul si la requête a échoué;
Pre : la requête passée en argument est valide;
Post : l'ensemble des résultats générés par l'évaluation de la requête, ou un result set nul si la requête a échoué, a été retourné;
Exception : InvalidQueryException - lancée si la requête n'est pas valide;

A.2.3 Méthodes de la fabrique abstraite PrologProvider

Je vais maintenant donner la spécification des méthodes fabriques déclarées par l'interface *PrologProvider* et accessibles à travers la classe *PrologFactory*.

Méthodes fabriques de clauses

- public PrologRule newRule(java.lang.String) throws InvalidRuleException; :
Retourne un objet *PrologRule* créé à partir de la chaîne passée en argument. Celle-ci doit être de la forme $Head : -Body$ où $Head$, la tête de la règle, est une structure Prolog et *Body* une conjonction de structures Prolog constituant le corps de la règle. Un exemple de règle bien formatée est exemple, *father(X, Y):-male(X), parent(X,Y).*. Le point final peut-être omis.
In : java.lang.String - représentation sous forme de chaîne de caractères d'une règle;
Out : PrologRule - instance de règle Prolog correspondant à la chaîne passée en argument;
Pre : la chaîne passée en argument contient une règle bien formatée;
Post : l'instance de PrologRule correspondant à la chaîne passée en argument a été retournée;
Exception : InvalidRuleException - lancée si la règle passée en argument sous forme de chaîne de caractères est mal formatée;

- public PrologFact newFact(java.lang.String) throws InvalidFactException; :
Retourne un objet *PrologFact* créé à partir de la chaîne passée en argument. Celle-ci doit être de la forme *name*(*argList*)., par exemple, *father(paul, jean).*. Le point final peut-être omis.
In : java.lang.String - représentation sous forme de chaîne de caractères d'un fait;
Out : PrologFact - instance de fait Prolog correspondant à la chaîne passée en argument;
Pre : la chaîne passée en argument contient un fait bien formaté;
Post : l'instance de PrologFact correspondant à la chaîne passée en argument a été retournée;
Exception : InvalidFactException - lancée si le fait passé en argument sous forme de chaîne de caractères est mal formaté;

- public PrologQuery newQuery(java.lang.String) throws InvalidQueryException; :
Retourne un objet *PrologQuery* créé à partir de la chaîne passée en argument. Celle-ci doit être de la forme $? - Query$ ou *Query*, avec *Query* de la forme *nom*(*arglist*).. Par exemple, *?-frere(paul, jean).* est une requête bien formatée. Le point final peut-être omis.
In : java.lang.String - représentation sous forme de chaîne de caractères d'une requête;
Out : PrologQuery - instance de requête Prolog correspondant à la chaîne passée en argument;
Pre : la chaîne passée en argument contient une requête bien formaté;
Post : l'instance de PrologQuery correspondant à la chaîne passée en argument a été retournée;
Exception : InvalidQueryException - lancée si la requête passée en argument sous forme de chaîne de caractères est mal formatée ou si la requête est non valide pour l'interpréteur Prolog utilisé;

Méthodes fabriques de théories

- public PrologTheory newTheoryFromFile(java.lang.String) throws InvalidTheoryException; :
Retourne un objet *PrologTheory* créé à partir du nom de fichier passé en argument.
In : java.lang.String - chemin du fichier contenant la théorie (i.e. le programme Prolog).;
Out : PrologTheory - instance de théorie Prolog correspondant au chemin du fichier passé en argument;
Pre : le fichier passé en argument existe et contient une théorie syntaxiquement correcte [6];
Post : l'instance de PrologTheory correspondant au chemin du fichier passé en argument a été retournée;
Exception : InvalidTheoryException - lancée si le fichier passé en argument n'a pu être trouvé ou si le programme contenu n'est pas syntaxiquement correct;

