一般論文

リフレクションの高速化技術

千葉 滋　立堀 道昭　佐藤 芳樹　中川 清志

要旨

リフレクションの研究は、計算モデルが先行したので、当初実装は素朴なものばかりで実用にはならないとされていた。本論文は、素朴な実装では実行時に全て動的におこなっていた処理を、機能に制限を加えつつも、静的におこなえるようにし、実行速度を改善する技術について述べる。我々が開発したこの技術により、C++ 言語や Java 言語のような実行効率が重視される言語でも、リフレクション機能を利用することが可能になった。また本論文は、リフレクションとアスペクト指向プログラミングとの関連を軸に、この分野の研究の今後の展望について、著者らの見解を述べる。

(Reflection had been regarded as an idea that was inapplicable for practical software development. This paper presents techniques that we have developed for better implementation of reflective computing. Our techniques move the bottleneck of reflective computing from runtime to load/compile time so that reflection could be pragmatic in C++/Java programming, in which execution performance is significant. This paper also mentions our perspective on this research area, in particular, with respect to the relation between reflection and aspect-oriented programming.)

Implementation Techniques for Faster Reflective Computing

Shigeru Chiba, Yoshiki Sato, Kiyoshi Nakagawa, 東京工業大学 数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Instiute of Technology.

Michiaki Tatsubori, 日本アイ・ビー・エム東京基礎研究所, IBM Tokyo Research Laboratory.

コンピュータソフトウェア, Vol.21, No.6(2004), pp.5–15.
[論文]2003 年 2 月 3 日受付.

1 はじめに

リフレクション (reflection) あるいは自己反映計算は、プログラム中から、そのプログラム自身をデータとして取り扱い、計算の対象にできるようにする技術である。通常のプログラミング言語では、データとして扱える対象は数値や文字列、ポインタ等であり、型や関数の定義自体はデータとして扱えない。

リフレクション機構を備えたオブジェクト指向言語では、クラスやメソッドの定義もオブジェクトとして表され、一般のデータと同様に扱える。言語によっては、メソッド呼び出し等の演算子の挙動の定義もデータとして扱えるものもある。この機能により、多くの有用な言語拡張が、言語処理系自体を改変せずに、ライブラリとして実装できるようになった。

リフレクションの研究は 1980 年代前半に、Smalltalk や Lisp 言語が元々ad hoc な形でもっていたプログラムをデータとして扱う機構に、意味付けをおこなう研究として始まった [33]。オブジェクト指向言語のリフレクションの研究も初期のころから存在し [24]、1990 年ごろには活発に研究されていた。とくに Common Lisp のオブジェクト指向拡張である CLOS の実装の研究から生まれた CLOS MOP [16] は、CLOS の標準仕様に取り込まれた。リフレクショ

ン機能の最初の実用化であるといえる。

リフレクションは、1990 年ごろには十分に有用な機能として認知されていたが、C++ 言語など、実行効率を重視するプログラミング言語にはなかなか採用されなかった。これは当時の実装の大半が、リフレクションの計算モデルをそのまま素朴に実装したものであり、実用に耐える実行速度を達成できないと一般に考えられていたためである。実際に開発されたリフレクション機能つきの言語処理系は、Lisp 言語をもとにしたものか、独自のオブジェクト指向言語であり、インタプリタが大半であった。

本論文は、オブジェクト指向言語のためのリフレクション機能を実行効率よく実装するために、我々がこれまで開発してきた技術について概観する。この実装技術によって、C++ 言語や Java 言語のような実行効率が重視される言語でもリフレクション機能を利用することが可能になり、数多くの有用な言語拡張を実現するライブラリを作成することができるようになった。我々の実装技術の特徴は、リフレクションの計算モデルに一定の制限を加え、等価だが効率のよい形に変換して実装できるようにした点にある。とくに、リフレクションにかかわる処理をできるかぎりにコンパイル時またはロード時に静的におこなえるようにして、効率よく動作するようにした。処理の一部を実行前に静的におこなって実行効率を改善する手法は最適化コンパイル技術の柱のひとつである。しかしリフレクションにかかわる処理の中から、静的におこなえる処理をコンパイラ等で自動的に抽出するのは非常に困難であった。我々は、プログラミング・インターフェースを工夫することで、この問題を回避した。

以下、2 章では、リフレクション機能を具現化と反映に分けて、それぞれの素朴な実装方法について述べる。このうち具現化機能は効率のよい実装が比較的容易で、Java 言語でも標準仕様に含まれている。3 章と 4 章では、効率のよい実装が自明でなかった反映機能のために、我々が開発した実装技術を述べる。3 章は反映機能のうちの動作リフレクションについて、4 章は構造リフレクションについて述べる。本論文ではさらにまとめとして、5 章でリフレクション機能とアスペクト指向技術との関連にふれながら、今後の研究の展望について著者らの私見を述べる。

2 リフレクションの実装技術

2.1 具現化 (reify)

