安全な自己反映計算にむけて
(Extended Abstract)

千葉 滋
筑波大学 電子・情報工学系
科学技術振興事業団さきがけ研究 21
Email: chiba@is.tsukuba.ac.jp

1 はじめに

自己反映計算 (reflection) とは、プログラムによって (自分自身または別な) プログラムの動作を変更する処理である。自己反映計算が可能であると、さまざまなプログラム変換を比較的容易にプログラムによって自動化できる。例えば分散を考慮せずに書かれたプログラムを、分散環境を考慮したプログラムに変換するなど、言語の拡張機能を実装するために利用することができる。また、ガベージコレクタなど実行時システムの動作も変更可能にしている処理系では、自己反映計算によって、アプリケーションの内容に合わせてガベージコレクションのアルゴリズムを最適化することができる。

これまでに自己反映計算を可能にした処理系が数多く提案、開発されてきたが、強力な自己反映計算機能はもろ刃の剣であることが知られている。自己反映計算機能が強力になればなるほど、複雑なプログラム変換や実行時システムの最適化が可能になるが、一方、自己反映計算をおこなうプログラム自体の開発が困難になる。プログラムの誤りにより、予期せぬプログラムの異常動作を引き起こしたり、異常な実行速度の低下をまねいたりする。

本論文では、我々が開発している Java 言語用の自己反映計算機構 Javassist [8, 3] を題材に、安全な自己反映計算をおこなうための枠組みについて論ずる。Javassist は、バイトコード (クラスファイル) のロード時に構成的な (structural) 自己反映計算をおこない、クラスの定義の内容を変更することを可能にする。しかし安易に構成的な自己反映計算機構を設計すると、プログラムを破壊してしまうような自己反映計算をおこなうプログラムが簡単に書けるようになってしまう。特に問題なのは、自己反映計算の誤りが自己反映計算の実行直後にはわからず、変更をおこなったクラスを最終的に Java 仮想機械 (JVM) にロードするまでわからないことである。JVM はバイトコード検査器を内蔵しているので、誤りを含んだバイトコードをロード、実行して、JVM 全体が誤動作することはない。しかしながら一連の自己反映計算実行後に誤りがわかったとしても、どの自己反映計算がその誤りを引き起こしたのか特定するのは容易ではない。

本稿で述べる枠組みでは、この問題を避けるため、compatible class という概念を導入する。あるクラス C の compatible class とは、C のクラス定義と置き換えても、型に関する不整合をおこさないような定義をもつクラスのことである。Javassist の個々の自己反映計算のプリミティブは、変更後のクラス定義が、常に変更前のクラス定義の compatible class となるように自己反映計算をおこなう。Compatible class とならないような変更をおこなうプリミティブはそもそも提

供されないか、あるいは実行時に例外を発生してエラーを通知する。この枠組みによって、自己反映計算によって型として誤ったクラス定義を生成してしまうことを防ぐ、あるいは少なくとも個々の自己反映計算のプリミティブを実行した直後に、誤った型を生成する処理であることを発見することができる。

2 自己反映計算によるプログラムの破壊

ある程度以上の機能をもった自己反映処理機構を使うと、プログラムの動作を破壊するような (メタ) プログラムを記述することは比較的容易である。以下では、いくつかの既存の典型的な自己反映処理機構を選び、どのようなメタプログラムがプログラムの動作を破壊するのか、具体的に紹介する。

2.1 メソッド呼出しの動作変更

多くの自己反映計算機構は、メソッド呼出しを実行時に横取りし、メソッド呼出の意味を変更できるようにしている。そのような自己反映機構に、OpenC++ [1] や MetaXa [6] などがある。このような自己反映計算は、behavioral reflection と呼ばれる。

メソッド呼出しが実行時に横取りされると、典型的には、呼び出されたオブジェクトに対応する別なオブジェクト (メタオブジェクトと呼ばれる) の次のようなメソッド (ここでは記述言語として Java を用いる) が呼ばれる。

```
Object methodWasCalled(Object target, Method method, Object[] params)
{
    // 適当な動作でメソッド呼出しを実行する。
    // 実行したメソッドの返り値を result とする。
    return result;
}
```

