Java 言語のための新たな自己反映機構の提案

OpenJava : Yet another reflection support for Java

立堀 道昭[†] 千葉 滋[††] 中田 育男[†††]
Michiaki TATSUBORI Shigeru CHIBA Ikuo NAKATA

†筑波大学 工学研究科
Doctoral Program in Engineering, University of Tsukuba
††筑波大学 電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
†††図書館情報大学 図書館情報学部
University of Library and Information Science

概 要

本論文で提案する OpenJava では，JDK の Reflection API を拡張する形で，既存の introspection に加えて intercession を提供する．これにより使い易く，効率の良いライブラリが書けるようになる．従来，効率の良い intercession の実装は困難であったが，OpenC++ で提案された compile-time MOP を採用し，実行効率の良い intercession を可能にする．ただし，JDK を自然に拡張するために，OpenJava では新たに動的な型に基づく compile-time MOP を開発した．これにより単一の MOP が introspection と intercession を提供できるようになり，プログラマは両者の区別を意識せずに自己反映計算を伴うライブラリを容易に記述できるようになる．

1 はじめに

JDK(Java Devloper's Kit)1.1 [1] では reflection の機能の一部である introspection が提供されており，これにより，プログラマは JavaBeans のようなサービスを容易に利用または提供できるようになった．一方で，もう一つの reflection の機能として intercession がある．intercession は，言語自体の拡張を備えたライブラリを作成することを可能にするもので，これにより，ユーザは，使い易く，実行効率の良いライブラリを利用できるようになる．しかし，実行効率の良い intercession の実装は困難であり，JDK でも提供していない．

OpenJava では，JDK で introspection を提供しているクラスである Class クラスを拡張し，introspection とともに intercession も提供する．実行効率の良い intercession を可能にするために，OpenC++ [5]で提案された compile-time MOP(Metaobject Protocol) を採用する．Compile-time MOP は，言語の振舞いの変更などのメタ計算を，ソースコードの変換として記述させる．コンパイル時にそのメタ計算を実行することで実行効率の良い intercession を提供することができる．

JDK の Reflection API がオブジェクトの動的な型に基づくのに対して，OpenC++ の compile-time MOP は静的な型に基づく．本論文では，動的な型に基づくようにした新たな compile-time MOP を提案する．これにより，従来の Java Reflection API を自然に拡張して，intercession を提供することが可能になり，ライブラリの作成者は容易に reflection 機構を利用できる．

以下では，まず，第2章で，JDK1.1 の introspection の概要を述べ，また，intercession を導入することの意義を述べる．これにより JDK に対する本研究の位置付けを示しておく．第3章で，JDK

に OpenC++ の compile-time MOP を導入し，実行効率のよい intercession を提供する方法について述べる．その際に JDK の introspection の自然な拡張として intercession を提供するための問題点を挙げる．そして，問題の解決法として，新たに，動的な型に基づく compile-time MOP を提案し，その実現方法を述べる．第 4章で他の研究との比較を論じ，最後に第 5章でまとめと今後の課題を示す．

2 JDK1.1 の reflection

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．このうち，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，コンポーネントソフトウェアである JavaBeans のようなサービスや Object Inspector 等，公開されたフィールド，メソッド，およびコンストラクタに関する情報を実行時に必要とするものを容易に作ることが可能になった．また，Object Serialization のようなサービスやデバッガやインタプリタ等，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] は分散プログラミングをサポートするが，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 を用意することでその機能を実現している．もし intercession の機能が利用できれば，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．ABCL 言語上で部分計算 [7] [8]を用いて intercession の実行効率を改善しているものに ABCL/R3 [9] 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り [8]，Java のための効率の良い部分計算法はまだ構築されていない．

3 JDK1.1 への intercession の導入

OpenJava では効率の良い intercession を可能にするために compile-time MOP を採用する．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 を提案する．

3.1 OpenC++ の compile-time MOP の採用

実行効率のよいコードの生成を実現するために，OpenC++ をまねて，MOP の intercession の abstraction としてソースコード変換を採用する．

