Javaに関する技術・応用表現大賞'97 技術部門

OpenJava - *Your Macro is Here ! -*

立堀 道昭 筑波大学 mich@softlab.is.tsukuba.ac.jp
小柳 光生 筑波大学 koyanagi@softlab.is.tsukuba.ac.jp

1.技術的主張の概要

何がしかの機能を効率良く，あるいは簡便に使えるように提供する ためにプログラミング言語を拡張したいことが多くある． 本 OpenJava システムでは， 自己反映計算により，Java 言語のために， 通常のライブラリとともにソースプログラムの変換方法の記述を容易に 加えることができる．
OpenJava では， JDK の Reflection API を拡張する形で OpenC++ の compile-time MOP（Metaobject Protocol）を Java 言語 に適用することにより，既存の introspection に加えて実行効率の 良い intercession を提供する． 一つの MOP で introspection と intercession を統合して提供するため， ライブラリの作成者が runtime と compile-time を意識する必要がなくなり， 自己反映計算による言語の拡張を伴うライブラリの記述が容易になる．

---

2. 背景

JDK1.1 [1] では reflection がサポートされる． Reflection のための計算をメタ計算というが， メタ計算の対象として扱うオブジェクトを Metaobject と呼び， メタ計算のための枠組を MOP(Metaobject Protocol) と呼ぶ． JDK の MOP では，クラスをメタオブジェクトとして扱い， そのために， java.lang.Class クラスなどが用意されている．
Reflection はおおまかに， introspection と intercession と呼ばれる 二つの機能に分けられる． しかし， Java Reflection API [2] で提供されているのは introspection のみである．

2.1. Java Reflection API による恩恵

Java Reflection API が提供するのは introspection である． introspection は実行時にクラスのメンバ等を調べたり 利用したりする機能で， 比較的容易に効率の良い実装が可能である．
Introspection の機能により， JavaBeans のようなサービス等のクラスの公開されたメンバに関する情報を 実行時に必要とするものや， Object Serialization のようなサービス等のクラスの実装情報を 必要とするものを容易に作ることが可能になった．

2.2. Java Reflection API に欠けるもの

Java Reflection API では intercession は提供していない． intercession は実行時にメソッド呼

び出しの動作を変更したり，クラスに新たなメソッドを追加したりする機能である．Intercession があれば，言語の拡張を伴うライブラリの作成 ができるようになり，ユーザは，使い易く実行効率の良いライブラリを 使えるようになる．例えば，RMI(Remote Method Invocation) [3] やHORB [4] は，クラスライブラリとともに専用のトランスレータや stub generator を用意することで分散プログラミングをサポートしている．もし intercession が利用できれば，専用のシステムを用意しなくとも HORB や RMI のような機能を提供できるようになる．
ところが，intercession の効率の良い実装は困難である．これは，通常，MOP が提供する abstraction はインタプリタであり，インタプリタ風に記述されたメタコードを効率良く実行することが 困難であることによる．部分計算を用いることにより intercession の実行効率を 改善している言語に ABCL/R3 [5] がある．しかし，一般に，Java のような言語では実行効率の良いコードを生成する部分計算の実現は困難である [6] .

---

3. 構成・新規性・有用性

OpenJava では効率の良い intercession を JDK の自然な拡張として 導入するために，動的な型に基づいた compile-time MOP を開発した．

3.1. 動的な型に基づく compile-time MOP

実行効率のよいコードの生成を実現するために，OpenC++ [7] をまねて，MOP の intercession の abstraction としてソースコード変換を採用した．ただし，JDK の自然な拡張として intercession を導入するために，新たに，動的な型に基づいた compile-time MOP を開発した．

3.1.1. Compile-time MOP

OpenJava の compile-time MOP では，OpenC++ と同様，メタクラス [8] にそのインスタンスとなるクラスメタオブジェクトに関する ソースコードを変換するためのメソッドを記述することにより 言語の振舞いを拡張する．
OpenC++ 風の Compile-time MOP を用いたメタプログラミングの方法を 簡単な例を挙げて説明する．ここでは，オブジェクトのメソッドが呼ばれるたびに メッセージを出力するように Java を拡張してみる．

```
metaobject MyPanel instanceof Verbose;
public class MyPanel extends Panel {
  public void add( Label label ) {
    // do something
  }
}
```

