Une technique générique de réification pour les langages à objets

Rémi Douence, Mario Südholt

École des Mines de Nantes, Département Informatique, 44307 Nantes cedex 3, France
Tel : +33 2 51 85 82 15, Fax : +33 2 51 85 82 49, {douence,sudholt}@emn.fr

Résumé

La réflexion gagne du terrain dans les applications pratiques comme attesté par la présence d'une API réflexive dans l'environnement de programmation JAVA et les récents travaux sur le middleware réflexif. Les systèmes réflexifs offrent de nombreuses interfaces de programmation, aussi connues sous le nom de protocole métaobjet (MOP). Leur conception dépend de certaines contraintes comme l'expressivité, l'efficacité et la sécurité. Puisque ces contraintes sont différentes d'une application à l'autre, nous devons être capables de fournir facilement des MOP sur mesure pour des ensembles de contraintes donnés .

Dans cet article, nous présentons une technique générique de réification à base de transformation de programme. Elle permet la réification sélective d'une partie arbitraire d'un interprète de langage à objets. La transformation de programme peut aussi être appliquée à différents interprètes. Chaque implémentation réflexive résultante fournit un MOP sur mesure différent dérivé directement de la définition originale de l'interprète.

Sujets : génie logiciel, architecture logicielle, paradigme de programmation, réflexion, transformation de programme.

1 Introduction

La réflexion, c'est à dire, la capacité d'un système à s'inspecter et se modifier lui même à l'exécution, gagne du terrain dans les applications pratiques : les logiciels modernes requièrent fréquemment de fortes propriétés d'adaptation dues à l'hétérogénéité et la dynamicité des environnements. La réflexion permet, par exemple, de déterminer dynamiquement les services offert par un hôte et autorise la modification de protocoles d'interaction à l'exécution. Concrètement, l'environnement de programmation JAVA [8] repose sur la réflexion pour l'implémentation du modèle de composant JAVABEANS et du mécanisme d'invocation de méthode à distance. De plus, l'adaptabilité est une propriété essentielle des systèmes middleware et plusieurs groupes effectuent actuellement des recherches sur le middleware réflexif [13].

Les systèmes réflexifs offrent de nombreuses interfaces de programmation, aussi connues sous le nom de protocole métaobjet (MOP). La conception d'un tel MOP est liée à certaines contraintes comme le pouvoir d'expression, l'efficacité et la sécurité. Par exemple, l'utilisation de la réflexion à des fins de déverminage peut nécessiter que le MOP fournisse un accès à la pile d'exécution. Le modèle de réflexion de JAVA, cependant, n'autorise pas que la pile (non typée) soit modifiée à l'exécution pour des raisons de sécurité.

Puisque ces contraintes différent d'une application à l'autre, nous devrions être capable de fournir un MOP sur mesure pour chaque ensemble particulier de contraintes. De plus, les contraintes changent fréquemment tout au long du cycle de vie d'un logiciel. Le développement d'un tel MOP sur mesure devrait donc être un processus léger. Les approches traditionnelles n'atteignent pas ce but : chacune fournit seulement un MOP spécifique qui peut rarement et difficilement être modifié (cf. la discussion sur les travaux connexes en Section 6).

Dans cet article, nous présentons un mécanisme générique de réification à base de transformation de programme pour des interprètes de langage à objets. Ce mécanisme peut être appliqué à différentes parties d'un interprète non-réflexif afin de générer des interprètes de plus en plus réflexif. Il peut aussi être

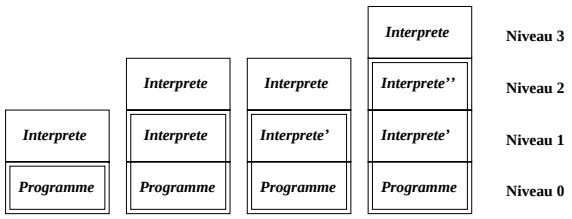

FIG. 1 – tours réflexives de Smith

appliqué à différents interprètes afin d'obtenir automatiquement différents interprètes réflexifs. Chaque implémentation réflexive résultante fournit un MOP différent directement dérivé de la définition originale de l'interprète.

