共通コンパイラインフラストラクチャに基づいた Java コンパイラの検討

首藤 一幸

2002 年 7 月 4 日版

1 背景

COINS プロジェクトは、複数言語、複数機種への対応が容易な共通コンパイラインフラストラクチャの設計、開発を行っている。フロントエンドとしては、現在、C 言語および FORTRAN 77 から HIR への変換器の実装が行われている。

平成 15 年からの第 II 期では、共通インフラストラクチャの実用性向上や、種々のソース言語への対応が容易であることを実証するために、ソース言語を 3 つまで増やすことを計画している。そこで新たに追加するソース言語の有力な候補として、*Java* **言語**が挙げられている。これは、Java 言語が C や FORTRAN 77 と並んで広く使われていること、高水準中間表現（HIR）が前提とする手続き型言語であること、が理由である（？）。

Java コンパイラを開発するには、新たなパーサの開発が必要となることは当然だが、共通インフラストラクチャ自体の中間表現、特に HIR の改変や機能追加も予想される。しかしそれ以前に、まず、何を入力として何を出力するのかというコンパイラとしての基本的な性質について、方針を定めなければならない。本稿では様々な選択肢ごとに得失を述べる。工学的選択の一助としたい。

2 フロントエンドの作り方

クラスファイルという選択肢

「Java コンパイラ」には、当然ながら、Java 言語で書かれたソースプログラムを入力として扱えることが要求される。しかし、Java 言語で書かれたソースプログラムを直接扱わず、ソースプログラムを Java 仮想マシン向けに変換した**クラスファイル**を入力とする「Java コンパイラ」も多い。これは厳密には「Java バイトコードコンパイラ」であるが、利点が多いので、このアプローチを採っているコンパイラも多い。Just-in-Time（JIT）コンパイラは自然と Java バイトコードコンパイラになるが、実行前にコンパイルを完了する Ahead-of-Time（AOT）コンパイラにも、クラスファイルを入力とするものは多い。AOT コンパイラ [1][2][3][4][5] はすべて、クラスファイルを入力として、C 言語のコードを出力する。

ソースプログラムからクラスファイルへの変換はあまり最適化の許されない比較的素直な変換であり、また、クラスファイルは Java 仮想マシンというスタックマシン用のコードである。そのため、実行効率の良さは、クラスファイルを処理するコンパイラ（やインタプリタ）が行うレジスタ割り付けや最適化にかかっている。つまり、ソースプログラムを実行前にいったんクラスファイル

に変換する場合、実行効率の如何はバイトコードコンパイラが握っている。Java 言語のコンパイラではなくてバイトコードコンパイラだからといって、重要性が低いといったことはまったくない。

また、Java 言語からクラスファイルへのコンパイラ [6][7] は容易に入手できるので、共通インフラストラクチャを単に Java コンパイラとして使う利用者にとっては、バイトコードコンパイラさえあれば Java 言語のコンパイラが提供されているも同然である。

バイトコードコンパイラの利点

Java 言語のコンパイラとバイトコードコンパイラには、それぞれに利点と問題がある。コンパイラの一利用者にとって、入力がクラスファイルであることの利点には次のものがある。

- ソースコードが手に入らないソフトウェアもコンパイルできる。

  商用ソフトウェアには、クラスファイルや、それを束ねた JAR ファイルだけが配布されるものも多い。

- Java 以外の言語も扱える。

  ソースコードを Java のクラスファイルに変換できる言語や、Java 仮想マシン上でインタプリタが動く言語が数多くある。スクリプト言語 Python の Java 実装 Jython [8]、Ruby の Java 実装 JRuby [9]、Kawa [10] や京大湯淺先生のぶぶといった Scheme 処理系などが知られている。文献 [11] には、Lisp と Scheme の類だけで 18 ものソフトウェアが挙げられている。

また、コンパイラの開発者にとっては次に挙げる利点もある。

- Java 言語よりもパーサの開発が容易。

  クラスファイルの形式は Java 言語のソースコードよりははるかに単純であり、字句の解釈が文脈に依ることもなく、バイトコード命令の種類もたかだか 200 前後である。