- public PrologTheory newTheoryFromFile(java.io.File) throws InvalidTheoryException; :
Retourne un objet *PrologTheory* créé à partir du fichier passé en argument.
In : java.io.File - fichier contenant la théorie (i.e. le programme Prolog).;
Out : PrologTheory - instance de théorie Prolog correspondant au fichier passé en argument;
Pre : le fichier passé en argument existe et contient une théorie syntaxiquement correcte;
Post : l'instance de PrologTheory correspondant au fichier passé en argument a été retournée;
Exception : InvalidTheoryException - lancée si le fichier passé en argument n'a pu être trouvé ou si le programme contenu n'est pas syntaxiquement correct;

- public PrologTheory newTheoryFromString(java.lang.String) throws InvalidTheoryException; :
Retourne un objet *PrologTheory* créé à partir du texte du programme passé en argument.
In : java.lang.String - texte du programme Prolog.;
Out : PrologTheory - instance de théorie Prolog correspondant au texte passé en argument;
Pre : le texte passé en argument est un programme syntaxiquement correct;
Post : l'instance de PrologTheory correspondant au texte du programme passé en argument a été retournée;
Exception : InvalidTheoryException - lancée si le programme passé en argument n'est pas syntaxiquement correct;

- public PrologTheory newTheoryFromStream(java.io.InputStream) throws InvalidTheoryException; :
Retourne un objet *PrologTheory* créé à partir de la source de données (InputStream) passée en argument. Cette méthode et une généralisation de la méthode *newTheoryFromFile()* permettant d'utiliser n'importe quel flux d'entrée comme source d'un programme Prolog et, en particulier, l'entrée standard.
In : java.io.InputStream - flux d'entrée contenant la théorie (i.e. le programme Prolog).;
Out : PrologTheory - instance de théorie Prolog correspondant au programme contenu dans le flux d'entrée passé en argument;
Pre : le flux d'entrée passé en argument est accessible et contient une théorie syntaxiquement correcte;
Post : l'instance de PrologTheory correspondant au flux d'entrée passé en argument a été retournée;
Exception : InvalidTheoryException - lancée si une erreur s'est produite lors de l'accès au flux d'entrée passé en argument ou si le programme contenu n'est pas syntaxiquement correct;

[6] La notion de "syntaxiquement correct" peut dépendre de l'interpréteur Prolog utilisé par le mécanisme d'interopérabilité. En effet certains d'entre eux ne supportent que la programmation au premier ordre et/ou ne possèdent pas de prédicats extra-logiques... Ceci constitue le seul endroit où la transparence du mécanisme d'interopérabilité se trouve quelque peu mise en faute...

Méthode fabrique de l'interpréteur abstrait

- public PrologEngine newEngine(); :
Retourne une instance de l'interpréteur abstrait correspondant à l'implémentation de Prolog utilisée.
In : void;
Out : PrologEngine - l'interpréteur abstrait correspondant à l'implémentation de Prolog utilisée;
Pre : -;
Post : une instance de l'interpréteur abstrait correspondant à l'implémentation de Prolog utilisée a été retournée;

Méthode fabrique d'instance de la classe *PrologFactory* :

- public static PrologFactory newInstance(java.lang.String) throws UnknownProviderException; :
Retourne une instance de fabrique d'objets Prolog correspondant à la fabrique abstraite *PrologProvider* passée en argument en donnant son nom totalement qualifié. Le format d'un nom totalement qualifié en Java est *monpackage.MonProvider* où *monpackage* est le package contenant la classe *PrologProvider* à utiliser et *MonProvider*, le nom de la classe *PrologProvider* à utiliser. **Remarque :** le chemin vers la classe *PrologProvider* passé en argument doit être déclaré dans le chemin des classes (classpath) de la machine virtuelle Java.
In : java.lang.String - nom complètement qualifié de la fabrique abstraite *PrologProvider* à utiliser.;
Out : PrologFactory - une instance de fabrique d'objets Prolog utilisant la fabrique abstraite passée en argument;
Post : une instance de fabrique d'objets Prolog utilisant la fabrique abstraite passée en argument a été retournée;
Exception : UnknownProviderException - lancée si la classe correspondant au nom complètement qualifié passé en argument n'a pu être trouvée ou instanciée;

Annexe B

Liste d'interpréteurs Prolog et de leur mécanisme d'interfaçage avec Java

La liste suivante reprend quelques implémentations de Prolog, interfaçables avec Java. Pour chacune d'elles, elle indique le mécanisme d'interopérabilité mis en jeu.

