ユーザにとって直観的なコンパイル時 MOP
A compile-time MOP intuitive for its user

立堀 道昭
筑波大学 工学研究科
mich@softlab.is.tsukuba.ac.jp

千葉 滋
筑波大学 電子・情報工学系
chiba@is.tsukuba.ac.jp

概要

我々は，自己反映計算に基づく拡張可能言語を，特に Java を拡張した OpenJava として実装することを目指しており，その設計について研究している．本稿では，まず，従来のコンパイル時 MOP は構文木をメタオブジェクトとして扱わせるために，メタレベルプログラムの記述が難しく，また，あまり局所的でない処理をする場合に向いていないという問題点を指摘する．そして，その問題を回避する，ユーザにとって直観的な言語の計算モデルに基づいたコンパイル時 MOP の設計について述べる．本 MOP では，プログラマが馴染み易いオブジェクト機構の操作をすることにより，間接的にソースコードの変換がなされる．したがって， OpenJava のプログラマは，望みの言語拡張を従来よりも簡潔に記述することができる．

1 はじめに

分散プログラミングや並列プログラミング等を実現するために特別な言語機構を用意すると便利なことが多い [6]．しかしながら，これらの言語機構は特定の分野のアプリケーションには適しているが，全ての分野で役に立つわけではない．様々なアプリケーションの開発に対応するためには，たくさんの言語機構を始めから盛りこんだ言語よりも，必要に応じて新しい言語機構を実装していけるような拡張可能言語の方が望ましい．

自己反映計算 [2] は，そのような拡張可能言語を設計するための方法論としてよく知られている．当初，自己反映計算は実行効率が悪く実用には向かないと思われていたが，様々な技術の登場によって最近ではこの問題は克服されつつある．本稿でとりあげるコンパイル時 MOP はそのような技術のひとつである．この技術は，自己反映計算をコンパイル時に行なうので，実行時の効率が非常によいことが特徴である．しかしながら，その一方で，普通の自己反映言語に比べてメタレベルプログラムの記述が難しいとされてきた．

従来のコンパイル時 MOP を観察すると，構文木の部分部分にまたがった，あまり局所的でない処理をしなければならない場合に，メタレベルプログラムの記述が難しくなることがわかる．これは構文木にもとづいた従来のコンパイル時 MOP に本質的な欠点である．そこで我々の Java [4, 5] に基づく自己反映言語 OpenJava [3] では，この欠点を回避するために，言語の計算モデルに基づいた新しい種類のコンパイル時 MOP を提供する． OpenJava のプログラマは，構文木に由来するわずらわしい問題から解放され，望みの言語拡張を従来よりも簡潔に記述することができる．

以下，本稿では，まず従来のコンパイル時 MOP の構造を概観し，その問題点を考察する．次に，我々はその問題を回避するために，言語の計算モデルに基づいた MOP を拡張可能な Java 言語である OpenJava のために設計したことを述べる． OpenJava では Java 言語の拡張を比較的簡潔に記述できることをを例を示しながら説明し，またこの MOP の実装方法の概要についても解説する．

2 従来のコンパイル時 MOP

通常の実行時 MOP を備えた言語処理系と異なり，コンパイル時 MOP を備えた言語処理系は，言語仕様の変更に関わる自己反映計算を，実行時ではなくコンパイル時に行なう．これによって，実行効率のよい処理系を作成するのが非常に困難であるという，従来，自己反映計算の欠点とされてきた問題を回避することができる．しかしながら一方で，今まで提案されてきたコンパイル時 MOP の設計では，メタレベルプログラムの

記述が，実行時 MOP と比べあまり直観的でなく，使いよいシステムとはいえなかった．本節では典型的なコンパイル時 MOP をいくつか紹介し，なぜメタレベルプログラムが書きづらいのかを考える．

一般にコンパイル時 MOP では，言語仕様を望むように変更するために，ソースコードの変換方法をメタレベルプログラムとして定義しなければならない．例えば，普通のメソッド呼び出しが，異なる計算機上のオブジェクトのメソッドの呼び出しになるように言語仕様を変更する場合，プログラマは元のプログラム：

```
p.move( 3 );
```