リフレクションの機能のうち、**具現化** (reify) あるいは**内観** (introspection) と呼ばれる機能は、実行効率のよい実装が容易である。このため C++ 言語では RTTI (RunTime Type Information) [36]、Java 言語では reflection API [15] として、機能を制限したものが既に標準仕様の一部になっている。

オブジェクト指向言語における具現化 (内観) は、クラスやメソッドの定義など本来データとして扱えないものを通常のオブジェクトに投影し、データとして扱えるようにする機能である。例えば以下の Java プログラムは、文字列変数 cname の値を名前としてもつクラスの start メソッドを呼び出す。

```
Class c = Class.forName(cname);
Method m = c.getMethod("start",
                       new Class[0]);
m.invoke(null, null);
```

このような処理は、上のように具現化機能を使わなくては記述できない。仮に次のようにプログラムを書くと、

```
cname.start();
```

これは cname オブジェクトの start メソッドを呼ぶ意味になってしまう。

Class や Method は、Java 仮想機械 (JVM) 内に (JVM の実装言語が扱うデータとして) それぞれ保存されているクラスやメソッドの定義を読み取るためのオブジェクトである。invoke メソッドは native メソッドで、インタプリタ (JVM) がメソッド呼び出しを実行するために使う JVM 内部の (JVM の実装言語で書かれた) ルーチンを呼び出す。

Java 言語の場合、クラスやメソッドの定義は実行中に変わらないので、具現化の実装は、インタプリタ内部のデータを表現するオブジェクトを作るだけであり、実装は比較的容易である。実行効率も悪くない。C++ 言語のような言語処理系がコンパイラである場合 (あるいは実行時コンパイラつきの JVM の場合) の実装法もほぼ同様である。まずクラスやメソッドの

定義を、デバッグ用のシンボル情報として保存するだけでなく、プログラム中の大域変数の中にも定数として保存する。そうすれば、その大域変数を読み取って実行中に Class や Method オブジェクトを作ることができる。また invoke メソッドを実現するためには、対応する機能をもつ実行時ライブラリをコンパイル時に生成するようにすればよい。

具現化機能の応用は広い。例えば Java 言語では、applet など、実行時に利用者が指定したクラスを動的にロードして、そのクラスのメソッドを呼び出すとき [23] に、この機能を使う。また遠隔ホスト上のオブジェクトに対するメソッド呼び出しを実現するための stub 生成器を実装する際にも、この機能は便利である [1]。

2.2 反映 (reflect)

具現化機能によって作られたクラスやメソッドの定義などを表すオブジェクトを操作して、フィールドの値を変更しても、元のクラスやメソッドの定義が連動して変わるわけではない。そのためには、変更を元のクラスやメソッドの定義に明示的に**反映** (reflect) させなければならない。しかし、この反映をおこなう機能は、効率のよい実装が困難と 1990 年代前半まで一般に考えられており、C++ や Java 言語では標準仕様に含まれていない。

反映機能を素朴に実装するなら、言語処理系を実行の最適化をおこなわないインタプリタのようなものとし、処理系の内部データやコードの一部を、実行中のプログラムの中から変更できるように処理系を作ればよい。インタプリタの内部データだけでなく、コードも変更可能にすれば、例えばメソッド呼び出しの挙動をプログラム中から変更することが可能になる。この実装方法は、後に**オープン実装** (open implementation) [20] という設計論に一般化された。

3-Lisp [33] 等、多くの処理系はこのような方式で実装されてきた。3-KRS [24] や ABCL/R [38] では、オブジェクトごとに別々のインタプリタを使えるので、特定のオブジェクト用のインタプリタのコードを変更し、そのオブジェクトの挙動だけを変更することが可能であった。各オブジェクトに対応するインタプリタのことを**メタオブジェクト** (metaobject) と呼ぶ。メタとつく理由は、通常のデータを扱うオブジェクト自体をデータとして扱うオブジェクトだからである。一般に、プログラム中でリフレクションを実行する部分のことを**メタプログラム**と呼ぶ。また一般には、上記のインタプリタの他に、プログラム中の非 first class データを具現化したオブジェクトも、メタオブジェクトと呼ばれる。

素朴な実装方法では、処理系が必然的にインタプリタになるため、反映機能の高速な実行は困難であると考えられていた。とくに 3-Lisp や ABCL/R では、リフレクションの計算モデルを明らかにすることが研究の主眼であったため、インタプリタ自身もその言語で書かれていた。例えば ABCL/R の処理系は、プログラムのインタプリタ実行をおこなうオブジェクトの集合体として説明されており、メタオブジェクトは処理系を構成するそれらのオブジェクトのことであった。この自己再帰的な構造のため、これらの言語の実行速度は極端に悪いと考えられていた。

一方、Smalltalk や CLOS MOP [16] は反映機能を提供するが、それによる実行速度の低下は小さい。反映機能で変更できるプログラムの挙動をいくらか制限しつつ、その制限を活用して、処理系であるインタプリタの実装を高度に最適化 [18] しているためである。Smalltalk の場合、最適化の結果、処理系はむしろコンパイラといえるものになっている。しかし、反映機能を利用するためには、最適化された処理系の構造を意識しなければならず、必ずしも利用が容易でないのが欠点である。