Interpréteur	Techniques d'interopérabilité
Amzi	JNI
B-Prolog	JIPL (Java Interface to Prolog) - JNI
BinProlog	JNI
CIAO Prolog	basé sur l'utilisation des sockets
GNUProlog en Java	implémenté en Java
JavaLog	implémenté en Java
K-Prolog (JIPL)	JNI
InterProlog	sockets et sérialisation d'objets
JIProlog	implémenté en Java
jProlog	implémenté en Java
Kiev	langage hybride Java-Prolog compilé pour la JVM
Moksa Prolog	implémenté en Java
MINERVA	implémenté en Java
Prolog Cafe	compilateur Prolog vers Java
SICSTus (Jasper)	JNI
SWI Prolog (JPL)	JNI
tuProlog	implémenté en Java
W-Prolog	implémenté en Java

Remarque : "implémenté en Java" signifie que l'interpréteur concerné est écrit en langage Java, et que l'interopérabilité est réalisée à l'aide d'appels à des méthodes Java.

Annexe C

Code source de l’exemple

Voici le code source complet de l’exemple de l’interpréteur présenté au Chapitre 14.

```
/*
 * Created on 10-juil.-2003
 *
 * Copyright 2003 Minne Frédéric
 *
 * PrologInterpreter is part of java prolog interface.
 *
 * java prolog interface is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * java prolog interface is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 * along with PrologInterpreter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

import java.io.IOException;

import be.ac.ucl.info.blc.interop.engine.*;
import be.ac.ucl.info.blc.interop.exception.InvalidQueryException;
import be.ac.ucl.info.blc.interop.exception.InvalidTheoryException;
import be.ac.ucl.info.blc.interop.exception.NoMoreResultException;
import be.ac.ucl.info.blc.interop.exception.QueryNotInitializedException;
import be.ac.ucl.info.blc.interop.exception.UnknownProviderException;
import be.ac.ucl.info.blc.interop.struct.*;

/**
 * PrologInterpreter.java <br />
```

```
 * J2PI-Interpreter : A simple Prolog interpreter using the Java to Prolog
 * Interface and the Joshua Engel's Prolog compiler.
 * @author Minne Frédéric
 * @version 0.2
 */
public class PrologInterpreter
{
  // instance of the Prolog object factory
  private static PrologFactory factory = null;

  // fully qualified name of the concrete factory for
  // Engel's Prolog compiler
  private final static String DEFAULT_PROVIDER =
"be.ac.ucl.info.blc.interop.compiler.PrologCompilerProvider";

  // instance of the Prolog abstract engine
  private static PrologEngine engine = null;

  // set debug mode
  private static boolean debug = false;

  // represents interpreter commands
  private final static String DEBUG = "-debug";
  private final static String CONSULT = "consult";
  private final static String EXIT = "bye.";
  private final static String FINDALL = "findall";
  private final static String NEXT = ";";
  private final static String QUERY = "query";
  private final static String NONE = "none";
  private final static String COUNT = "count";
  private final static String HELP = "help.";
  private final static String ABOUT = "about.";

  private static boolean more = true;

  /**
   * Display the results contained in a PrologResultSet.
   * @param r result set to display.
   */
  public static void printResultSet(PrologResultSet r)
    throws NoMoreResultException
  {
    if (r.isEmpty())
    {
      String hasresult;
      hasresult = r.hasSolution() ? "Yes" : "No";
      System.out.println(hasresult);
    }
    else
    {
      while (r.hasNext())
      {
        PrologResult re = r.next();
        printResult(re);
        System.out.println();
      }
```

```
    }
  }

  /**
   * Display the content of a PrologResult.
   * @param r the result to display.
   */
  public static void printResult(PrologResult r)
  {
    if (r.isEmpty())
    {
      more = false;
      String hasresult;
      hasresult = r.hasSolution() ? "Yes" : "No";
      System.out.print(hasresult);
    }
    else
    {
      PrologVariable[] vars;
      vars = r.getSubstitution();
      int n = vars.length;
      System.out.println("Yes");
      for (int i = 0; i < n; ++i)
      {
        System.out.print(vars[i].getStringRepresentation());
        if (!(i == n - 1))
        {
          System.out.print(", ");
        }
      }
      more = true;
    }
  }

  /**
   * Display interpreter help screen.
   */