が

```
remote_call( p, "move", 3 );
```

となるようなソースコード変換のアルゴリズムを，メタレベルプログラムとして定義する．

ソースコード変換のアルゴリズムの記述方法は様々なものがあるが，現在提案されているコンパイル時 MOP は，皆，変換アルゴリズムを構文木の変形の形で表現する．例えばもっとも単純なコンパイル時 MOP は次のような構造をとるだろう．

- まずメタオブジェクトとは，構文木の各ノードであり，構文要素 (宣言，式， if 文など) に応じて異なるクラスのインスタンスである．
- 各メタオブジェクトは，対応する部分木を変形し，変形後の部分木を返す， `translate()` というメソッドを持つ．

言語処理系は，コンパイルに先立って構文木の根から葉にむかってこの `translate()` を再帰的に呼び出し，ソースコード全体を変換するので，プログラマはメタオブジェクトを生成するクラスのサブクラスを定義し `translate()` を再定義することで，望みの言語仕様を実現するようにソースコード変換を行なわせることができる．

変換のスコープ制御

上で述べたような単純なコンパイル時 MOP を採用している処理系は実際には存在しない．他の自己反映言語と同様，コンパイル時 MOP を採用した場合も，言語仕様の変更を特定の範囲に制限するためのスコープ制御が重要であるからだ．コンパイル時 MOP の場合，制御されるスコープは，変換を実際に適用するソースコードの範囲となる．

スコープ制御の方法は処理系によってさまざまである．例えば MPC++ [9] では Delegation ， EPP [10] では System Mixin ， A* [11] ではパターンマッチによる方法を採用している．ここでは OpenC++ [1] で使われているメタクラスによる方法を紹介する．

OpenC++ のプログラマは，構文木のノードを表すメタオブジェクトのクラスを変更できない．代わりに各ノードの `translate()` メソッドは，その部分木が表す式（や構文）の型を調べ，その型に応じたクラスメタオブジェクトを呼び出す．例えばその部分木がメソッド呼び出し：

```
p.move( 3 );
```

であるとすると， `translate()` はオブジェクト `p` のクラスメタオブジェクトの `translateMethodCall()` を呼び出す．部分木の変形は実際にはこのメソッドにより実行され， `translate()` は結果を親ノードにそのまま返すだけである．

OpenC++ のプログラマは，望みのソースコード変換を実現するために，クラスメタオブジェクトを生成するクラスのサブクラスを定義し，構文木の各ノードの `translate()` から呼び出される `translateMethodCall()` のようなメソッドを再定義する．こうすることで， OpenC++ ではある特定のクラスのインスタンスについてだけソースコード変換を行なうようなことが，簡単に実現できるようにしている．

従来のコンパイル時 MOP の問題点

いくつかのコンパイル時 MOP が提案され，実際に応用研究が始まると，それらの MOP の有用性と同時に，いくつかの典型的なソースコード変換を行なわせるのが容易ではないという問題点も明らかになってきた．例えば，あるクラスの定義の中に指定された名前のメソッドがない場合にだけ，その名前のメソッドを自動的に定義してクラスに追加する，というような変換を行なわせるのは難しい．

これは MOP の設計上，構文木の各ノードの `translate()` メソッドが呼び出される順序は，ポストオーダでほぼ固定されており，離れたところにある部分木の情報をもとに，ある別の部分木の変形を行なう，という処理を記述しづらいためである．例でいうと，クラスにメソッドを追加するのは，クラス定義に対応する部

分木の根ノードの `translate()` である．しかし指定された名前のメソッドがあるかどうかを調べるには，クラス定義の中に含まれる各メソッド定義を参照しなければならず，`translate()` 中で部分木の内部を探索する処理を記述しなければならない．あるいは各メソッド定義に対応する部分木の `translate()` を変更して，適当な場所に定義されたメソッドの名前を保存するようにし，クラス定義の `translate()` がそれを読み出せるようにしなければならない．