上で定義された **`MyPanel`** クラスの振舞いは メタクラスである **`Verbose`** クラスにより拡張される．具体的には，**`Verbose`** クラスは **`MyPanel`** クラスに関係するソースコードの 変換を定義する．ここでは，**`Verbose`** クラスは 以下のように定義する．

```
public class Verbose extends openjava.Class
{
```

```
  public Ptree translateMethodCallStatementUsed(
      Ptree obj, Ptree member, Ptree args ){
    return Ptree.makeStatements(
        "System.out.println( \""
        + member.toString()
        + " is called.\");"
        + super.translateMethodCallStatementUsed(
              obj, member, args ).toString()
      );
  }
}
```

メタクラスで定義されたメソッド **translateMethodCallStatementUsed()** により，
MyPanel クラスを使ったソースコード：

```
    MyPanel panel = new MyPanel();
    ....
    panel.add( label );
    ....
```

は，コンパイル時に以下のように変換される．

```
    MyPanel panel = new MyPanel();
    ....
    System.out.println( "add is called." );
    panel.add( label );
    ....
```

3.1.2. java.lang.Class の自然な拡張のための問題点

JDKでは introspection を利用できるようにするために API として **java.lang.Class** クラスに各種の メソッドが用意されている． OpenJava ではこの **Class** クラスを拡張して intercession を利用できるようにするためのメソッドを用意し， 新たなクラス **openjava.Class** として 提供する．
ただし， **Class** クラスに， compile-time MOP の intercession のためのメソッドを そのまま追加するわけにはいかない． それは introspection が動的な型に基づくのに対し，
OpenC++ の compile-time MOP の intercession が 静的な型に基づくことによる． 例えば，
変換されるソースコードが以下の様であるとき，

```
public static void test( Panel panel ) {
  ....
  panel.add( label );
  ....
}
```

変数 **panel** の指す実体のオブジェクトの型は **Panel** クラスだけとは限らず， そのサブクラスの **MyPanel** クラス である可能性もある． もし **MyPanel** クラスであったとすると， 動的な型に基づけば実体のオブジェクトの型である **MyPanel** クラスのメタクラスが適用されるが， 静的な型に基づけば変数 **panel** の型である **Panel** クラスのメタクラスが 適用されることになる．
一つのオブジェクトに対し二つのメタクラスが適用されるのは 好ましくない． それは，
モデルが複雑になるためである． まず，プログラマは introspection と intercession とで メ

タクラスを切替えて使わねばならなくなる． また，intercession のためのソースコード変換を行なうメソッド中で introspection のメソッドを使えなくなる． もし使えるようにしてしまうと，コンパイル時に intercession のメタ計算を 行なえなくなり， compile-time MOP による効率の良い実装が阻害される． やはり，プログラマが容易に利用できように， 統一して提供する必要がある．

3.1.3. 解決法

上記の問題を解決するために， OpenJava では動的な型に基づく compile-time MOP を開発した． intercession も introspection と同様， 動的な型に基づいてメタクラスが選択されるので， JDK の自然な拡張として提供することができるようになる． この compile-time MOP の実装方法を変える． 3.1.3 節の例のメタクラスとクラスを使って簡単に説明すると， 3.1.2 節の例のベースレベルのソースコードは そのままにしておき， 代わりに **`MyPanel`** のクラス定義のソースコードを 以下のように変換する．

```
public class MyPanel extends Panel {
  public void original_add( Label label ) {
    // do something
  }
  public void add( Label label ) {
    System.out.println( "add is called." );
    original_add();
  }
}
```

変換の要点は， 静的に判別できるオブジェクト以外の変換において， メタクラスである **`Verbose`** クラスには オブジェクト **`panel`** を呼ぶ側のソースコードの変換を 許さないようにしていることである． 代わりに，呼ばれた側のソースコードを変換して同等の拡張を行なっている． なぜならば，オブジェクト **`panel`** を呼ぶ側では 静的な型と動的な型が一致しないかもしれないが， 呼ばれた側では一致するからである． オブジェクト **`panel`** を呼ばれた側では 静的な型と動的な型が必ず一致するので， コンパイル時にも動的な型に基づいたメタクラスを選択し， 変換を実行させることができる． これならば，もし intercession のためのメソッド中で introspection のメソッドが呼び出されていたとしても，その結果をコンパイル時に計算してしまうことができる．