このメソッドは、引数として、呼び出されたオブジェクトを `target`、呼び出されたメソッドを `method`、横取りしたメソッド呼出しの引数を `params` として受け取る。そしてメソッド呼出しを標準とは異なる動作で実行し、実行結果をそのまま返す。

このような自己反映計算機構を利用したメタプログラムは、次のような形でプログラムを破壊する可能性がある。まず、メタプログラムはメソッド呼出し機構の動作仕様を完全に変えることができるので、例えばまったくメソッド呼出しをおこなわず、常に `null` を返すようにできる。この場合、元のプログラムはまったく動作しなくなる。また、`methodWasCalled()` が返す値の型が、元々呼び出されたメソッドの返り値の型と一致しなければ、実行時に型エラーが発生する。後者のような誤りは起こりがちであるが、実行時になるまで誤りが発見できないので、メタプログラムの開発上問題である。`methodWasCalled()` の返り値の型を Object 型でなく、元々呼び出されたメソッドの返り値と同じ型にすれば、実行時に型エラーが発生することはないが、自己反映計算機構のアプリケーションは `methodWasCalled()` の返り値の型が Object 型であることによる一般性を利用していることが多い。

2.2 構成的な自己反映計算

構成的 (structural) な自己反映計算は、例えばクラスの定義の内容をメタプログラムから変更することを可能にする。原理的には任意のプログラム変換が、静的あるいは動的に、抽象化されたインタフェースを通して実行可能になる。このような自己反映計算を可能にする機構としては、CLOS MOP [7]、ObjVlisp [5]、OpenC++ [2] などがある。

構成的な自己反映計算を提供する機構は機能を強化するのが容易である。しかし無闇に機能を強化すると、メタプログラムが簡単にプログラムを破壊できるようになってしまう。例えばメタプログラムが、あるクラスのメソッドを削除してしまったとする。もしそのメソッドが他のクラスから呼ばれている場合、実行時にメソッドが発見できず実行時エラーが発生するだろう。CLOS MOP や OpenC++ は、構成的な自己反映計算を静的に実行するので、実行時エラーは発生せず、必要なメソッドが不足していることはコンパイル時あるいはロード時に発見される。しかしながら、発見されるのは一連の自己反映計算が終了した後なので、どの自己反映計算のプリミティブがそのエラーを引き起こしたのかを特定するのは、必ずしも容易ではない。

3 安全な自己反映計算

本章では、自己反映計算によって誤ってプログラムを破壊してしまうことをできるだけ防ぐ方法を、構成的な自己反映計算に関して述べる。この方法の下では、可能な自己反映計算に一定の制約があたえられ、型として誤ったクラスを生成するような自己反映計算のプリミティブは提供されないか、少なくとも自己反映計算の実行時に引数を検査して即座にエラーを発生する。個々のプリミティブを実行した直後にエラーを発見できるので、プログラムの修正が容易である。一連の自己反映計算が終わった後、コンパイル時、ロード時、あるいは実行時までエラーの存在がわからないということはない。

以下では、まず compatible class という概念を提案し、次にこれを使って設計された我々の自己反映計算機構 Javassist が、どのような制約を自己反映計算に課しているかを述べる。

3.1 Compatible classes

あるクラス C の compatible classes とは、直感的には、クラス C の定義と置き換えても型エラーをおこさないような定義を持ち、かつ名前 C をもったクラスのことである。プログラムの中からクラス C の定義だけを、compatible class の定義と差換えたとしても、型およびシグネチャに関するエラーは発生しない。ただし単純にクラス C の定義を差換えるだけでなく、エラーの発生を防ぐためにプログラム全体を機械的に変換してもよいものとする。型およびシグネチャに関するエラーとは、クラス C に存在しないメソッドの参照など、クラス C の定義をおきかえることにより副次的に発生する他のクラスの定義に関するエラーを含む。