L'article est structuré comme suit : après une brève introduction des tours réflexives de Smith sur lesquelles notre travail est basé, la Section 2 présente l'architecture de notre système à base de transformation. Notre technique générique de réification est détaillée dans la Section 3. La Section 4 présente plusieurs exemples de programmation réflexive à l'aide de METAJ [12], un interprète réflexif pour JAVA implementé à l'aide de notre technique. Le Section 5 illustre comment un raffinement de l'interprète non-réflexif produit un interprète réflexif plus expressif. Nous terminons l'article par une discussion de travaux connexes et une conclusion.

La réflexion selon Smith. Le travail de Smith sur 3-Lisp [16], une référence à cause de sa généralité et de ses bases sémantiques claires, définit la réflexion avec la notion de tours d'interprètes.

Dans la Figure 1, la tour de gauche montre un *Programme* utilisateur (i.e. au niveau 0) et son *Interprète*. A l'exécution, un calcul réflexif crée une couche d'interprétation supplémentaire afin que l'*Interprète* du niveau 1 fasse désormais partie intégrante du programme (il est inclus dans la double boite). Le *Programme* peut maintenant modifier la sémantique défini par l'*Interprète* du niveau 1 pour obtenir *Interprète'* comme montré dans la troisième tour. Enfin, quand une sémantique non-standard *Interprète"* de l'*Interprète'* est requise, un nouveau niveau d'interprétation peut être introduit comme illustré par la quatrième tour.

Un exemple classique de programmation réflexive consiste à introduire des traces pour le déverminage. Dans la Figure 1, l'*Interprète'* pourrait générer des traces alors que *Programme* est évalué. La nouvelle couche d'interprétation *Interprète"* de la quatrième tour serait requise pour tracer l'*Interprète'*.

2 Architecture du processus de réification

La réflexion selon Smith est construite sur la base d'un interprète non-réflexif. Aussi, nous avons implémenté en JAVA un interprète non-réflexif pour JAVA. Nous n'avons pas mis en oeuvre tout le langage, omettant des constructions non essentielles comme p.ex. les entiers, les flottants et les boucles, leur intégration ne posant pas de nouveaux problèmes.

Comme montré dans la Figure 2, notre interprète non-réflexif de JAVA prend en entrée un programme non-réflexif `Prog.java`. En fonction des capacités réflexives souhaitées, le concepteur de langage transforme un sous ensemble de l'interprète non-réflexif. Cette transformation introduit deux classes pour chaque classe transformée. Par exemple, la classe `Instance.java`, qui représente les instances dans l'interprète, devient `BaseInstance.java` et une version différente de `Instance.java`.

L'interprète réflexif se base sur la définition des classes transformées pour construire les niveaux de la tour réflexive. Ceci est au coeur de notre approche : les niveaux de la tour sont *effectivement batis* sur la base de la *définition de l'interprète* comme dans le modèle de Smith. En effet, `BaseInstance.java` est une entrée de l'interprète réflexif dans la Figure 2. Ainsi, la sémantique opérationnelle de l'interprète réflexif est formellement dérivée de la sémantique du non-réflexif. De plus, notre approche est sélective et complète car la transformation est applicable à n'importe qu'elle classe de la définition de l'interprète non-réflexif.