このように，従来のコンパイル時 MOP は，Lisp マクロで処理できるような例やその発展形については，非常にうまく取り扱えるが，そうでない上のような例はうまく取り扱えない．このことは，通常の実行時 MOP に対するコンパイル時 MOP の使いにくさとして，指摘されてきた．

3 OpenJava MOP

2節で述べた問題点は，コンパイル時の自己反映計算において，構文木というモデルが常に適しているわけではないことを示している．例えばクラス定義の構文は複雑であるために，大きくクラスにまたがるような変換を表現するのに，このモデルは向いていない．実際，2節の例で挙げた，あるメソッドが定義されていない場合，そのメソッドを自動的にクラスに追加する変換をうまく表現できない．しかし，このような変換は，オブジェクト機構からすると単純な操作であり，容易に表現できるべきである．

構文木のノードをメタオブジェクトにするのは，コンパイル時 MOP だけであり，他の MOP では，クラスやメソッドなど first-class でない言語要素をメタオブジェクトにしている．こうすることによって，メタレベルでは，それらを first-class な言語要素として扱えるようになり，それを使って言語仕様を拡張することができる．このようなメタオブジェクトの選び方をとる実行時 MOP では，言語仕様の拡張を直観的に表現できると指摘されている [8]．

そこで従来のコンパイル時 MOP の欠点を克服するために，OpenJava でも，クラスやメソッドなど，本来の計算モデルでは first-class でない言語要素をメタオブジェクトにする．構文木のノードをメタオブジェクトにしないので，CLOS MOP などの実行時 MOP と同様に，直観的に言語仕様を拡張できるようになる．ただし，OpenJava はコンパイル時 MOP なので，メタオブジェクトを操作できるのはコンパイル時だけである．この制限によって，実行時 MOP の実行効率の悪さを回避している．このような考えに基づいて，OpenJava ではクラス，メソッド，フィールドなどをメタオブジェクトにしている．

それぞれのメタオブジェクトは関連する情報をフィールドとして持っている．例えばクラスメタオブジェクトの場合，次のような情報を持つ：

1. 説明のための特殊なコメント[1]
2. クラスのパッケージと名前
3. クラス名のスコープやクラスの種類などの属性
4. 継承している親クラスや実装しているインターフェース
5. フィールドやメソッドなどのメンバとそのイニシャライザ