Java 言語の場合、クラス C の compatible class C' は次のような性質を満たす。

- クラス C が public クラスであるならば、C' も public クラスである。クラス C が public クラスでないならば、C' は public クラスでもそうでなくてもよい。
- クラス C が final クラスであるならば、C' は final クラスでもそうでなくてもよい。クラス C が final クラスでないならば、C' も final クラスでない。

- クラス C が abstract クラスであるならば、C' も abstract クラスである。クラス C が abstract クラスでないならば、C' も abstract クラスでない。
- クラス C の親クラスが P であるならば、クラス C' は P から直接または間接に継承する。つまり P は C' の祖先クラスである。
- インタフェース I がクラス C のインタフェースなら、I はクラス C' のインタフェースである。
- Public/protected メソッド m がクラス C で直接宣言されているなら、m はクラス C' でも直接宣言されているか、C' の親ないし祖先クラスから C' が m を継承している。
- Private メソッド m がクラス C で直接宣言されているなら、m はクラス C' でも直接宣言されている。C' で宣言されている m は、public または protected でもよい。
- クラス C' は同名かつ同シグネチャのメソッドを複数宣言していない。
- Public/protected フィールド f がクラス C で直接宣言されているなら、f はクラス C' でも直接宣言されているか、C' が継承している 1 つ以上のクラスが f を直接宣言している。
- Private フィールド f がクラス C で直接宣言されているなら、f はクラス C' でも直接宣言されている。C' で宣言されている f は、public または protected でもよい。
- クラス C' は同名のフィールドを複数宣言していない。
- static メソッドおよび static フィールドがクラス C で宣言されているなら、クラス C' で宣言されている対応するメソッドおよびフィールドも static である。
- クラス C' が直接宣言するメソッド m は、型的に正しい。例えば m の中で参照しているメソッドやフィールドが存在する。

3.2 Javassist

Javassist は、Java 言語中で構成的な自己反映計算を可能にするための機構である。この機構は標準の JVM を使えるようにするため、バイトコード (クラスファイル) をロードする際にだけ、自己反映計算を許しているのが特徴である。非常に単純な見方をするならば、Javassist はバイトコードを修正し、クラス定義をロード時に変更するための Java のクラス・ライブラリであるともいえる。ただし JOIE [4] など類似のクラス・ライブラリと比較するとインタフェースの抽象度が高く、バイトコードの知識がなくともクラス定義を変更できるのが特徴である。

Javassist は Java 言語が標準で装備する機能を使って、JVM がバイトコードをロードする際、これを横取りする。そして横取りしたバイトコードをメタプログラムの指示にしたがって修正し、それを JVM に返して実際にロードさせる。Javassist によって横取りされたバイトコードは、クラスごとに CtClass (compile-time class) オブジェクトによって表現される。メタプログラムはこの CtClass オブジェクトを通してバイトコードを修正する。

クラス CtClass には、大きく分けて、クラス定義を観察するためのメソッドと、それを修正するためのメソッドが定義されている。メタプログラムはこれらのメソッドを呼び出すことで、クラス定義の内容を調べ、必要なら修正をおこなう。クラス定義を観察するためのメソッドは、Java

言語に標準で装備されている reflection API が提供するメソッドとほぼ同等である。`getName()` (クラスの名前を得る)、`getSuperclass()` (親クラスを得る)、`getDeclaredMethods()` (直接宣言されているメソッドを得る) などが用意されている。一方、クラス定義を修正するためのメソッドは、`setSuperclass()` (親クラスを変更する)、`addMethod()` (メソッドを追加する) などが用意されている。またクラス定義の中のメソッドの定義を部分的に修正するメソッドも用意されている。

クラス定義を修正するためのメソッドとしては、原理上、クラス定義の任意の要素を修正するためのメソッドを提供することが可能である。しかしながら Javassist は、メタプログラムによる誤ったクラス定義の破壊を可能な限り防ぐため、提供する CtClass のメソッドの種類に制限を加えている。クラス定義を修正するメソッドは、次のような性質を満たすものだけが提供される。