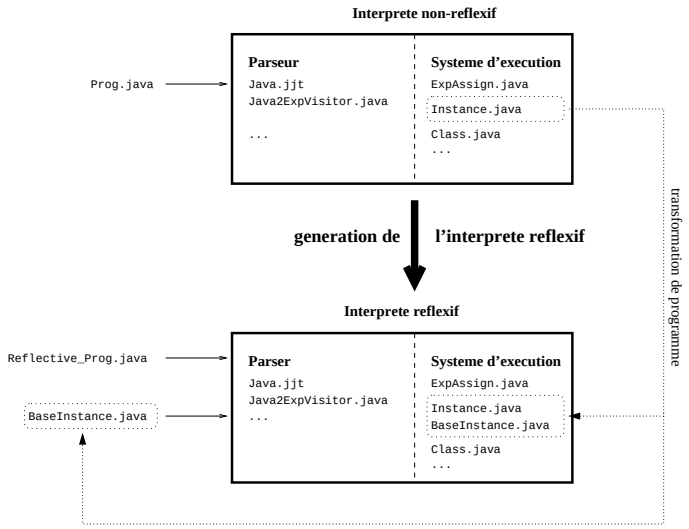

FIG. 2 – Architecture

Nous avons mis en oeuvre une version de cette architecture, l'interprète réflexif JAVA résultant est appelé METAJ [12].

3 Réification générique par transformation de code

Dans cette section, nous donnons une vue d'ensemble de notre technique générique de réification, qui peut être appliquée à n'importe quelle classe définissant l'interprète. Ensuite nous l'appliquons en détail à la classe `Instance` qui implémente des instances dans l'interprète.

3.1 Vue d'ensemble de la technique générique de réification

La réification d'un objet ne doit pas changer sa sémantique mais seulement sa représentation et offrir un accès à cette nouvelle représentation. Par exemple, il est impossible d'accéder à la liste des champs d'un objet avec notre interprète non-réflexif (bien qu'une telle liste existe dans la mémoire de l'implémentation sous jacente). La représentation réifiée d'un objet fournit un accès à cette liste. Une fois que la représentation interne a été exposé, l'accès à cette structure permet de modifier la sémantique de l'objet (p. ex. en ajoutant un nouveau champ à la liste). Notez que cette forme de réification structurelle de la mémoire de l'interprète permet de réaliser les notions traditionnelles de réflexion structurelle et comportementale.

Pour illustrer cette idée, considérons un objet `pair` avec deux champs `fst` et `snd` qui est implémenté dans la mémoire de l'interprète par une $\overline{\texttt{Instance}}$ (par la suite, $\overline{\texttt{C}}$ dénote une instance de la classe `C`). Essentiellement, une entité réifiable peut avoir deux représentations différentes (cf. Figure 3) : soit une représentation de base, soit une représentation réifiée. Puisque la réification d'un objet ne change pas sa sémantique, l'objet doit fournir la même interface (i.e. les mêmes méthodes) dans les deux représentations. Cette interface commune est implémentée à l'aide d'un objet aiguillage [1] : l'$\overline{\texttt{Instance}}$ dénotée par `pair`.

L'objet aiguillage pointe vers la représentation active : soit la représentation de base ($\overline{\texttt{BaseInstance}}$ dans la Figure 3a) soit la représentation réifiée ($\overline{\texttt{Instance}}$ dénotée par `pair. reify()` *et* $\overline{\texttt{BaseInstance}}$ dans la Figure 3b). L'objet aiguillage exécute les appels de méthode entrants en fonction de la représentation active : quand la représentation de base est active, l'aiguillage *délègue* simplement l'appel de méthode entrant à la représentation. Quand la représentation réifiée est active, l'aiguillage *interprète* l'appel de méthode.

[1] La technique d'aiguillage est proche des patrons de conception *bridge* et *state* introduits dans [6].

