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

J2PI :

A terminer :

- Nouveaux constructeurs pour les théories;
- Classe *PrologTerm* et ajout des méthodes *assertz(Term)* et *revoke(Term)*;
- Documentation de l'interface;
- Documenter les conversions de types;
- Implémentation à l'aide de tuProlog (en cours);
- Nettoyer le code;

Améliorations :

- Modification du constructeur de requêtes pour qu'il puisse accepter n'importe quel type Java : *newPrologStructure(String name, PrologVariable[] args)* et *newQuery(PrologStructure c)*;
- Passage d'objets Java au moteur Prolog;
- Gestion des exceptions;
- Amélioration et correction des bugs du compilateur de Engel;

Les améliorations seront documentées mais pas nécessairement implémentées...

P2JI :

Conversion :

trois niveaux en réalité mais seuls deux sont visibles par le programmeur :

1. types Prolog;
2. types manipulés par les extensions;
3. types Java "natifs" du moteur Prolog;

les niveaux 1 et 2 sont connus du programmeur alors que le niveau 3 est interne à l'interface.

conversion (provisoire) :

- term -> PrologTerm
  - foncteur -> PrologStructure
  - liste -> PrologList
  - variable -> PrologVariable
  - atome -> PrologAtom
    - string ou char -> PrologString wrapper pour java.lang.String
    - number -> PrologNumber wrapper pour java.lang.Number
      - integer -> PrologInt (32 bits) wrapper pour java.lang.Integer ou int (je n'ai pas encore choisi et la conversion de l'un à l'autre est facile)
      - float -> PrologFloat (32 bits) wrapper pour java.lang.Float ou float (idem que pour integer...)

Remarque c'est différentes classes seront des *Wrapper* contenant des objets du type Java correspondant et des méthodes pour les conversions (soit méthodes d'instance soit méthodes statiques...)

Interface :

Se limitera aux appels à des méthodes de librairies statiques (à définir)...

Éléments :

1. un IDL (voir questions et solutions) ou une directive de déclaration d'un appel Java : *import (FUNCTOR, class, method)*. Par exemple : import(fact/2, lib.Math, fact);
2. utilisation d'un précompilateur qui va créer une extension à partir de la directive d'importation en faisant appel à une *ExtensionFactory* (Abstract Factory Pattern) qui cachera les détails d'implémentation propres à chaque moteur Prolog et dont le rôle sera d'emballer la méthode Java

appelée dans une extension utilisable par le moteur Prolog. La plupart de ceux-ci fournissent de telles interfaces d'extension.

Questions :

1. Comment paramétrer le précompilateur pour qu'il sache quel moteur Prolog utiliser ?
2. Comment obtenir la signature de la méthode Java à appeler (on en a besoin pour appeler les méthodes de conversion) ?

Solutions :

1. Une autre directive *precomp(PROVIDER)* ?
2. IDL... j'y ai déjà songé mais je n'ai pas encore de solution vraiment claire -> en discuter... une autre solution, plus lourde, est l'utilisation de l'API de reflection... Le problème est alors de construire le parser pour l'IDL... ou d'utiliser XML.

IDL :

Le IDL devrait contenir :

- une partie déclaration où l'on indique la méthode Java à appeler;
- une partie déclaration de types;
- (opt.) une partie usage où l'on indique, si nécessaire différents usages pour le prédicat;

une telle description IDL pourrait ressembler à :

```
import{
  functor:sum/3;
  types{
    _1:integer;
    _2:integer;
    _3:integer;
  }
  method:lib.Math.sum(_1, _2);
}
```

ou, si les usages sont utilisés :

```
import{
  functor:sum/3;
  types{
    _1:integer;
    _2:integer;
    _3:integer;
  }
  usages{
    ?_1?_2!_3:lib.Math.sum(_1,_2);
    ?_1!_2?_3:lib.Math.diff(_3,_1);
    !_1?_2?_3:lib.Math.diff(_3,_2);
  }
}
```

ou, en XML :

```
<import functor="sum/3">
  <types>
    <param name="_1" type="integer" />
    ...
  </types>
  <method>
    lib.Math.sum(_1, _2)
  </method>
</import>

<import functor="sum/3">
  <types>
    <param name="_1" type="integer" />
    ...
  </types>
```

```
    <usages>
      <usage condition="?_1?_2!_3" call="lib.Math.sum(_1,_2)" />
      ...
    </usages>
  </import>
```

La partie types permettrait d'appeler les bonnes méthodes de conversion...

Problème qui peut se poser : unification : ce que l'on doit faire n'est pas *_3 = lib.Math.sum(_1, _2)* mais *unify(_3, lib.Math.sum(_1,_2))* :

- soit ils sont gérer par le moteur;
- soit c'est l'interface qui s'en charge;
- soit on les oublie (si aucune des deux solutions ci-dessus ne peut être appliquée) et on le documente...

Améliorations :

- Ajout de la création d'objets Java depuis Prolog;
- Gestion des exceptions;
- ...

Ne seront pas implémentées, juste discutées...

Perspectives :

On pourrait partant de l'interface définie ici penser à réaliser un langage hybrides OO-Logique basé sur Prolog et Java...

Rédaction :

Plan :

I. **Intro** : en gros elle est terminée, il manque juste une petite discussion sur la granularité des requêtes au moteur Prolog
II. **Spécification** de l'interface : ici plusieurs parties :
    1. Une petite introduction pour présenter la spécification et expliquer qu'on va avoir deux interfaces
    2. J2PI : interface Java vers Prolog, là tout est quasiment terminé, il faut juste remettre à jour les spécification des méthodes et ajouter une section sur l'architecture et l'intuition qui lui est liée (moteur Prolog générique, a priori on ne parle pas du pattern ici);
    3. P2JI : ici tout reste à faire, suivre un plan comparable a J2PI
III. **Implémentation** de l'interface : Devrait prendre la forme d'un guide d'utilisateur ?
    1. J2PI : décrire le pattern et son implémentation, dire comment on l'instancie, parler de wrapper/factory et justifier et donner les guides d'implémentation pour le programmeur;
    2. P2JI : décrire l'architecture et dire comment on l'instancie...
IV. **Exemples** :
    1. Exemples pour J2PI : Interpréteur, Tour de Hanoi, Analyse Syntaxique;
    2. Exemples pour P2JI : Tour de Hanoi, Analyse Syntaxique;
    3. Comparer les deux versions de la Tour de Hanoi;
V. **Conclusions** et **perspectives** :
    1. Comparaison 1 – 2 langages, pourquoi est-ce intéressant ?;
    2. Exemples pratiques (analyse syntaxique, traitement de la langue naturelle, gestion des interface réseau dans Windows NT...);
    3. Perspectives :
        i. Que se passe-t-il si plusieurs niveaux (+/-) ?;
        ii. Vers un langage hybride...;
VI. **Annexes** :
    1. Implémentation avec Oolong Prolog : problèmes et solutions;
    2. Implémentation avec tuProlog : problèmes et solutions;
    3. Bibliographie;
    4. Éventuellement un ou deux textes qui pourraient être utiles;
    5. Code source;