類似の着想は後に ABCL/R の実装にも応用された。処理系の実装に様々な最適化をほどこし、例えば、通常は反映機能をもたないコンパイラを使ってプログラムを高速実行し、反映機能が必要になった段階で、相対的に低速だが反映機能が利用できるインタプリタに切り替えるようにしている。最適化により、反映機能が組み込まれていない処理系 (ABCL/1 [40]) に比べて、ABCL/R の速度低下が、反映機能を使わない限り 10 倍以下に抑えられることが報告されている [26]。

このように、比較的効率のよい反映機構の実装方法

が提案されてはいたが、主な処理系は Lisp 言語上のものであった。このためか、C++ 言語のような、処理系にコンパイラを使う実行速度が重視される言語で、反映機能を効率よく実装する方法は、あまり明らかではなかった。1990 年ごろの多くの研究者は、依然として、反映機能を実用的に使うことは困難と考えていた (文献[35] の 14.2.8.1 節)。

3 動作リフレクション

本論文では、我々が C++ や Java 言語のために開発した効率の良い反映機能の実装技術について述べる。反映機能は大別して動作リフレクションと、構造リフレクションに分けられる。本章では、まず動作リフレクション用に我々が開発した実装技術について述べ、次章で構造リフレクション用の実装技術を述べる。

動作リフレクション (behavioral reflection) とは、メソッド呼び出しやフィールド参照のような演算子 (-> や . 演算子) の挙動を変更するためのものである。これにより、遠隔メソッド呼び出し [42] や永続 (persistent) オブジェクト[34] の機能を、ライブラリとして実装できるようになる。また高信頼性 (dependability) を実現するライブラリ [10] の実装に用いることもできる。これらの機能は、リフレクションを使わない場合、その機能をもった専用の言語処理系を新たに開発しないかぎり実現できなかった。

3.1 フックを使った実装

我々は C++ 言語でも動作リフレクションを可能にするため、インタープリタの内部構造をオープンにして変更可能にする従来の実装方法ではなく、コンパイルするときにコードにフックを埋め込む実装方法を新たに開発した。またこの方法に基づいて OpenC++ version 1 と呼ぶ処理系を実際に実装した[5]。

我々の方法では、変更された演算子の新しい挙動を、従来方法と同様にインタプリタ風のコードで定義する。しかし、元になる処理系としてインタプリタは使わず、プログラム全体はコンパイルされて実行される。ある演算子の挙動を変更した場合、コンパイルされたコードの中のその演算子に対応する部分に、制御の流れをフック (hook) するコードを埋め込む。フックは、プログラムの実行がその演算子に達したときに、その演算子の新たな挙動として指定されたインタプリタ風のコードを呼び出す役割を果たす。これにより、リフレクションによって挙動が変更されていない部分のプログラムは、通常通りコンパイルされ効率の良いコードで実行される。一方、挙動が変更された部分だけは、新たな挙動として指定されたコードを使ってインタプリタ実行される。

この方法では、言語処理系が各演算子ごとのフックを埋め込む正しい位置を知っていなければならないので、あらかじめ決められた種類の演算子の挙動しか変更できない。つまり動作リフレクションによる変更対象となる演算子の種類が制限される。OpenC++ の場合、変更可能な演算子はメソッド呼び出しなど、オブジェクトの操作に関連するものだけである。その一方、演算対象のオブジェクトごとにメタオブジェクトを変えて、特定のオブジェクトを対象とするメソッド呼び出しの挙動だけを変更できるようにした。

OpenC++ ではさらにメタオブジェクトの設計を工夫して、インタプリタ風のコードといっても、プログラムの文字列を受け取って実行するコードではなく、演算子のオペランドの値を直接受け取って演算を実行するコードで演算子の挙動を定義できるようにした。これは C++ の演算子のオーバーロード (operator overloading) 関数 [36] によく似たコードである。これにより、C++ の演算子のオーバーロードによって演算子の挙動を変更した場合に近い実行効率を達成できた。ただし、その反面、OpenC++ では引数の遅延評価などを動作リフレクションで実現できなくなった。また新たな挙動を定義しやすいように、OpenC++ では標準の挙動を実現するメタオブジェクトのクラスがあらかじめ提供されている。このクラスのサブクラスを定義することで、新たな挙動を実現するメタオブジェクトを容易に作れる。

我々の方法は、C++ の演算子のオーバーロードと似ているが、本質的に異なる部分もある。我々の方法では、演算子のオペランド (メソッド呼び出しの引数列) や演算結果 (メソッド呼び出しの戻り値) 等に対して具現化をおこなう。例えばメソッド呼び出しの挙

動を定義するインタプリタ風コードは、つぎのようなメソッドとして記述される。

```
Result invokeMethod(Arguments args) {
    略
}
```