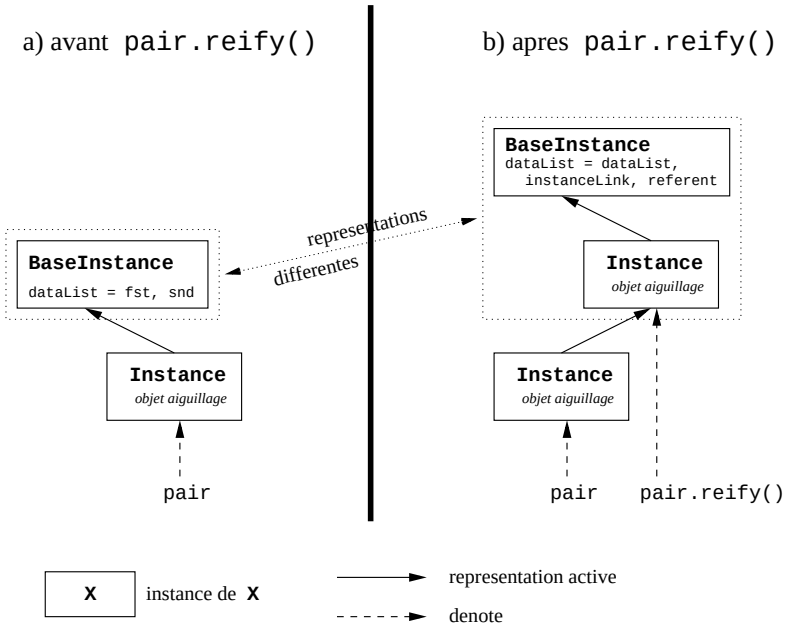

FIG. 3 – Avant et après la réification de l'objet `pair`

```
class Instance {
  public Class instanceLink; // ref. to Class
  public DataList dataList;  // field list
  Instance(Class instanceLink, DataList dataList) {
    this.instanceLink = instanceLink;
    this.dataList = dataList; }
  // field access
  Data lookupData(String name) {
    return this.dataList.lookup(name); }
  ...
}
```

FIG. 4 – Original class `Instance`

Qu'un objet soit accédé à travers son aiguillage ou par sa représentation réifiée n'est pas significatif, c'est à dire, les modification de l'objet en passant par `pair` sont visibles en passant par `pair.reify()`. Évidemment, les deux accès fournissent des interfaces différentes : p.ex. le champ `fst` de `pair` peut être accédé soit par `pair.fst` soit par `pair.reify().dataList. lookup("fst")` dans l'exemple. Cette propriété est communément référencée comme la connexion causale entre niveaux.

3.2 Exemple : rendre `Instance` réifiable

Basé sur la technique d'implémentation précédemment introduite, notre technique de réification est une transformation de programme qui peut être appliquée à n'importe qu'elle classe de l'interprète original. Nous détaillons ici le cas de `Instance` (cf Figure 4), la transformation générale est définie formellement dans [5]. Elle a deux grandes étapes :

1. Introduire la classe `BaseInstance` (cf. Figure 5) qui définit la représentation de base de la classe originale `Instance`.
2. Redéfinir la classe `Instance` (cf. Figure 6) afin qu'elle implémente l'aiguillage correspondant.

```
class BaseInstance {
  Class instanceLink;
  DataList dataList;
  Instance referent;
  BaseInstance (Class instanceLink, DataList dataList,
                Instance referent) {
    this.instanceLink = instanceLink;
    this.dataList = dataList;
    this.referent = referent;}
  Data lookupData(String name) {
    return this.dataList.lookup(name); }
  ...
}
```

FIG. 5 – Class `BaseInstance`

> Cette classe offre les mêmes méthodes que la classe originale `Instance` et implémente de plus une méthode `reify()` qui crée la représentation réifiée et bascule de la représentation de base à la représentation réifiée.

La Figure 5 détaille la classe de base générée : la classe originale est renommée et un champ `referent` est introduit. Dans la mémoire un objet est implémenté par un aiguillage *et* une représentation active. Le champ `referent`, qui est initialisé dans le constructeur et pointe en retour de la représentation vers l'aiguillage, permet de distinguer l'aiguillage de la représentation : si `this` n'est pas suivi par un point (utilisé pour accéder à un champ ou une méthode) dans la classe originale, alors il doit être remplacé par `this.referent` dans la classe de base afin de dénoter l'aiguillage. Ce probléme est typique des techniques à base de wrappers qui introduisent deux identités pour un objet.

La classe aiguillage générée, montrée dans la Figure 6, a deux champs : `representation` qui pointe soit vers la représentation de base soit vers celle réifiée, et un champ booléen `isReified` qui mémorise quelle est la représentation active. Son constructeur crée une représentation de base par défaut.