これらの情報は，クラスメタオブジェクトが提供するメソッドを通じて，自由に読み出し，変更が可能である．

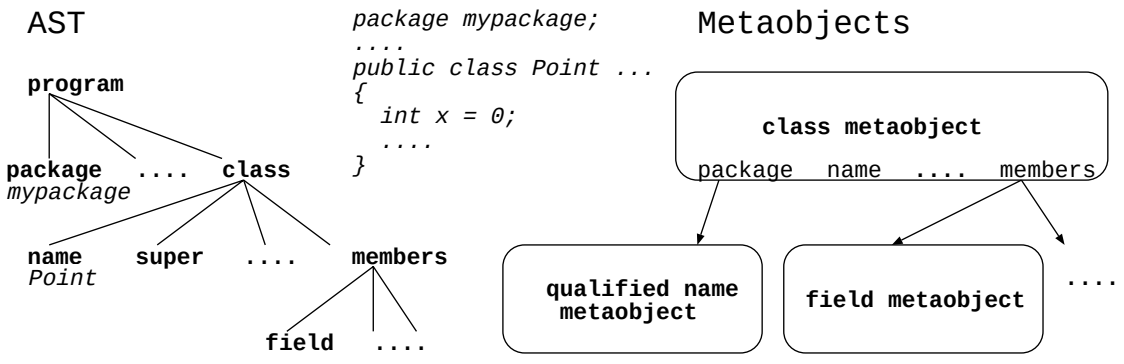

図 1: AST and Metaobjects

[1] オブジェクトのふるまいには関係ないが Java ではこれに基づいてドキュメントが生成され，重要な要素であるため入れてある

このように MOP を設計すると，構文木を使った MOP では異なるメタオブジェクトに別れてしまう情報も，意味的に関連があれば，ひとつのクラスメタオブジェクトの中にまとめられる．このため，2節で述べた問題点をうまく回避することができる．例えば，Java のプログラムでは，ファイルの先頭の package 宣言はクラスが記述されている部分と構文上，やや離れていることがありうるが，これらは意味的には強い関連がある．構文木を使った MOP では，package 宣言とクラス宣言は違うメタオブジェクトになってしまうが，OpenJava では package 宣言はクラスメタオブジェクトのフィールドのひとつである (図 1).

OpenJava のメタプログラムの例を示す．ある名前のメソッドまたはフィールドがクラスに含まれているかどうかを調べる関数は次のように記述できる．クラスメタオブジェクトが持っている情報にアクセスする基本的なメソッドの組み合わせだけで，簡潔に記述できている点に注意されたい．[2]

```
static boolean doesTheNameExist( Class clazz, String name ){
  Iterator members = clazz.getMembers();
  while(members.hasMoreElements()){
    Object member = members.nextElement();
    if(member instanceof Method){
      Method method = (Method) member;
      if(method.getName().equals( name ))  return true;
    }else if(member instanceof Field){
      Field field = (Field) member;
      if(field.getName().equals( name ))  return true;
    }
  }
  return false;
}
```

ここで，クラスメタオブジェクトのメソッド getMembers() ではインターフェース Member を実装しているクラスのインスタンスであるメンバメタオブジェクトを取得することができる．メソッドメタオブジェクトは Method クラスのインスタンスであるので，あるメンバメタオブジェクトがメソッドメタオブジェクトやフィールドメタオブジェクトであるかどうかは Java の instanceof 演算子を用いて調べればよい．ここでは，オブジェクト member がクラス Method のインスタンスであるか調べて，その場合にのみ，クラス Method のメソッド getName() 通じてメソッドメタオブジェクトからメソッドの名前を取得して調べている．

この関数を用いて，クラス定義の中に指定された名前のメソッドがない場合だけその名前のメソッドを自動的に定義してクラスに追加する例は，次のように表現される．

```
if(! doesTheNameExist( class_metaobject, "move" )){
  Method m = Method.make( "void move() { ... }" );
  class_metaobject.addMember( m );
}
```

このコードは "move" という名前のメンバがない場合にメソッド move() を追加する．まず，文字列からメソッドメタオブジェクトを生成し，生成されたメソッドメタオブジェクトはメソッド addMember() を通じてクラスメタオブジェクトに追加され，最終的にはソースコード変換に反映される．ここでは，もはや構文木の変換という抽象モデルは現れていない．

本 MOP の利点

本 MOP では，メタオブジェクトの抽象モデルはクラスやメソッドなどベースレベルのプログラムでも普通に用いられる概念であり，メタレベルのプログラマが親しみ易い．従来の構文木がメタオブジェクトである MOP では文法構造に精通していなければ正しくメタオブジェクトを扱うことができなかった．例えば，従来の MOP では，クラス定義の構文のどこにベースクラスの名前が現れ得るのか知らなければ，それを利用することはできなかったが，本 MOP ではプログラマはクラスメタオブジェクトのメソッドを知りさえすればよく，それを用いればあとはシステムが保証してくれる．

[2] 実際にはメソッドメタオブジェクトだけを取り出すメソッド getMethods() などが便利のために用意されており，それらを使えばより簡潔に記述ができるのだが，ここでは重要でない．

また，ある程度高級なプログラミング言語の文法では，同じ意味のことをユーザの都合や好みにより異なる書き方で書けるのが普通である．構文上は異なっていても意味が同じならば，構文上の違いは意味を持たないため，同等に取り扱う必要がある．単純な例では， Java の場合，以下の 2 通りの変数宣言：

```
Object[] a;
Object a[];
```