メソッドの引数列はそのまま渡されず、Arguments オブジェクトに変換されてから渡される。このオブジェクトは、引数の個数や型を抽象化するので、invokeMethod メソッドを特定の引数の個数や型に依存しない総称的 (generic) な形で記述できるようになる。なお Arguments のメソッドを呼んで、引数の個数を調べたり、個々の値を取り出すことも可能である。OpenC++ では、Arguments は引数列を直列化 (serialization) することもできる。戻り値の扱いも同様である。戻り値は Result オブジェクトの形で返す。処理系が埋め込んだフックは、そのオブジェクトから値を取り出し、実際の戻り値とする。C++ の演算子オーバーロードは具現化処理をおこなわない。仮にメソッド呼び出し用の演算子をオーバーロードできたとしても、新しい挙動を 1 つ定義するために、引数の個数や型ごとに別々のコードを多数用意しなければならない。我々の方法では、1 個のコードで全てのメソッドの挙動を定義できる。

フックを埋め込む方法ではリフレクションを使わない限り、プログラムを普通にコンパイルして実行できる。動作リフレクションにともなう速度低下はない。動作リフレクションで演算子の挙動を変えると、その部分だけインタプリタ風のコードで実行されるので実行速度が低下する。しかし実用上、挙動が変更される演算子は一部なので、速度低下の影響は無視できることが多い。OpenC++ の場合、動作リフレクションを使ってメソッド呼び出しの標準的な挙動を実装し直すと、通常の C++ の仮想関数呼び出しと比べて 1 回の呼び出しに数倍から 10 倍の時間がかかるようになった [5]。オーバヘッドの大半は引き数列の具現化によるので、引数の個数が増えるとオーバヘッドも増大する。

3.2 フックの埋め込み方法

OpenC++ の処理系は、コンパイル時のソースコード変換によってフックを埋め込む。したがって、どの演算子の挙動を変更するかは、コンパイル時に静的に決まる。フックの埋め込み方にはいくつかの手法があり、我々のソースコード変換による手法以外にも、後に、いくつかの手法が特に Java 言語向けに提案された。

例えば MetaXa [12] や Guaraná [30] は、改変された Java 仮想機械 (JVM) を使ってフックを埋め込む。とくに Just-in-time コンパイラと連携することで、高い性能を達成している。

また Kava [39] は、ロード時の Java バイトコード変換でフックを埋め込む。コンパイル時ではなく、ロード時にフックを埋め込むので、どの演算子の挙動を変更するか、実行直前に決定できる。また OpenC++ では、メソッド呼び出しの挙動を変更するためのフックを、呼ばれたメソッドの側に埋め込むが、Kava では呼ぶ側に埋め込む。

また JVM のもつデバッガ用インタフェースを利用する手法も提案されている [31]。フックの代わりにブレークポイントを利用するので、実行時にフックを埋め込むことがでいる。しかしこの方法は実行速度の低下が大きいのが欠点である。そこで我々は、デバッガ用インタフェースがもつ、ロード済みのクラスファイルを破棄し、新しいものをロードし直す機能も利用し、可能なものから順にフックを静的に埋め込んだクラスファイルに実行時に置き換えてゆく手法を提案した [44] [32]。

3.3 コンパイル時 MOP

フックを埋め込む方法は、動作リフレクションが使われない場合の実行速度を大きく改善した。しかし挙動が変更された演算子の実行はインタプリタ風のコードで定義されるので、動作リフレクションが使われた場合の実行速度はあまり改善されない。インタプリタ風のコードによる実行効率の悪さを回避するため、ABCL/R の研究グループは部分評価を応用した方法を開発し、優れた成果をおさめた [25] [27]。しかし、C++ や Java 言語向けの性能のよい部分評価器

を開発するのは難しく、この方法はそのまま適用できなかった。

部分評価器を使わずに実行効率の問題を回避するために、我々はコンパイル時 MOP (MetaObject Protocol) と呼ぶ方法を開発し、これを OpenC++ version 2 [2] として実装した。また Java 言語版として OpenJava [37] を実装した。

この方法では、演算子の新しい挙動をインタプリタ風のコードではなく、プログラムの断片の構文木を受け取り、それを適当に変形して返す、Lisp のマクロ関数のようなメソッドとして定義する。処理系はフックを埋め込む代わりに、コンパイル時にこのマクロ関数を呼び出し、変形されたプログラムの断片で元の断片を置き換える (マクロ展開する)。演算子の新しい挙動を定義するには、その演算子を根とする構文木を受け取って、新しい挙動を実現するようなコードの構文木に変形するマクロ関数を定義すればよい。

例えば、メソッド呼び出しのたびにログ出力をおこなうようにメソッド呼び出しの挙動を変更するとする。元のプログラムの断片が C++ 言語で

```
rectangle->draw(x, y);
```