3.2. システム構成

3.2.1. システムの動作

図1のように，OpenJava のソースコードはシステムにより一旦 Java の ソースコードに変換し，さらに， Java 言語のコンパイラによって バイトコードが生成している． 実際，バイトコードの生成は javac により行なっており， 将来 JDK のバージョンが変わってもシステムを対応させることが容易である． また，生成されるコードは通常のバイトコードであるため， 通常の Java Virtual Machine 上で実行できる．

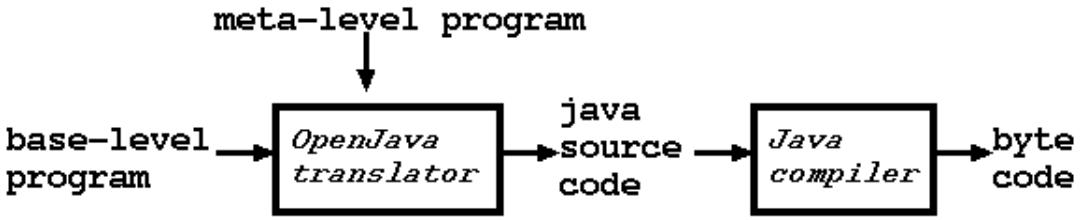

図**1:** OpenJava のシステムの動作

3.2.2. *OpenJava API*

OpenC++ の MOP では，全ての構文要素を１つのクラスで取り扱い， 構文木をリストとして扱う． リストは自由に操作でき， また，構文要素リストの何番目と指定することで， 含まれている構文要素を取り出す．
OpenJava では，ソースコードの各構文要素を取り扱うための クラス群を用意した． クラスによって構文要素を型付けすることにより， 構文エラーを含むソースコードを変換時に生成することを防ぐ． また，構文要素に適応したメソッドを用意するので， プログラマは正しい構文の範囲内で自由にソースコードの変換ができるし， 意味のある名前のついた各メソッドを呼ぶことで操作するために， 誤った操作を起こしにくい．

---

4. システムの実行例

やや複雑な変換を行なう例として， Java を拡張して C 言語の **enum** のような機能を与える． メタクラスを A.1 のように定義する． これにより，クラス定義のソースコード A.2 は ソースコード A.3 に変換されている．
実行は以下のように行なう．

```
owl% ls -F
EnumMacroProcessor.oj  ProtocolID.oj          openjava@
owl% java openjava.main.Main EnumMacroProcessor.oj
Reading from EnumMacroProcessor.oj
metaclass : null
registered userKeywords[] = null
javac EnumMacroProcessor.java
owl% ls -F
EnumMacroProcessor.class  EnumMacroProcessor.oj     openjava@
EnumMacroProcessor.java   ProtocolID.oj
owl% java openjava.main.Main ProtocolID.oj
Reading from ProtocolID.oj
metaclass : EnumMacroProcessor
registered userKeywords[] = { enum }
javac ProtocolID.java
owl% ls -F
EnumMacroProcessor.class  ProtocolID.class          openjava@
EnumMacroProcessor.java   ProtocolID.java
EnumMacroProcessor.oj     ProtocolID.oj
```

---

5. まとめ

OpenJava では， OpenC++ の compile-time MOP と同様， ソースコードの変換を MOP の abstaction とすることで， Java で効率良く intercession ができるようにした．
OpenC++ の compile-time MOP は静的な型に基づくものであったが， 新たに，動的な型に基づく compile-time MOP を開発した． これにより， Java Reflection API で実現されている動的な型に基づく introspection の自然な拡張として intercession を提供できるようになった． 一つの MOP として統一して提供するので， プログラマは introspection と intercession の区別を意識しなくて済む． しかも， Intercession の動作のほとんど全てをコンパイル時に行ない， 実行時にはメソッド呼び出しのオーバーヘッドがあるだけなので， 生成コードは，従来の compile-time MOP の良い実行効率を保つ．
また， ソースコードの各構文要素を構文的な意味によってクラス で取り扱うようにした． クラスの型付けにより，ライブラリの作成者は構文の間違いを コンパイル時に知らされるため，未然にミスを防ぐことができる．
プログラマは OpenJava を用いることにより， reflection による 言語の拡張を伴うライブラリを容易に作成することが可能になった．

参考文献

[1] Doug Kramer : "JDK 1.1.3 Documentation", Sun Microsystems, Inc., (1997).