  private static void displayHelp()
  {
    System.out.println();
    System.out.println("J2PI-Interpreter version 0.2 : Help :\n");
    System.out.println("* use \'" + EXIT + "\' to exit J2PI-Interpreter.");
    System.out.println(
      "* use \'" + CONSULT + "(theory_file_path).\' to load a theory.");
    System.out.println(
      "* use \'"
        + FINDALL
        + "(query_to_find).\' to display all the result for the query.");
    System.out.println(
      "* use \'"
        + COUNT
        + "(query_to_find).\' to count the number of results of the query.");
    System.out.println(
      "* use \'" + HELP + "\' to display this help screen.");
    System.out.println();
  }
```

```
  /**
   * Display informations about J2PI-Interpreter.
   */
  private static void displayAbout()
  {
    System.out.println(
      "\nJ2PI-Interpreter version 0.2 is "
        + " (c) 2003 Frederic Minne under the Gnu General Public License "
        + "and uses parts of Oolong Java Assembler "
        + "(c) 1999 Joshua Engels under the Gnu General "
        + "Public License (see license notes for more details)\n");
    System.out.println("J2PI-Interpreter was design in 2003 "
        + "for the memoire \'Etude de l'interopérabilite de deux "
        + "langages de programmation basee sur la machine virtuelle de Java\'.");
    System.out.println();
  }

  /**
   * Load a theory into the interpreter
   * @param theory the name of the file containing the theory to load
   * @throws InvalidTheoryException if the theory is not valid
   */
  private static void consult(String theory)
    throws InvalidTheoryException
  {
    System.out.println("loading " + theory);

    // create an instance of PrologTheory
    PrologTheory t = factory.newTheoryFromFile(theory);

    // load the theory
    engine.consult(t);

    // loading succeed
    System.out.println("Yes");
  }

  /**
   * Find all solutions for the given query.
   * @param query the query to solve.
   * @throws InvalidQueryException if the query is not valid.
   * @throws NoMoreResultException from printResultSet
   */
  private static void findAll(String query)
    throws InvalidQueryException, NoMoreResultException
  {
    // create an instance of PrologQuery
    PrologQuery q = factory.newQuery(query);

    // get all solutions for the query
    PrologResultSet rs = engine.findAll(q);

    // display the solutions
    if (rs.hasSolution())
      System.out.println(rs.size() + " answers :");
```

```
    printResultSet(rs);
  }

  /**
   * Count the number of solutions for the given query.
   * @param query the query.
   * @throws InvalidQueryException if the query is not valid.
   */
  private static void count(String query)
    throws InvalidQueryException
  {
    // create an instance of
    PrologQuery q = factory.newQuery(query);

    // get all solutions for the query
    PrologResultSet rs = engine.findAll(q);

    // display the number of solution found
    System.out.println("Yes");
    System.out.println("count=" + rs.size());
  }

  /**
   * find the solutions for the given query one by one.
   * @param query the query to solve.
   * @throws InvalidQueryException if the query is not valid
   * @throws QueryNotInitializedException if the query is
   * not initialized (must not be thrown)
   * @throws IOException from System.in.read.
   */
  private static void find(String query)
    throws InvalidQueryException, QueryNotInitializedException,
      IOException
  {
    PrologQuery current = factory.newQuery(query);

    // find the first solution for the query
    PrologResult r = engine.find(current);
    System.out.print(
    "use \'"
      + NEXT
      + "\' to get the next result, \'.\' to return to prompt.\n\n");

    // display the solution
    printResult(r);

    // ask user for more solutions
    StringBuffer nxt;
    for (;;)
    {
      nxt = new StringBuffer();
      for (;;)
      {
        char c = (char) System.in.read();
        if (c == '\n')
          break;
```

```
        if (c == '\r')
          continue;
        nxt.append(c);
      } //for

      // if another solution is needed
      if (nxt.toString().equals(NEXT))
      {

        // find the next solution for the query