- JIT コンパイラに仕立てることができる（かもしれない）。

  まずは AOT コンパイラとして開発したバイトコードコンパイラでも、プログラム実行中に呼び出して JIT コンパイラとして働かせることができるかもしれない。その際は、ダイナミックリンカ相当の、リンク、シンボル解決といった処理も実装する必要があろう。これによって、実行時コンパイルを前提とした研究の基盤として用いることができるようになる。

  コンパイラ自体は事前にコンパイルしておいても、OpenJIT [12][13] のように実行中に自身をコンパイルしてもよいだろう。

Java 言語コンパイラの利点

一方、クラスファイルには、最適化や並列化の基盤となるコンパイラインフラストラクチャにとっては都合の悪い点がある。ループといった**高級言語の構造を持たない**点である。Java バイトコードは、オブジェクトの作成といった高級な機能も持つ反面、基本的にはスタックマシンの機械語であるため、メソッド内の制御フローは、ジャンプ命令（`goto`, `if<条件>`, `if_icmp<条件>`, `if_acmp<条件>`, `ifnonnull`, `ifnull`）やメソッド内サブルーチン呼び出し（`jsr`, `ret`）だけで表現される。クラスファイルの元となったソースプログラム中にあったループ、ブロックなどの構造が、クラスファイルでは失われているのである。

伝統的な中粒度並列化の主な対象はループであり、並列化を考えない最適化においてもループは重要な対象である。Intel 社のコンパイラ [14] が行うような SIMD 命令を用いたベクトル化、SMP 向け並列化の対象もやはりループである。

とはいえ、逆コンパイルを行って、バイトコード列からループ構造を復元することも、バイトコード列によっては可能である [15]。実際に、クラスファイル用逆コンパイラがいくつか配布または市販されている [16][17]。手で書いたクラスファイルの逆コンパイルは困難であるし、あらゆるクラスファイルを逆コンパイルできるわけではないが、これらの逆コンパイラは高い割合でソースコードを復元する。また、完全な逆コンパイルを目指さずとも、ループとその誘導変数の検出くらいに目標を限れば、簡単なアルゴリズムでたいていのクラスファイルからほとんどのループを抽出できるかもしれない。

しかしながら、コンパイラインフラストラクチャとしては、HIR 中に現れるループが、ソースプログラムから直接変換されたものではなくて逆コンパイルで間接的に得られたものでよいのか？という問題もあろう。

フロントエンドについての選択

Java 対応を考えた場合、フロントエンドについての検討事項とそれに対する選択肢としては次のものが考えられるだろう。

- 何を入力とするのか？
  - Java 言語のソースプログラム
  - クラスファイル
- 何に変換するのか？
  - HIR
  - LIR
- HIR 上でループ構造を抽出するのか？
  - する
  - あきらめる

ソースプログラムとクラスファイル、双方のパーサを開発するという選択肢もあるだろう。GCC に統合されている Java コンパイラ GCJ [18]（4.1 節）は、ソースコードとクラスファイルのどちらにも対応している。双方に対応しているように見えるコンパイラでも、ソースコードはいったんクラスファイルに変換しているという恐れがあるが、GCJ の場合はどちらを入力とするかによって生成されるコードがまったく異なるので、個別にパーサを用意していると思われる。

Java の入力を AOT コンパイラ [1][2][3][4][5] で処理して、得られた C 言語のコードを共通インフラストラクチャの C フロントエンドに入力する、という手順も考えられる。しかし、これはさすがに Java 対応とは言えないだろう。

Java フロントエンドを使う場合はループ構造はあきらめる、という選択もあるかもしれない。ループを対象とする研究には C、FORTRAN 77 フロントエンドを使い、Java フロントエンドを使った場合は、コード生成までは可能だが、ある種の最適化や並列化はできない、とするのである。もしくは、HIR 上でのループ構造の抽出や逆コンパイルは将来の課題、次のステップであると位置付けて、まずはクラスファイルから HIR への単純な変換器を作るという方針もあり得るだろう。