La méthode `lookupData` de l'aiguillage a la même signature que sa version originale. Quand la représentation de base est active active (i.e. `isReified` est faux), l'appel de méthode est délégué à la représentation de base. Quand la représentation réifiée est active, (i.e. `isReified` est vrai), l'appel de méthode est interprété : l'expression correspondante est syntaxiquement analysée (`Parser.java2Exp()`), un environnement local est construit (`argsE.add()`) à partir des arguments et du champ `representation` de l'aiguillage, puis l'appel de méthode est évalué (`eval()`). Notez que par soucis de simplicité ce code est volontairement naïf. L'implémentation effective peut être optimisée : par exemple, l'appel à l'analyseur syntaxique peut être remplacé par l'arbre de syntaxe correspondant.

La méthode `reify()` construit une représentation réifiée de la représentation de base en interprétant une expression `new BaseInstance()`. L'objet `Class` définissant la classe `Instance` est clonée afin que la comportement de chaque objet réifié puisse être spécialisé indépendamment. (De plus, partager les classes peut introduire des cycles lors de la réification : notamment pour la classe `Class`.) Si du partage est requis le programmeur d'application peut le réaliser en manipulant explicitement des références. Enfin, la représentation réifiée est installée comme représentation active et sa référence est retournée.

Notez que la définition textuelle de la classe `BaseInstance` a été lu et stoquée dans `Main. globalE` au démarrage de l'interprète. De cette façon `reify()` crée une couche supplémentaire d'interprétation réellement basée sur la définition de l'interprète.

Cette technique (automatisable) de réification repose sur seulement deux hypothèses :

1. Chaque construction syntaxique de JAVA est représentée par une expression appropriée dans l'interprète. Nous supposons que toutes ces expressions peuvent être évaluées en utilisant la méthode `eval(argsE)` où `argsE` contient les valeurs des variables libres dans l'expression courante.
2. Nous supposons que les définitions en JAVA de toutes les classes réifiables ont été lues et stoquées comme des objets `Class` dans l'environnement `globalE`. ces objets doivent être clonables.

```
class Instance {
  Object representation;
  boolean isReified;
  Instance(Class instanceLink, DataList dataList) {
    this.isReified = false;
    this.representation =
      new BaseInstance(instanceLink, dataList, this);
  }
  Data lookupData(String name) {
    if (this.isReified) {
      // interpret lookup method call
      Exp exp = Parser.java2Exp("reifiedRep.lookupData(name)"));
      // pass already evaluated values
      Environment argsE = new Environment(null, null, null);
      argsE.add("name", name);
      argsE.add("reifiedRep, this.representation);
      Data result = exp.eval(argsE);
      // unpack result
      return (Data)result.read();
    } else
      return ((BaseInstance)this.representation).lookupData(name);
  }
  ...
  Data reify() {
    if (!this.isReified) {
      // copy the base class BaseInstance
      Class baseInstance= Main.globalE.lookup("BaseInstance").clone();
      // create and initialize new representation
      Exp exp = Parser.java2Exp("new BaseInstance(baseRep_instLink,
                                                  baseRep_dataList,
                                                  baseRep_referent)");
      Environment argsE = new Environment(null, null, null);
      argsE.add("BaseInstance", baseInstance);
      argsE.add("baseRep_instLink", this.representation.instanceLink);
      argsE.add("baseRep_dataList", this.representation.dataList);
      argsE.add("baseRep_referent", this.representation.referent);
      Data result = exp.eval(argsE);
      // set interface state
      this.isReified = true;
      this.representation = result.read();
    }
    return new Data(this.representation);
  }
}
```

FIG. 6 – Dispatch class `Instance`

```
class Pair {
  String fst, snd;
  Pair(String fst, String snd) {
    this.fst = fst;
    this.snd = snd; }
  String toString() {
    return "(" + this.fst + "," + this.snd + ")"; }
}
class BaseInstanceWithTrace extends BaseInstance {
  Method lookupMethod(String name) {
    // trace method-called
    System.out.println("method called : " + name);
    return this.instanceLink.methodList().lookup(name); }
}
class Main {
  void main() {
    Pair pair = new Pair("1", "2");
   // 1 - introspection : test existence of a field
    BaseInstance metaPair = pair.reify();
    System.out.println(metaPair.dataList.member("third"));
   // 2 - intercession : change method-call semantics
    BaseInstance metaMetaPair = metaPair.reify();
    metaMetaPair.instanceLink = BaseInstanceWithTrace;
    System.out.println(pair.toString()); }
}
```