は，書き方は異なるがまったく同じ意味であり，参照変数 a の型はクラス Object の配列型であるという宣言を意味している．しかし，従来の MOP ではプログラマはこれらの違いに注意を払わなければならず，1つの意味に複数通りの書き方ができる場合はそれぞれの書き方についての処理を記述する必要があった．そのためには，言語の文法に精通していることが要求される．本稿で提案する MOP では，メタオブジェクトがそのような構文上の違いを吸収しているため，プログラマはそれらを一元的に取り扱うことができる．他にも，クラス名におけるパッケージ名は省略可能であること，すなわち，

```
import java.lang.String;
```

という文脈で String と java.lang.String は同じ意味を持つことや，自己参照子 this は省略可能であること，クラスフィールドの初期化方法等，一元的に取り扱うべき場合は非常に多い．

ただし，ここでは，以下の 2 通りの文：

```
for( i = 0; i <= 3; i++ ) ;
i = 3;
```

は同じ意味になる，というような，計算の結果の意味レベルまでは取り扱わない．ここでいう意味とは，言語の仕様によりコンパイルの結果が同じになると保証されるレベルである．

実装方法

メタオブジェクトに対する操作は，他のコンパイル時 MOP と同様，本 MOP でも最終的には構文木に反映される．しかし，他の MOP と異なり本 MOP では，メタオブジェクトの操作に合わせて，構文木を破壊的代入で変形していかなければならない． Lisp によるプログラミングなどで一般によく知られているように，単純な破壊的代入を使った木構造の変形では，意図しない部分木の共有をおこしプログラマが混乱しやすい．例えば：

```
Class point = ...;
Class rect = ...;
Method m = point.getMethod( "f" .. );  // メソッド f を取り出す
rect.addMethod( m );                   // クラス rect にも f と同じメソッドを加える
m.setName( "g" );                      // メソッド f の名前を g にする
```

とした場合，クラス point のメソッドの名前は f から g に変わる．では，クラス rect に追加されたメソッドの名前は f のままであろうか， g に変わるのであろうか．

OpenJava では，この問題を回避するために， addMethod() のような破壊的代入をおこすメソッドは，必ず引数のコピーを作って代入する．したがって上の例では，クラス rect のメソッドは f のままである．

さらに，より本質的な問題として，メタオブジェクトの操作に合わせて，随時，破壊的代入で構文木を変形していくとすると，変形にともなって introspection (内観) の結果も随時変わってしまうという問題がある．多くのメタレベルプログラムは introspection の結果に応じてソースコード変換をするので，そのソースコード変換によってさらに introspection の結果が変化するような MOP はプログラマにとって馴染みにくいことが知られている [7].

そこで本 MOP では，メタオブジェクトを操作しても，即座に構文木を変形しないようなオプションを用意する．この場合 introspection の結果は，常に元のソースコードの内容に即した結果を返す．破壊的代入による構文木の変形は，最終的にソースコードを出力する直前に自動的に実行される．具体的には次のようなアルゴリズムで処理がおこなわれる:

(1) メタレベルのプログラムを実行するが， setName() のような破壊的代入をともなうようなメソッドが呼ばれたときにも，実際に変更を行わない．代わりに，履歴を保持するためのキューに
   - 操作の対象となったオブジェクトへのリンク
   - 操作の内容（メソッドのシグネチャ）

- 代入されるメタオブジェクトのコピー (意図しない共有を避けるため)

をひとまとめにして入れておく.

(2) メタレベルのプログラムが全て終了した後，メタオブジェクトのリンクをたどり，上の履歴にもとづいて，構文木を変形する.

(3) 変形された構文木にそって，ソースプログラムを出力する.

4 まとめと今後の課題

本稿では，まず，いくつかの典型的なソースコード変換が，従来のコンパイル時 MOP では容易でないことを指摘し，それを解決するための MOP の設計と実装について述べた.

従来のコンパイル時 MOP では，メタプログラムで構文木をメタオブジェクトとして取り扱わせる．システムが構文木をたどるような設計では，離れたところにある部分木の情報をもとに，ある別の部分木の変形を行なうような処理を記述するのが難しい．さらに，プログラマは意図する記述と離れた文法などに注意を払う必要がある.