であるとすると、この構文木を引数にとり、

```
(puts("draw"),
 rectangle->draw(x, y));
```

のように変形した構文木を返すメソッドを書けばよい。

フックを埋め込む方法と異なり、マクロ展開後のコードはプログラムに直接埋め込まれ、他の部分と一緒にコンパイルされるので、フックをはじめ動作リフレクション自体に由来するオーバヘッドはなくなる。フックを埋め込む方法では、メソッド呼び出しの引数列を具現化し、型や個数を抽象化したオブジェクトに変換した。これにより 1 個のインタプリタ風コードで、多数のメソッド呼び出しの挙動を包括的に定義できるようにした。マクロ関数を使う方法でも、メソッド呼び出しの引数列や演算子のオペランドは構文木に変換されてからマクロ関数に渡される。この変換は具現化とみなせ、同様に、1 個のマクロ関数で多数のメソッド呼び出しの挙動を包括的に定義できる。フックを埋め込む方法では、具現化を実行時におこなうので速度低下の原因となったが、マクロ関数を使う方法ではコンパイル時におこなうので実行時の速度には影響しない。また個々のプログラムの断片に最適化されたコードに変形するようにマクロ関数を定義できる。インタプリタ風のコードで包括的に定義した際に見られるような速度低下はない。

直感的には、この方法は、部分評価器が生成するであろうコードを生成するマクロ関数を手で書く方法といえる。任意の入力に対応しなければならない部分評価器と違い、特定種類のプログラムの断片だけに対応すればよいマクロ関数を書くのは容易である。OpenC++, OpenJava は、これによって C++ や Java 向けの性能のよい部分評価器の開発の困難さを回避する一方、マクロ関数を書かせるため、メタプログラムの書き手に負担をかけているといえる。

一方、コンパイル時 MOP では、どの演算子の挙動を変更するかをコンパイル時に静的に決定しなければならない。また、新しい挙動を直感的にわかりやすいインタプリタ風のコードではなく、マクロ関数として定義しなければならない。マクロ関数の定義を容易にするため、OpenC++, OpenJava では、例えばマクロ関数中で、プログラムのテキストを表す文字列を構文木に変換する機能を用意している。また、構文木には型情報等を付加し、それを利用した変形が可能になるようにしている。また、メソッド呼び出しのような、オブジェクトを操作する演算子の場合、特定のクラスのオブジェクトに対する挙動だけを新しいものに置き換えることができる。例えば、Point オブジェクトのメソッド呼び出しの挙動だけを変更する、ということが可能である。

コンパイル時 MOP に似た技術として、オープン・コンパイラと呼ばれる技術がある [22]。これはコンパイラの内部データやコードの一部を変更できるようにコンパイラを作成するというものであり、オープン実装 [20] の一種である。システム例としては、Java 仮想機械の just-in-time コンパイラをオープンにした OpenJIT [28] や、並列処理拡張された C++ コンパイラをオープンにした MPC++ [14] がある。また、オープンな実装の Java のソースコード変換器 EPP [13] もある。

コンパイル時 MOP とこれらのオープン・コンパ

イラとの違いは、プログラミング・インタフェースである。オープン・コンパイラでは、基本的にモジュール性を重視して設計されたコンパイラの内部をそのままオープンにしている。多くの場合、プログラミング・インタフェースは、ビジターパターン [11] で構文木をたどるためのクラスである。一方、コンパイル時 MOP はあくまで動作リフレクションの実装技術であるので、プログラミング・インタフェースは、挙動の記述の仕方を除き、フックを埋め込む方法で実装された動作リフレクションと同様である。言語処理系の内部構造にかかわりなく、メタオブジェクトが元の (ベースレベル) プログラムを具現化し、演算子の挙動を制御するという、リフレクション独自の抽象モデルをプログラミング・インタフェースとして提供する [3] [43]。原理的には、他のオープンコンパイラの上に、メタオブジェクトに基づいた抽象モデルを実現するライブラリを置いて、コンパイル時 MOP を実装することも可能である。

4 構造リフレクション

構造リフレクション (structural reflection) とは、プログラム中の型やクラスの定義を変更したり、新たに定義したりする機能である。動作リフレクションは、演算子 (特にメソッド呼び出し) の挙動を変えることを目的とするが、構造リフレクションはプログラムの静的な側面である型やクラスの定義を変えること、新たに追加することを目的とする。例えば、親クラスを変えたり、フィールドを追加することが目的である。

メソッドの定義を変えることも可能なので、機能的には構造リフレクションは動作リフレクションを包含しており、構造リフレクションの機能を使って動作リフレクションを実装するライブラリを書くことも可能である [4]。逆に、純粋な動作リフレクションだけで、新しいクラスを定義する機能がないシステムでは、対象となるプログラムに合わせた新しいクラスの定義を実装上必要とするような言語拡張に対応できない。

構造リフレクションは例えば ObjVlisp [8] や CLOS MOP [16] 等で提供されている。これらの処理系では、型やクラスの定義を表しているインタプリタ内部のデータをプログラムから変更できるようにして構造リフレクションを実現している。このため、この実装方法を C++ 言語のような、実行コードやメモリ上のデータ配置を最適化する言語へそのまま応用することはできないと考えられてきた。Java 仮想機械は、ロード済みのクラス定義を破棄して、新しいクラス定義を再ロードする機能を内蔵するので、この HotSwap 機構を利用すると、Java 言語で構造リフレクションが実現できるように一見思える。しかし新しいクラス定義で変更できるのはメソッドの中身だけであり、新しいフィールドをクラスに追加することなどはできない。これは構造リフレクションが、メモリ上の効率的なデータ配置を困難にするからと思われる。