FIG. 7 – Programmation réflexive avec la classe Pair

4 Programmation réflexive

Dans cette section, nous exprimons plusieurs exemples classiques de programmation réflexive dans notre cadre. Ces exemples de notre interprète réflexif en action devraient aider le lecteur à maîtriser les rouages du système.

Considérons le problème de tester l'existence d'un champ. Dans la Figure 7, une classe `Pair` est définie et une instance `pair` est créée. Dans l'interprète, l'objet `pair` est présenté par une `Instance` (voir Figure 3a). Notre méthode générique de réification fournit l'accès à une représentation de cette `Instance` que nous nommons `metaPair` (`pair.reify()` dans la Figure 3b). Les champs d'une instance sont stoquée dans une `DataList` fournissant une méthode `Boolean member(String dataName)` qui permet de vérifier l'appartenance d'un champ nommé `dataName` dans la liste courante.

Notre second exemple introduit des traces d'appel de méthode pour le déverminage. La classe `BaseInstance` de l'interprète définit la méthode `Method lookupMethod(String name)` qui retourne la méthode à appeler. Les appels de méthodes peuvent être tracés en définissant une classe `BaseInstanceWithTrace` dont la méthode `lookupMethod()` affiche le nom de ses paramètres. Afin que les appels de méthode sur l'instance `pair` soient tracés, la classe `BaseInstance` doit être dynamiquement remplacé par `BaseInstanceWithTrace`. Une réification de `pair` fournit un accès à une `Instance` dont le champ `instanceLink` dénote la classe `Pair`. Une séquence de deux réification sur `pair` fournit l'accès à une `Instance` dont `instanceLink` dénote la classe `BaseInstance`. Ce lien peut alors être modifié pour dénoter la classe `BaseInstanceWithTrace`. Un appel de méthode de l'objet `pair` affiche alors le nom de la méthode.

5 Générer différents MOP

Nous avons déjà mentionné que le choix d'un ensemble de classes réifiables détermine complètement un MOP. Nous pensons que c'est une propriété clé de notre approche car elle fournit une base au déve-

```
class Instance {
    ...
    // add two new methods
    Data send(Msg msg) {
      return msg.to.receive(msg); }
    Data receive(Msg msg) {
      return msg.to.lookupMethod(msg.methodId)
               .apply(msg.argsE, msg.to); }
}
class ExpMethod extends Exp {
    ...
    Data eval(Environment localE) {
      // as before evaluate receiver and arguments : o, argsE
      ...
      // new code : determine sender, build and send message
      Instance self = (Instance)(localE.lookup("this").read());
      Msg msg = new Msg(self, o, this.methodId, argsE);
      return self.send(msg); }
}
```

FIG. 8 – Nouvel interprète original

```
class BaseInstanceWithSenderTrace extends BaseInstance {
    Data send(Msg msg) {
      System.out.println("method called " + msg.methodId
                         + " by " + msg.from);
      return super.send(msg); }
}
```

FIG. 9 – Extension (utilisateur) de `BaseInstance`

loppement systématique de MOP sur mesure. Dans cette section, nous modifions l'envoi de message dans l'interprète non-réflexif afin d'obtenir un MOP de grain plus fin qui distingue l'émetteur du récepteur d'un message.