3　バックエンドが生成するコードのバイナリインタフェース

アプリケーションバイナリインタフェース（ABI）とは、同一の ABI に従うアセンブリコードや機械語コードが相互運用可能となる、というものである。これは例えば、関数呼び出しの際に引数と返り値をどうやって（レジスタ、スタック、…）受け渡すかという呼び出し規約を含む。

C や FORTRAN 77 では、ABI はプロセッサアーキテクチャやリンカが規定する。一方、Java や C++といったオブジェクト指向言語では、その規定だけでは相互運用には不充分であり、処理系が追加の規約を設けている。

まずは、追加規約にはどのようなものがあるのかの例を挙げ、GCC に従う、JNI に従うといったいくつかの指針を挙げる。

- 呼び出し規約

  インスタンスメソッドの呼び出しでは、呼び出し対象オブジェクト（レシーバ）をどう渡すかという問題があり、クラスメソッドではクラスオブジェクトをどう渡すかという問題がある。

  単一の処理系に複数の規約が混在する場合すらある。例えば、次の Java コードがあった場合、

```
package APackage;
class AClass {
  native int aInstanceMethod(int a);
  native static int aClassMethod(int a);
}
```

  Sun 社の Classic VM 本来の ABI である Native Method Interface（NMI）ではこうなるが：

```
int32_t AClass_aInstanceMethod(struct HAClass *, int32_t);
int32_t AClass_aClassMethod(struct HAClass *, int32_t);
```

  同じ Classic VM 内でも、他の Java 処理系との相互運用を考えて作られた Java Native Interface（JNI）という規約ではこうなる：

```
jint Java_AClass_aInstanceMethod(JNIEnv *, jobject, jint);
jint Java_AClass_aClassMethod(JNIEnv *, jclass, jint);
```

  ここでは、受け渡し方法を C 言語で表現した。NMI では、インスタンスメソッドの第一引数は呼び出し対象オブジェクトである。一方、クラスメソッドに渡される第一引数は常に NULL である。それに対して、JNI では、クラスメソッドには第二引数としてクラスオブジェクトが渡される。

  また、GCJ では、GCC の C++コンパイラ g++が次のコードをコンパイルした結果と同じインタフェースとなる。

```
class ::APackage::AClass : public ::java::lang::Object  {
  virtual jint aInstanceMethod (jint);
  static jint aClassMethod (jint);
}
```

  これらの間に、相互運用性は期待できない。

- 引数の型並びを mangling する際のエンコード法

  リンカはシンボルとして文字列しか扱えないため、オブジェクト指向言語のメソッドオーバーロードを扱うために、メソッド（関数）がどういう型の引数をとるかを文字列にエンコードすることが一般的に行われている。

  例えば、あるクラスで、同名のメソッドを 2 つ、次のように宣言した場合：

```
package APackage;
class AClass {
  int aMethod(int a, short b)  {...}
  int aMethod(double a, float b)  {...}
}
```

  NMI ではどちらも同じシンボルになってしまう。JNI では、それぞれ、次のシンボルとなる：

```
Java_AClass_aMethod__IS
Java_AClass_aMethod__DF
```

  GCJ では次のシンボルとなる。このエンコード法は、GCC の C++コンパイラ g++と同じである：

```
_ZN8APackage6AClass7aMethodEis
_ZN8APackage6AClass7aMethodEdf
```

- Java の基本型の、ヒープ上、スタック上、レジスタ上表現

  例えば `byte` 型であれば、オブジェクトのインスタンス変数や配列の場合は 1 バイト、スタックとレジスタ上では 1 ワード、というように決める必要があろう。

- オブジェクトのメモリ上表現

  C 言語の場合、構造体中でメンバがどのように整列されているかが、コンパイル済みオブジェクト間の相互運用性を左右する。オブジェクト指向言語の場合も、基本的に、オブジェクトのメモリ上表現はインスタンス変数を集めた構造体であると考えてよい。しかし、メンバ変数がどう整列されているかだけでなく、仮想呼び出しのための表（vtable）をどのように持つか、といった規約が処理系ごとに異なり（図 1）、これを決める必要がある。