我々は、どのような構造リフレクションをおこなうかをコンパイル時またはロード時に静的に決定するという制限を加え、代わりに効率のよい実行を可能にする実装方法を開発した。またこの実装方法に基づき、OpenC++、OpenJava を拡張して構造リフレクションを可能にした。さらに構造リフレクションの内容をロード時に決定できる Java 言語用の処理系として Javassist を実装した [4] [41] [7]。

この実装方法では、構造リフレクションを指示するコード (メタプログラム) はロード時またはコンパイル時に実行され、その結果指示される型やクラスの変更にしたがって処理系がプログラムを静的に変換する。OpenC++、OpenJava はソースレベルで、Javassist はバイトコードのレベルでプログラムを変換する。これにより、構造リフレクションの利用による実行時のオーバヘッドを避けることができる。我々の経験によれば、リフレクションの典型的な応用例である各種の言語拡張では、実行時に動的に構造リフレクションをおこなう必要性はほとんどないので、この制約は実用上許容できると考える。必要な処理は、ロード時またはコンパイル時にメタプログラムがプログラムの残りの部分の内容を具現化して調べ、それに基づいて適切な変更をその部分へ加えることである。

例えば Javassist を使うと、プログラム中のクラスやメソッドを Java Reflection API と同様に具現化したオブジェクトをメタプログラム中で取り扱えるようになる。ただしメタプログラムはロード時またはコ

ンパイル時に実行されるので、Reflection API と異なり、このオブジェクトを介してメソッドを呼び出したりすることはできない。一方、Javassist では、このオブジェクトを介して、クラスやメソッドの定義を変更することもできる。変更結果は、ロード時またはコンパイル時に元のプログラムに静的に反映される。

Javassist ではメタプログラムをロード時またはコンパイル時しか動かせないが、この制限を除くと、プログラミング・インタフェースは ObjVlisp や CLOS MOP と同等である。Javassist は指示された変更をバイトコードの変換でプログラムに反映するが、この際、指示された変更内容をバイトコードの変換に翻訳して当てはめる。変更の指示は、クラスやメソッドを表すオブジェクトを介してなされるので、他と同様、ソースコード中の語彙に基づいて作られた比較的理解しやすい抽象モデルを介して指示できるといえる。例えば次のプログラム

```
CtClass p = 略;
CtMethod m = CtNewMethod.make(
    "void xmove(int dx) { x += dx; }",
    p);
p.addMethod(m);
```

は、Point クラスに xmove という新しいメソッドを追加する。p がさす CtClass オブジェクトは、Point クラスを表すオブジェクトである。新しいメソッドを表す CtMethod オブジェクトを作り、p の addMethod メソッドを呼んで追加する。新しいメソッドの定義が文字列で与えられているが、Javassist はこれを対応するバイトコードに翻訳し、Point のクラスファイルの中に挿入する。

リフレクション独自の抽象モデルとソースコードまたはバイトコードとの間の翻訳を暗におこなう点が、我々のシステムとオープン・コンパイラに分類される類似のシステムとの違いである。例えば BCEL [9] や JMangler [21] は、Javassist に似た Java のバイトコード変換をおこなうためのシステムだが、利用者はどのようにバイトコードを変換するかシステムに明示的に指示しなければならない。したがって利用者は Java バイトコードの知識を持っていなければシステムを使えない。一方、Javassist の場合、利用者は通常の Java 言語の知識さえ持っていれば、リフレクション独自の抽象モデルを通じて変換を指示できる。バイトコードをどのように変換すべきかは Javassist が判断する。EPP [13] のようなソースレベルで Java のプログラム変換をおこなうシステムでも同様である。利用者はコンパイラの構文木の知識が必要になり、Javassist ほど予備知識なしに使えない。

5 まとめにかえて — アスペクト指向プログラミング

本論文では、著者らのリフレクションに関するこれまでの研究を概観した。最後に、まとめにかえて、今後の研究の展望を著者らの主観を交えて述べる。

約 20 年前に研究が始まったリフレクション技術は、多くの研究者らの手により実用的に利用できる技術となった。具現化機能だけとはいえ、Java 言語をはじめ主要な言語にも標準機能として取り込まれた。またリフレクションを使った多くの実用システムが作られ、例えば著者らの OpenC++ や OpenJava も多数の利用者を得ている。