Dans l'interprète original, `ExpMethod.eval()` évalue un appel de méthode en implémentant la séquence `lookupMethod();apply()`. Aussi, le comportement du recepteur d'un appel de méthode peut etre facilement modifié en changeant la définition de `lookupMethod()` (comme illustré par l'insertion de traces dans la section précédente). Cependant, une modification concernant l'émetteur d'un appel de méthode (voir [1] pour une motivation de rendre l'émetteur explicite dans le contexte de la programmation distribuée) est beaucoup plus difficile à réaliser. Un tel changement nécessiterait la modification de toutes les instances de `ExpMethod`, i.e. toutes les occurrences de l'opérateur `'.'`, dans l'arbre de syntaxe abstraite. En effet, nous devons vérifier si l'objet désigné par `this` dans ce contexte a un comportement non-standard.

Une solution à ce problème consiste à modifier l'interprète non-réflexif afin que sa version réflexive fournisse un MOP permettant l'accès à l'émetteur d'un appel de méthode. Intuitivement, nous divisons l'envoi de message en deux parties : le coté émetteur et le coté récepteur. D'abord, nous introduisons une nouvelle classe `Msg`, essentiellement un quadruplet contenant, pour chaque appel de méthode, l'émetteur `from`, le récepteur `to`, le nom de méthode `methodId` et les arguments correspondants `argsE`. Ensuite, deux méthodes sont ajoutées à la définition de `Instance` dans l'interprète original : `send()` et `receive()` pour gérer les messages (voir Figure 8). Enfin, `ExpMethod.eval()` est modifiée afin de créer et d'envoyer un message au récepteur.

Cette nouvelle version de l'interprète non-réflexif est rendu réflexive en appliquant la même transformation de programme. De la même façon que dans la section précédente, l'utilisateur peut alors introduire, par exemple, des traces dans les émetteurs (voir Figure 9) ou changer un appel local de méthode en appel distant.

Cet exemple souligne deux avantages de notre approche : les programmeurs d'applications peuvent toujours travailler avec le MOP minimal adapté à leurs besoins et le concepteur de langage peut étendre le MOP incrémentalement selon les besoins des programmeurs.

6 Travaux connexes

Une comparaison entre différents systèmes réflexifs est intrinsèquement difficile à cause de la grande variété et de la complexité conceptuelle des modèles de réflexion et des implémentations réflexives. Par exemple, la définition détaillée du MOP de CLOS nécessite un livre complet [9] et la comparaison entre CLOS et SMALLTALK a déjà fait l'objet d'un chapitre de livre [4].

En conséquence, nous restreignons notre comparaison à trois propriétés de base que notre modèle de réflexion vérifie (la première et la deuxième caractérisent l'approche de Smith, la troisième est fondamentale pour notre but de construction de MOP sur mesure) :

1. *(tour)* Il y a une tour potentiellement infinie d'interprètes.
2. *(sémantique)* L'interprète au niveau n interprète *le code* de l'interprète au niveau $n-1$.
3. *(sélectivité & complétude)* Tout objet dans la définition d'un interprète au niveau n peut être réifié et a une représentation accessible au niveau $n+1$.

En ce qui concerne la première propriété, la plupart des systèmes réflexifs sont basés sur des tours et fournissent un nombre potentiellement infini de niveaux. Deux exceptions notables sont OPEN-C++ [15] et IGUANA [7] dont les MOP offrent seulement un métaniveau.

Deuxièmement, notre approche est basée sur la sémantique en accord avec les travaux de Smith sur 3-LISP [16] pour les langages fonctionnels ainsi que 3-KRS [10] et AGORA [11] pour les langages à prototypes. Les autres approches de la réflexion concernant les langages à objets (incluant OBJVLISP [3], SMALLTALK [2] [14], CLASSTALK [2], CLOS [9]) ne sont pas basées sur la sémantique car les interprètes des niveaux supérieurs ne prennent pas en entrée le code des interprètes des niveaux inférieurs. Les différents niveaux sont plutôt représentés par une structure de pointeurs appropriée. Cette technique permet des implémentations plus efficaces mais n'a pas de base sémantique. De plus, ces langages réflexifs sont des entités monolithiques complexes alors que notre approche modulaire comprend deux parties simples : un interprète non-réflexif et un opérateur `reify()`.