- Java 仮想マシンの基本機能の呼び出し方

  C や FORTRAN のプログラムは、最低限、ローダなどがあれば実行できる。しかし、Java 言語のソースコードやクラスファイルは、GC を含めたメモリ管理機構、クラスローダや、Reflection API に必要なクラス情報管理機構があることを仮定している。これらはライブラリであると考えることもできるが、ここではランタイムと呼ぶことにする。

  仮に、GC はまったく行わず、クラスの実行時ロードも禁止するとしても、オブジェクトの生成、例外の throw、モニタ操作など、ランタイムが提供すべき基本機能は依然残る。当然ながら、こういった基本機能を呼び出すための API、ABI は、処理系ごとに独自のものがある。例えば、オブジェクトのメモリ領域確保は、Classic VM の NMI では `allocObject(...)` であり、GCJ では `JvAllocObject(...)` である。

  ただし、こういった基本機能の呼び出し方は、JNI がインタフェースを規定しているので、それに従っておくことで、バイナリ間の互換性を保てる可能性が高い。

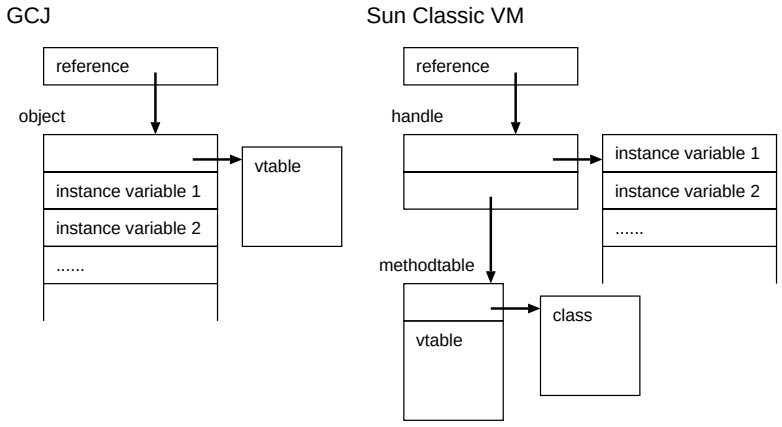

図 1: オブジェクトのメモリ上表現の例

既存の処理系に準拠するという選択

ABI をどう決めるかにもいくつか候補があり、やはりそれぞれの得失がある：

- 既存のものにならう。
  - Java 仮想マシンや AOT コンパイラ: GCJ、Classic VM の NMI、HotSpot VM、...
  - JNI
- 独自のものを設計する。

COINS プロジェクトの目的はコンパイラの開発なので、既存処理系の規約に準拠することの利点が大きい。その処理系のランタイムを流用できるため、開発対象をコンパイラに絞れるのである。

例えば GCJ に準拠することで、自然と、GCJ のライブラリ `libgcj` が提供する基本機能や、Boehm GC [19] を利用することになる。既存の Java 仮想マシンと組み合わせて実行するためには、JNI や NMI、またはその Java 仮想マシン内の ABI に準拠したネイティブメソッドを出力することとなろう。実際に、一部の商用 AOT コンパイラはその方針を採っている。また、Java 仮想マシンと AOT コンパイラを組み合わせている実例として、HP の Chai [20] がある。TurboChai という AOT コンパイラであらかじめ生成しておいたネイティブコードを、組み込み向け Java 仮想マシン ChaiVM から呼び出すことができるようになっている。

相互運用性を最も高くできるのは、JNI に準拠するという方針だろう。JNI は元々C/C++でメソッドを実装するための API だが、ABI としても機能するように設計されている（と筆者は分析している）。コンパイラ側では、メソッドを、JNI の呼び出し規約や mangling 方式に従ってコンパイルし、ランタイムの機能は JNI の API を通して呼び出すのである。こうすることで、コンパイラが生成するコードは、ほとんどの Java 仮想マシンと組み合わせて実行できるようになるだろう。GCJ もまた、JNI をサポートしている。