現在、この分野の研究者の多くは、**アスペクト指向プログラミング** (AOP: Aspect Oriented Programming) [17] の研究に移ってきている。AOP は、ポスト・オブジェクト指向技術とでも呼ぶべきプログラムの新しいモジュール化技術である。20 年近くにわたる研究により、リフレクションの主要な応用のひとつは、**関心事の分離** (separation of concerns) を促進するような言語拡張である、との認識がもたれるようになった。関心事とは、エラー処理や画面の更新といった、開発者から見たソフトウェアの機能単位である。ソフトウェアは、関心事ごとに別々のモジュールに分けて実装できることが望ましいが、現実には難しい。そこで、より高度なモジュール化を可能にするようにリフレクションを応用する萌芽的研究が、AL-1/D [29] をはじめ、数多くなされた。このような研究の影響を受け、AOP の研究は、その中の特に、オブジェクト指向技術に代表されるような従来技術ではうまくモジュール化できない**横断的関心事** (crosscutting concerns) をモジュール化することを狙いとして、1990 年代後半に始まった。

オブジェクト指向技術では、クラスがモジュールの単位となる。しかし、どのようにクラスを設計しても、複数のクラスにコードが分散してしまう処理がある。例えば、図形エディタを作る場合、画面上の直線や円などの図形をクラスとするのが自然だが、それでは図形の状態が変化した際の画面の再描画処理が全ての図形クラスに重複して含まれてしまう。全ての図形クラスに共通の親クラスを作り、そこに再描画処理関連のコードを集める方法はこの場合うまく働かない。この再描画処理のような処理を横断的関心事という [19]。

AOP 言語は、横断的関心事を独立したモジュールにすることを可能にする。そのようなモジュール内のコードは、他のモジュール (例えばクラス) の中の指定されたコードが、指定された条件の下で実行されたときに、同時に (その直前直後に) 実行される。このような同時実行を実現するためのコンパイル技法に、リフレクションで開発された技術が応用されている。実際、代表的な AOP 言語である AspectJ [19] の処理系がおこなう処理は、Javassist が可能にする構造リフレクションの機能で実装可能である。おおまかにいって、AspectJ の inter-type 宣言は構造リフレクションに、advice は動作リフレクションに対応する。

現在の AOP の研究課題のひとつは、横断的関心事にかかわるコードを実行する場所を柔軟に指定する方法である。AspectJ では、プログラム中のメソッド呼び出しやフィールド参照などを比較的単純な規則で選んで指定することしかできない。我々は現在、Javassist を Java 言語で AOP をおこなうためのライブラリとして利用し、より高度な実行場所の指定をおこなえるようにする研究をしている。また現在 Javassist を使うには比較的複雑な Java プログラムを書かなければならないが、AspectJ 風の簡潔な構文による記述で Javassist を使うためのシステム Josh [6] の開発もおこなっている。

リフレクションの研究では、比較的自由にプログラムの挙動や使われている型やクラスの定義を変更できるようになった。その一方、リフレクションを使ったプログラムは難しくなりがちである。AOP は、リフレクションの重要な応用であった高度なモジュール分割に特化し、リフレクションから直接必要ない機能をそぎ落とし、抽象モデルを簡素化し、直感的でわかりやすい構文でプログラムできるようにしようとしている、と考えられる。実際、リフレクションでは総称的な記述のために重要であった具現化機能は、AOP ではあまり重視されていないように思える。AOP では開発ツールによるプログラミングの支援も重要な課題である。現在は、研究 seeds 主導で進んできたリフレクションの研究が、横断的関心事という実用面の needs により AOP と形を変えて修正を受けている過程なのかもしれない。

参考文献

[1] Buschmann, F., Kiefer, K., Paulisch, F., and Stal, M.: The Meta-Information-Protocol: Run-Time Type Information for C++, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*, 1992, pp. 82–87.

[2] Chiba, S.: A Metaobject Protocol for C++, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 30, No. 10, ACM, 1995, pp. 285–299.

[3] Chiba, S.: Macro Processing in Object-Oriented Languages, *Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98)*, IEEE Press, 1998, pp. 113–126.

[4] Chiba, S.: Load-time structural reflection in Java, *ECOOP 2000*, LNCS 1850, Springer-Verlag, 2000, pp. 313–336.

[5] Chiba, S. and Masuda, T.: Designing an Extensible Distributed Language with a Meta-Level Architecture, *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, Springer-Verlag, 1993, pp. 482–501.

[6] Chiba, S. and Nakagawa, K.: Josh: An Open AspectJ-like Language, *Int'l Conf. on Aspect Oriented Software Development (AOSD'04)*, 2004.

[7] Chiba, S. and Nishizawa, M.: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, *Proc. of Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, Springer-Verlag, 2003, pp. 364–376.

[8] Cointe, P.: Metaclasses are first class: The ObjVlisp model, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1987, pp. 156–167.

[9] Dahm, M.: Byte Code Engineering with the JavaClass API, Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999.

[10] Fabre, J., Nicomette, V., Pérennou, T., Stroud,

R. J., and Wu, Z.: Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming, *Proc. of the 25th IEEE Symp. on Fault-Tolerant Computing Systems*, 1995, pp. 489–498.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns*, Addison-Wesley, 1994.

[12] Golm, M. and Kleinöder, J.: Jumping to the Meta Level, Behavioral Reflection Can Be Fast and Flexible, *Proc. of Reflection '99*, LNCS 1616, Springer, 1999, pp. 22–39.