本稿で提案するコンパイル時 MOP では，言語の計算モデルをメタオブジェクトとして取り扱わせるため，プログラマは直観的にソースコードを操作できる．そのような MOP の設計と実装方法について，特に Java 言語を拡張した自己拡張可能言語 OpenJava の設計を例にとって述べた． OpenJava を使うプログラマは，この特徴のため，自己反映計算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る.

今後の課題として特に以下のことが挙げられる.

様々な観点からの意味モデル

目的とする拡張によっては，メタレベルのプログラマが扱いたい対象はソースコードそのものではなく，ソースコードからコンパイル時に得られるよりハイレベルな情報である．例えば，ソースコード中の変数の型やデータ参照の相互依存情報などが考えられる．これらの情報もまた，メタオブジェクトとして容易に扱えるように， MOP の設計を発展させていかなければならない.

複数のメタクラスの合成

プログラムの再利用性を向上させるために，複数の既存のメタクラスを合成し異なる拡張を組み合わせて使えるようにする研究が行なわれてきた [12].

OpenC++ では，メタクラスを注意深くプログラミングすれば，メタクラスの合成は，多重継承の機構を用いて実現できる． OpenJava では Java には多重継承の機構がないという問題点があるが，多重継承が出来るように拡張を行なうメタクラスを用意すれば同等のことはできると思われる．あるいは，多重継承を用いなくとも，メタクラスによる変換を順次適用していく方式を定める方法でも，メタクラスの合成が可能であると考えられる.

しかし， OpenC++ の場合にも共通していえることだが，単純な合成機構だけでは，ユーザが誤ったメタクラスを合成してしまうことを防げない．言語の同じ部分を拡張の対象とするメタクラスの場合，合成すると，正しく動作しなくなってしまう組み合わせがあるからである．これを避けるため，そのような誤った合成を検出し警告する機能をもった MOP を開発する必要がある.

参考文献

[1] Shigeru Chiba : A Metaobject Protocol for C++, In *Proceedings of OOPSLA'95, ACM SIGPLAN Notices Vol.30, No.10, pp.285-299*, 1995.

[2] Brian Cantwell Smith : Reflection and Semantics in a Procedural Language, *M.I.T. Laboratory for Computer Science Report MIT-TR-272*, 1982.

[3] Michiaki Tatsubori : OpenJava WWW page., `http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/`, 1997-1998.

[4] J. Gosling, B. Joy, and G. Steele : *The java language specification*, Addison-Wesley, 1997.

[5] Doug Kramer : *JDK 1.1 Documentation*, Sun Microsystems, Inc., 1997.

[6] Bal, H. E., J. G. Steiner, and A. S. Tanenbaum : Programming Languages for Distributed Computing Systems, *Computing Surbeys, Vol.2, No.3, pp.261-322*, 1989.

[7] Shigeru Chiba, Gregor Kiczales, and John Lamping : Avoiding Confusion in Metacircularity: The Meta-Helix, in *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS), LNCS 1049, Springer, pp.157-172*, 1996.

[8] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa : Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Language Using Partial Evaluation, In *Proceedings of OOPSLA'95, ACM SIGPLAN Notices Vol.30, No.10, pp.300-315*, 1995.

[9] Ishikawa, Y., A. Hori, M.Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, and K. Kubota : Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach -, In *Proceedings of Reflection 96, pp.153-166*, 1996.

[10] Yuuji Ichisugi : EPP and Lods home page., `http://www.etl.go.jp/etl/bunsan/~ichisugi` , 1997.

[11] Ladd, D. A. and J. C. Ramming : A* : A Language for Implementing Language Processors, *IEEE Trans. on Software Engineering, Vol.21, No.11, pp.894-901*, 1995

[12] Philippe Mulet, Jacques Malenfant, and Pierre Cointe : Towards a methodology for explicit composition of metaobjects, In *Proceedings of OOPSLA'95, ACM SIGPLAN Notices Vol.30, No.10, pp.316-330*, 1995.