しかし JNI への準拠にも問題がある。相互運用性、移植性の代償として、性能上のペナルティがある。JNI の呼び出し規約や基本機能呼び出し API は、各 Java 仮想マシン、ランタイム本来のものとは食い違っていることが普通である。そのため、実行系本来のインタフェースと JNI の間を仲立ちするための wrapper 関数が用意される。コンパイラが生成したコードが JNI 経由で呼び

出されたり、JNI 経由でランタイムの機能を呼び出したりするたびに、wrapper 関数の呼び出しがオーバーヘッドとなるのである。さらに、オーバーヘッドはそれだけでは済まない。例えばメソッド呼び出しは、JNI では次の手順を踏む：

```
void Java_AClass_ANativeMethod(JNIEnv *env, jobject obj, ...) {
  // クラスオブジェクトを得る
  jclass klass = (*env)->FindClass(env, "java.lang.Object");
  // メソッド ID を得る
  jmethodID id = (*env)->GetMethodID(env, klass, "toString", "()Ljava/lang/String;");
  // 呼び出す
  jstring value = (*env)->CallObjectMethod(env, obj, id);
}
```

関数を 3 回呼び出さねばならない。その 3 関数も、ランタイム本来の機能を wrap しているものである。また、それぞれの関数のアドレスを得るためにポインタを辿らねばならない。関数のアドレスをキャッシュすることも考えられるが、いずれにせよ、メモリからアドレスをロードしての間接呼び出しになるので、わずかにオーバーヘッドはある。

Java 仮想マシンや AOT コンパイラからどれかひとつを選び、その ABI に従うという選択はどうだろうか。例えば、GCJ の CNI（Cygnus Native Interface for C++/JavaIntegration）や Sun 社 Classic VM の NMI などである。他にも Java 仮想マシンは数多くあり [21][22][23][24] それぞれが自身の ABI を持っていることだろう。JNI とは異なり、性能上のペナルティは小さくできる。しかし、生成コードが特定の実行系に依存してしまうという問題がある。また、将来 ABI が変更される恐れが JNI より高いだろう。追従するための手間がかかるかもしれない。

特定の実行系に準拠するとしたら、どれを選ぶのが得策だろうか。Classic VM は Java 2 SE（JDK） 1.4 以降には含まれなくなったので、Classic VM への依存は得策ではないだろう。GCJ への準拠がよいのではないかと考えている。次に挙げる利点がある：

- GCC は非常に広く使われている。開発が止まったり入手できなくなる恐れも小さい。
- AOT コンパイラなので生成されたコードの観察が容易である。GCJ が生成したコードを共通インフラストクラチャが生成したコードに置き換えることで公平な比較もできる。
- 品質も、日常使えるくらいに向上してきた。Java 実行系の標準的なベンチマーク SPEC JVM98 も動くらしい。
- 無料であり、ソースコードも手に入り、研究目的に使いやすい。他の AOT コンパイラの多くは売りものである。
- GCC の中では C++と Java の相互運用性が高いので（4.1 節）、共通インフラストラクチャも、C++に対応した場合に Java/C++を組み合わせやすくなる。

4 関連アクティビティ

4.1 GCJ

GCJ [18] は GCC の Java コンパイラである。Java 言語のソースコードとクラスファイルをアセンブリコードにコンパイルできる。ソースコードをクラスファイルにコンパイルすることもでき

る。C コンパイラ gcc コマンドと同様に、gcj コマンドで呼び出せる。また、gij コマンドでインタプリタも起動できる。

AOT コンパイラ（gcj）とインタプリタ（gij）はあるが、JIT コンパイラはまだない。インタプリタには最近 direct threading が実装されたが、デフォルトでは有効にはなっていない。

GC として、C/C++から利用できる保守的 GC ライブラリとして有名な Boehm GC [19] を採用している。その作者である Hans J. Boehm 氏も積極的に GCJ の開発に参加している。