[13] Ichisugi, Y.: The extensible Java pre-processor EPP, `http://staff.aist.go.jp/y-ichisugi/epp`, 1998.

[14] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., and Kubota, K.: Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —, *Proc. of Reflection 96*, Apr. 1996, pp. 153–166.

[15] Java Soft: Java^{TM} Core Reflection API and Specification, Sun Microsystems, Inc., 1997.

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

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J.: Aspect-Oriented Programming, *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, Springer, 1997, pp. 220–242.

[18] Kiczales, G. J. and Rodriguez Jr., L. H.: Efficient Method Dispatch in PCL, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990, pp. 99–105.

[19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G.: An Overview of AspectJ, *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, Springer, 2001, pp. 327–353.

[20] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., and Murphy, G.: Open Implementation Design Guidelines, *Proc. of the 19th Int'l Conf. on Software Engineering (ICSE'97)*, 1997, pp. 481–490.

[21] Kniesel, G., Costanza, P., and Austermann, M.: JMangler — A Framework for Load-Time Transformation of Java Class Files, *Proc. of IEEE Workshop on Source Code Analysis and Manipulation*, 2001.

[22] Lamping, J., Kiczales, G., Rodriguez, L., and Ruf, E.: An Architecture for an Open Compiler, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*, 1992, pp. 95–106.

[23] Liang, S. and Bracha, G.: Dynamic Class Loading in the Java^{TM} Virtual Machine, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1998, pp. 36–44.

[24] Maes, P.: Concepts and Experiments in Computational Reflection, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1987, pp. 147–155.

[25] Masuhara, H., Matsuoka, S., Asai, K., and Yonezawa, A.: Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1995, pp. 300–315.

[26] Masuhara, H., Matsuoka, S., Watanabe, T., and Yonezawa, A.: Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1992, pp. 127–144.

[27] Masuhara, H. and Yonezawa, A.: Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Languages, *ECOOP'98 - Object Oriented Programming*, LNCS 1445, Springer, 1998, pp. 418–439.

[28] Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y., and Kimura, F.: OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java, *ECOOP 2000*, LNCS 1850, Springer-Verlag, 2000, pp. 362–387.

[29] Okamura, H. and Ishikawa, Y.: Object Location Control Using Meta-level Programming, *Proc. of the 8th European Conference on Object-Oriented Programming*, LNCS 821, Springer-Verlag, 1994, pp. 299–319.

[30] Oliva, A. and Buzato, L. E.: The Design and Implementation of Guaraná, *Proc. of 5 th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, The USENIX Association, 1999, pp. 203–216.

[31] Popovici, A., Gross, T., and Alonso, G.: Dynamic Weaving for Aspect-Oriented Programming, *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, ACM Press, 2002, pp. 141–147.

[32] Sato, Y., Chiba, S., and Tatsubori, M.: A Selective, Just-In-Time Aspect Weaver, *Proc. of Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, Springer-Verlag, 2003, pp. 189–208.

[33] Smith, B. C.: Reflection and Semantics in Lisp, *Proc. of ACM Symp. on Principles of Programming Languages*, 1984, pp. 23–35.

[34] Stroud, R. J. and Wu, Z.: Using Metaobject Protocols to Implement Atomic Data Types, *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, Springer-Verlag, 1995, pp. 168–189.

[35] Stroustrup, B.: *The Design and Evolution of C++*, Addison-Wesley, 1994.

[36] Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, special edition, 2000.

[37] Tatsubori, M., Chiba, S., Killijian, M.-O., and Itano, K.: OpenJava: A Class-based Macro System for Java, *Reflection and Software Engineering*, LNCS 1826, Springer Verlag, 2000, pp. 119–135.

[38] Watanabe, T. and Yonezawa, A.: Reflection in

an Object-Oriented Concurrent Language, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1988, pp. 306–315.

[39] Welch, I. and Stroud, R.: From Dalang to Kava — The Evolution of a Reflective Java Extension, *Proc. of Reflection '99*, LNCS 1616, Springer, 1999, pp. 2–21.

[40] Yonezawa, A.(ed.): *ABCL: An Object-Oriented Concurrent System*, The MIT Press, 1990.

[41] 千葉滋, 立堀道昭: Java バイトコード変換による構造リフレクションの実現, **情報処理学会論文誌**, Vol. 42,No. 11(2001), pp. 2752–2760.

[42] 千葉, 益田: 自己反映言語 Open C++とその分散処理への適用の実際, **コンピュータソフトウェア**, Vol. 11,No. 3(1994), pp. 33–48.

[43] 立堀道昭, 千葉滋, 板野肯三: クラスオブジェクトを用いた Java 言語用マクロ処理系, **情報処理学会論文誌**, Vol. 41,No. 8(2000), pp. 2327–2338.

[44] 佐藤芳樹, 千葉滋: 効率的な Java Dynamic AOP システムを実現する Just-in-Time weaver, **情報処理学会第４２回プログラミング研究会** *(2002-4)*, 情報処理学会, 2003.