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 [6]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 translateMethodCallStatement(
      Ptree obj, Ptree member, Ptree args ){
    return Ptree.makeStatements(
        "System.out.println( \""
        + member.toString()
```

```
            + " is called.\");"
            + super.translateMethodCallStatement(
                  obj, member, args ).toString()
          );
      }
    }
```

メタクラスで定義されたメソッド translateMethodCallStatement() 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 );
....
```

このように変換されたソースコードはさらに図 1のように，Java 言語のコンパイラによりバイトコードにコンパイルされるため，通常の Java Virtual Machine 上で実行できる．

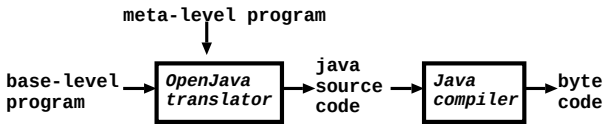

図 1　OpenJava のシステムの動作

3.2　java.lang.Class の自然な拡張とそのための問題点

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，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，3.1節の例で，変換されるベースレベルの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．その理由に，まず，モデルが複雑になる点が挙げられる．この場合，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．やはり，プログラマが容易に利用できように，統一して提供すべきである．さらに，効率の良い実装が阻害される点が挙げられる．intercession のためのソースコード変換を行なうメソッド中で introspection のメソッド呼び出しの結果が使われた場合，コンパイル時に intercession のメタ計算を行なえなくなってしまう．

3.3　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いた compile-time MOP を提案する．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．

そのために，compile-time MOP の実装方法を変える．3.1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，3.2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，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．

上で示した簡単な方法では呼び出し側のソースコードに依存した変換はできない．それは，呼び出される場所により得られる情報が変わるためである．しかし，呼び出しごとに呼ばれた側のコードを複製して別々に変換することで解決できる．ただし，呼び出し側をそれに合わせて多少変換する必要はある．例えば，呼び出し側にはメタクラスの影響を受けるクラスのオブジェクトとそうでないオブジェクトとを分ける分岐を設け，クラスには呼び出される場所ごとの複数のメソッドを用意すればよい．この際，オーバヘッドはオブジェクトを調べることによるものだけの小さいもので済ませられる．

4 関連研究

Cognac [10]では，Java 言語のためのものではないが，runtime MOP と compile-time MOP の統合をはかっている．OpenJava の compile-time MOP の abstraction がソースコード変換であるのに対し，Cognac の compile-time MOP は abstraction はコンパイラの中身（実行コードの生成など）である．Java 言語の実行コードは VM(Virtual Machine) 上のスタックマシンであり，その特性から，実行コードの最適化を目的とする場合でもソースコード変換のみで十分といえる．

5 まとめと今後の課題

本研究の OpenJava では，OpenC++ 風の compile-time MOP を採用し，Java で効率良く intercession が利用できるようにした．

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 を提案した．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，OpenJava を使うプログラマは introspection と intercession の区別を意識する必要がなくなり，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．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，従来の compile-time MOP の良い実行効率を保ちながら，動的な型に基づくものにできた．

今後の課題としては OpenJava の実装の完了と，その生成コードの実行効率の検証が挙げられる．

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] 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.

[6] Pierre Cointe : “Metaclasses are First Class : the ObjVlisp Model”, in Proc. of the ACM OOPSLA’87, SIGPLAN Noteices Vol 22, No.12, pp.156-162, 1987.

[7] 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.

[8] U.Meyer : “Tecniques for partial evaluation of imperative programs”, in Proc. of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, SIGPLAN Noteices Vol.26, No 9, pp.94-105, 1991.

[9] 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.

[10] Kenichi Murata, R.Nigel Horspool, Eric G.Manning, Yasuhiko Yokote and Mario Tokoro : “Unification of Compile-time and Run-time Metaobject Protocols”, in ECOOP Workshop in Advances in Meta’95, 1995.