GCC の中では Java と C++がよく統合されていることも、GCJ の特徴である。例えば、次のようなことが可能である：

- Java のオブジェクトを C++のオブジェクトとして扱う。
- Java の `java.lang.Integer#getInteger(String)` を C++から `java::lang::Integer::getInteger(jstring)` として呼び出す。
- Java で `throw` した例外を C++で `catch` する。逆も可能。

クラスファイルだけでなく、Java 言語のソースコードをコンパイルできるとはいえ、ループといったソースコード中の構造は活用していない（らしい）。中間表現が、共通インフラストラクチャで言えば LIR に相当する GCC の RTL だからである。ソースコードをコンパイルしたものとクラスファイルをコンパイルしたもののどちらが高い性能を発揮するかは、プログラムによって変わってくる。

参考文献

[1] The Sumatra Project, *Toba: A Java-to-C Translator*, `http://www.cs.arizona.edu/sumatra/toba/`.

[2] COMPOSE Project, *Harissa home-page*, `http://compose.labri.u-bordeaux.fr/prototypes/harissa/`.

[3] Keishiro Tanaka, *j2c/CafeBabe java .class to C translator*, `http://www.gimlay.org/ andoh/java/j2c/`.

[4] Tower Technology Corporation, *Tower Technology Home*, `http://www.towerj.com/`.

[5] Excelsior, LLC, *Excelsior JET*, `http://www.excelsior-usa.com/jet.html`.

[6] Jikes Project, *Jikes' Home*, `http://www.ibm.com/developerworks/oss/jikes/`.

[7] DMS - The Kopi Project, *The Kopi Project*, `http://www.dms.at/kopi/`.

[8] Jim Hugunin et al., *Jython Home Page*, `http://www.jython.org/`.

[9] Jan Arne Petersen et al., *JRuby*, `http://jruby.sf.net/`.

[10] Per Bothner, *Kawa, the Java-based Scheme system*, `http://www.gnu.org/software/kawa/`.

[11] Robert Tolksdorf, *Programming Languages for the Java Virtual Machine*, `http://grunge.cs.tu-berlin.de/ tolk/vmlanguages.html`.

[12] OpenJIT project, *OpenJIT: A Reflective JIT Compiler for Java*. http://www.openjit.org/.

[13] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, Y. Kimura, *OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java*, Proc. of 14th European Conference on Object-Oriented Programming (ECOOP '2000), June 2000.

[14] Intel Corporation, *Intel(R) C++ and Fortran Compilers Home Page*, http://developer.intel.com/software/products/compilers/.

[15] 丸山冬彦, 小川宏高, 松岡聡, *Java* **バイトコードをデコンパイルするための効果的なアルゴリズム**, 情報処理学会論文誌, Vol. 40, No. SIG10 (PRO 5), pp. 39–50, December 1999.

[16] Hanpeter van Vliet, *Mocha, the Java Decompiler*, http://www.brouhaha.com/ eric/computers/mocha.html.

[17] Pavel Kouznetsov, *Jad - the fast Java decompiler*, http://www.geocities.com/SiliconValley/Bridge/8617/jad.html.（現在このページはありません）

[18] The GCC Team, *The GNU Compiler for the Java Programming Language (GCJ)*, http://gcc.gnu.org/java/.

[19] Hans J. Boehm, *A garbage collector for C and C++*, http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[20] Hewlett-Packard Company, *Embedded Software from HP Chai*, http://www.hp.com/products1/embedded/.

[21] , Jim Pick et al., *Kaffe.org*, http://www.kaffe.org/.

[22] , Etienne M. Gagnon, *SableVM*, http://www.sablevm.org/.

[23] , John Leuner, Stephen Crawley, *kissme - A free Java Virtual Machine*, http://kissme.sf.net/.

[24] , Philip Fong et al., *The Aegis VM Project*, http://aegisvm.sf.net/.

A 更新の履歴

2002 年 7 月 4 日 最初の版。