Troisièmement, notre approche permet au concepteur de langage de précisément sélectionner quels mécanismes du langage sont réflexifs. A l'exception de IGUANA et OPEN-C++, tous les systèmes réflexifs cités ci dessus ne possèdent pas cette caractéristique. Enfin, notez que notre approche partage une notion générale de complétude avec 3-LISP, 3-KRS et AGORA : le modèle de programmation est défini par l'interprète et *tous ses mécanismes* peuvent être rendus réifiables. A l'opposé, les autres systèmes réflexifs ne basent pas la réflexion sur un modèle formellement défini mais ils implémentent un MOP ad hoc et la notion de complétude n'a pas de sens dans ce cas.

7 Conclusion

Dans cet article, nous avons présenté une technique de transformation de programme pour générer des interprètes réflexifs de langages à objet à partir d'interprètes non-réflexifs. Cette technique permet de produire rapidement des MOP sur mesure. Les nouveaux MOP peuvent être développés isolément ou en raffinant des MOP existants comme montré dans la Section 5. Le cadre de définitions de langages réflexifs à objets résultant est le premier à satisfaire les trois propriétés de base mentionnées dans la Section 6. En conséquence, notre approche sépare bien les parties réflexives et non-réflexives d'un système, améliorant ainsi la compréhension des MOP générés. Un prototype nommé METAJ [12] est disponible.

Il existe plusieurs pistes possibles de travaux futurs concernant l'exploration des capacités réflexives pour la métaprogrammation (construction d'une taxinomie des MOP) , l'efficacité (utilisation de techniques d'évaluation partielle) et la formalisation de notre approche (transformation d'une sémantique en sémantique réflexive).

Remerciements. Ce travail a bénéficié de remarques de Kris de Volder, Shigeru Chiba, Noury Bouraqadi, Matthias Braux et Thomas Ledoux.

Références

[1] J. McAffer. Meta-Level Programming with CODA. Prooceedings of ECOOP 1995.

[2] J.P. Briot, P. Cointe. Programming with Explicit Metaclasses in SMALLTALK. Proceedings of ACM Object Oriented Programming, Systems, Languages, Applications, pp 419-431, 1989.

[3] P. Cointe. Metaclasses are First Class Objects : the OBJVLISP Model. Proceedings of ACM Object Oriented Programming, Systems, Languages, Applications, pp 156-167, 1987.

[4] P. Cointe. CLOS and SMALLTALK : a comparison. In "Object-Oriented Programming : The CLOS perspectives ?", A. Päpcke (ed.), MIT Press, 1993.

[5] R. Douence, M. Südholt. Generic reification of object-oriented languages. Publication interne no. 00-3-INFO. Ecole des Mines de Nantes. à paraître. 2000.

[6] E. Gamma, R. Helms, R. Johnson, J. Vlissides. Design Patterns. Addison-Wesley, 1995.

[7] B. Gowing, V. Cahill. Meta-Object Protocols for C++ : The IGUANA Approach. Proceedings of Reflection'96. 1996.

[8] JAVA home page. Sun Microsystems, Inc. `http ://java.sun.com`

[9] G. Kiczales, J. des Rivières, D. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[10] P. Maes. Concepts and Experiments in Computational Reflection. Proceedings of ACM Object Oriented Programming, Systems, Languages, Applications, pp 147-155, 1987.

[11] W. De Meuter. AGORA : The Story of the Simplest MOP in the World. In "Prototype-based Programming", J. Noble et al. (ed.), Springer Verlag, 1998.

[12] `MetaJ` home page : `http ://www.emn.fr/sudholt/research/metaj.`

[13] P. Cointe (ed.). Proceedings of Reflection'99, LNCS 1616, Springer Verlag, 1999.

[14] F. Rivard. SMALLTALK : a Reflective Language. Proceedings of Reflection'96, pp 21-38, 1996.

[15] S. Chiba. A Metaobject Protocol for C++. Proceedings of OOPSLA. 1995.

[16] B.C. Smith. Reflection and Semantics in LISP. Proceedings of ACM Symposium on Principles of Programming Languages, pp 23-35, 1984.