[2] JavaSoft : "Java(TM) Core Reflection API and Specification", Sun Microsystems, Inc., (1997).

[3] JavaSoft : "Java(TM) Remote Method Invocation Specification", Sun Microsystems, Inc., (1997).

[4] Satoshi Hirano : "HORB : Distributed Execution of Java Programs", *in WWCA97*, (1997).

[5] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai and Akinori Yonezawa : "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Language Using Partial Evaluation", *in OOPSLA'95*, (1995).

[6] Charles Consel and Olivier Danvy : "Tutorial Notes on Partial Evaluation", *in Proc. of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, (1993).

[7] Shigeru Chiba : "A Metaobject Protocol for C++", *in Proc. of the ACM Conference on OOPSLA'95*, SIGPLAN Noteices Vol.30, No.10, pp.285-299, (1995).

[8] Pierre Cointe : "Metaclasses are First Class : the ObjVlisp Model", *in Proc. of the ACM OOPSLA'87*, SIGPLAN Noteices Vol 22, No.12, pp.156-165, (1987).

付録

C 言語の **enum** のような機能を与える例．

A.1. メタクラスの定義

```
// EnumMacroProcessor.oj
```

```
public class EnumMacroProcessor extends openjava.Class
{
  private static final String ENUM = "enum";
  private static final String APPLY = "apply";
  private static final String[] userkeyword = { ENUM };

  /**
   * Overrides to define a new keyword
   */
  public String[] getUserKeywords() {
    return userkeyword;
  }

  /**
   * Overrides to detect enum declarations.
   */
  public FieldDeclarationList
  translateFieldDeclarationList(
      Environment            env,
      FieldDeclarationList   fieldlist )
  throws PtreeException {
    if(fieldlist == null)  return null;

    for( int i = 0, len = fieldlist.getLength(); i < len; i++ ){
      FieldDeclaration field = fieldlist.nth( i );
      if(field.getModifierList().includes( ENUM )){
        /* cnames will be the List of constant variable names */
        List cnames = getNames( (FieldVariableDeclaration) field );
        /* adds the new fields of constant variables */
        addConstantFields( fields, cnames );
        /* deletes [enum apply int = {...}; */
        fields.delete( i );
        break;
      }
    }

    /* continue other translations */
    return super.translateFieldDeclarationList( env, fields );
  }

  /**
   * Returns the constant variable's names.
   */
  private List getNames(
      FieldVariableDeclaration fvdecl )
  throws PtreeException {
    /* vd must be [enum int apply = {...}] */
    VariableDeclarator vd
        = (VariableDeclarator) fvdecl.getVariableDeclaratorList().getFirst();
    /* vd.getLeft() must be [apply] */
    if(! vd.getLeft().equals( "apply" )) {
      mopErrorMessage( "enum error : " + leftname + " is to be \"apply\"." );
    }
    /* vd.getRight() must be [{...}] (not be Expression) */
    if(vd.getRight() instanceof Expression)
      mopErrorMessage( "enum error : bad initializer format." );
    }
```

```
    /* returns [...] */
    return vd.getRight().getArrayInitializerList();
  }

  /**
   * Adds new constant variable's declarations to the field-list.
   */
  private void addConstantFields(
      FieldDeclarationList fields,
      List names )
  throws PtreeException {
    for( int i = 0, len = names.getLength(); i < l; i++ ){
      fields.add( FieldVariableDeclaration.make(
                  "public static final int " +
                  names.nth( i ).toString() +
                  " = " + i + ";" )
          );
    }
  }

}
```

A.2. 変換前のクラス定義のソースコードの一部

```
// ProtocolID.java
metaobject ProtocolID instanceof EnumMacroProcessor;
public class ProtocolID
{
  enum int apply = {
    SUCCESS, BAD_REQUEST, AUTH_REQUIRED, FORBIDDEN,
    NOT_FOUND, SERVER_ERROR, NOT_IMPLEMENTED
  }
}
```

A.3. 変換により生成されたクラス定義のソースコードの一部

```
public class ProtocolID
{
  public static final int SUCCESS = 0;
  public static final int BAD_REQUEST = 1;
  public static final int AUTH_REQUIRED = 2;
  public static final int FORBIDDEN = 3;
  public static final int NOT_FOUND = 4;
  public static final int SERVER_ERROR = 5;
  public static final int NOT_IMPLEMENTED = 6;
}
```