- クラス C の定義に対して、メソッドを実行した結果は、クラス C の compatible class であるか、エラーを生成する。

したがって CtClass のメソッドには、既存のフィールドの型を変更するようなものが存在しない。また新しいメソッドを追加するために `addMethod()` が用意されているが、既に存在するメソッドと同名かつ同シグネチャのメソッドを追加しようとすると、エラーを発生する。

上の性質を満たさないメソッドを使ったメタプログラムが、必ずプログラムを破壊してしまうわけではない。例えば既存のフィールドの型を変更したとしても、そのフィールドを操作している全ての式も共に変更して、新しい型に対応できるようにすれば、最終的に得られるプログラムは正しく動作する。しかしながら、一連の自己反映計算によるクラス定義の修正が、結果的に正しいクラス定義を生成したとしても、その途中で一時的せよ非 compatible class の出現を許してしまうと、Javassist は個々の自己反映計算 (CtClass のメソッドの個々の呼出し) が正しいか否か機械的に検査できなくなってしまう。一連の自己反映計算終了後にクラス定義が compatible class であるか否かを検査する方法では、仮に誤りが発見されたとき、個々の自己反映計算のうちどこが誤っていたのかを発見することが困難である。

Javassist が提供する CtClass のメソッドは、クラス定義を局所的に変更するものばかりであるが、これは Javassist がおこなうバイトコードの修正も各クラスに局所的であることを意味しない。例えば Java バイトコードでは private メソッドと public メソッドとで、メソッドを呼出すためのバイトコードが異なる。あるクラス C で宣言されているメソッド m が private から public に自己反映計算の前後で代わったら、クラス C で宣言されている全てのメソッドについて、m の呼出しに使われているバイトコードを変更しなければならない。もし変更が他のクラスのメソッドに及ぶ場合には、変更の必要を記録しておき、後にそのクラスがロードされる際に、暗黙のうちにバイトコードの変更をおこなう。変更を必要とするクラスが既にロードされた後であり変更ができない場合には、その自己反映計算はエラーを発生する。

4 まとめ

本論文では、野放図な自己反映計算によって、メタプログラムが誤ってプログラムを破壊してしまうことがないよう、可能な自己反映計算の種類に一定の制限をかける方法について述べた。この方法では、compatible class という概念を導入し、型として誤ったクラスを生成するような自己反映計算を不可能にする。そのような自己反映計算を可能にしてしまうような機能は、そもそも提供されないか、あるいは自己反映計算の実行時に即座にエラーを発生する。

現在の Javassist の設計では、既に存在するクラスを修正している限り、メタプログラムが型として誤ったクラスを生成することはできない。しかし Javassist は新しいクラスを、既存クラスを参照せずに無から生成する機能も用意している。メタプログラムがこの機能を使い、新しいクラスを無から生成し、既存クラスと同じ名前を最後に与えて、その既存クラスを置き換えてしまうと、Javassist はクラス定義が型として正しいか否かを検査することはできるが、仮に誤っていた場合、どの自己反映計算によってその誤りが引き起こされたか発見することはできない。この問題に対する対処は今後の課題である。

また本稿で述べた方法では、型に関する不整合は排除できるが、これだけでは安全な自己反映計算を達成できたとは言い難い。自己反映計算後のプログラムの振舞いにも、一定の制約をかけられるような仕組みが望まれる。

参考文献

References

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

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

[3] Chiba, S., "Load-time structural reflection in Java," in *Proc. of ECOOP'2000*, Springer Verlag, 2000. To appear.

[4] Cohen, G. A., J. S. Chase, and D. L. Kaminsky, "Automatic Program Transformation with JOIE," in *USENIX Annual Technical Conference '98*, 1998.

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

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

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

[8] 千葉, "Java バイトコードを編集するための API," in **第 *2* 回プログラミングおよび応用のシステムに関するワークショップ** *SPA '99*, 日本ソフトウェア科学会, 3 1999.