        PrologResult r2 = engine.findNext();

        // display it
        printResult(r2);

        // if there is no more solution exit loop
        if (!more)
        break;
      } //if
      else
      {
        more = false;
        break;
      } //else
    } //for
  }//find

  /**
   * Main method for the interpreter.
   * @param args ignore the command line arguments.
   */
  public static void main(String[] args)
  {
    String provider = DEFAULT_PROVIDER;

    int argc = args.length;
    switch (argc)
    {
      case 0 :
        break;
      case 1 :
        if (args[0].equals(DEBUG))
        {
          debug = true;
          break;
        }
      default :
        System.out.println(
          "Usage : "
            + "java PrologInterpreter options");
        System.out.println(
          "Options : \n \t -debug : debug mode");
        System.exit(0);
    }

    try
```

```
{
  // instanciate the Prolog object factory
  // with the Joshua Engel's
  // Prolog compiler concrete factory.
  factory = PrologFactory.newInstance(provider);
}
catch (UnknownProviderException e1)
{
  System.err.println(e1.getMessage());
  System.exit(-1);
}
// get the abstract Prolog interpreter
// for the given concrete factory
engine = factory.newPrologEngine();

System.out.println(
  "\nJ2PI-Interpreter : A simple Prolog interpreter"
    + "\nusing Java to Prolog Interface\n");
System.out.println("\tuse \'" + HELP + "\' to get some help;");
System.out.println("\tuse \'" + ABOUT + "\' to"
  + "get more information about J2PI-Interpreter;");
System.out.println("\tuse \'" + EXIT + "\' to quit.\n");

// parse command loop
for (;;)
{
  try
  {
    System.out.print("?-");
    StringBuffer sb = new StringBuffer();
    // read command
    for (;;)
    {
      char c = (char) System.in.read();
      if (c == '\n')
        break;
      if (c == '\r')
        continue;
      sb.append(c);
    }
    String command = sb.toString();
    // check command formating
    if (!(command.endsWith(".")))
    {
      System.out.println("Error, command must end with \'.\' ");
      continue;
    }
    // command = bye. ?
    if (command.equals(EXIT))
    {
      System.out.println(
        "Thanks for using J2PI-Interpreter.\nBye !");
      // exit
      break;
    }
    // command = help. ?
```

```
        if (command.equals(HELP))
        {

          // display help screen
          displayHelp();
          continue;
        }
        // command = about ?
        if (command.equals(ABOUT))
        {

          // display information screen
          displayAbout();
          continue;
        }
        // command is a consult command
        if (command.startsWith(CONSULT))
        {
          // get the theory to load
          String theory =
            command.substring(
              CONSULT.length() + 1,
              command.length() - 2);

          // call to consult method
          consult(theory);

          System.out.println();
          continue;
        }
        // command is a findall command ?
        if (command.startsWith(FINDALL))
        {
          // get the query
          String query =
            command.substring(
              FINDALL.length() + 1,
              command.length() - 2);

          // find all solutions
          findAll(query);

          System.out.println();
          continue;
        }
        // command is a count command ?
        if (command.startsWith(COUNT))
        {
          // get the query
          String query =
            command.substring(
              COUNT.length() + 1,
              command.length() - 2);

          // call the count method
          count(query);
```

```
          System.out.println();
          continue;
        }

        // the command is a query
        // create a new instance of PrologQuery

        // call the find method
        find(command);

        System.out.println();
      } //try
      // if an error occurs
      catch (Exception e)
      {
        System.out.println("No");
        System.err.println("ERROR:" + e.getMessage());
        if (debug)
          e.printStackTrace();
        continue;
      }
    } //for
  } //main
}//class
```