オブジェクト指向言語によって記述された、携帯電話、PDA のアプリケーションプログラム圧縮方式

関口 龍郎 †‡　　大岩 寛 †　　米澤 明憲 †

{cocoa,oiwa,yonezawa}@is.s.u-tokyo.ac.jp
東京大学 理学部 情報科学科 †
科学技術振興事業団 さきがけ研究 21 機能と構成 ‡

1　はじめに

近年、IBM workpad, Sharp ザウルスなどの PDA に手軽にプログラムを転送し、実行する枠組が普及しはじめている。*i*-アプリと呼ばれる Java に基づく移動コードを実行する携帯電話も 2001 年 1 月より販売されている。

このような計算環境では CPU 性能、メモリ、ネットワークなどの資源が著しく制限されており、計算資源の豊かな環境でのプログラムの実行方式とは異なった工夫が必要になる。

この論文では、計算資源の制限された環境で動作させるためにプログラムのサイズを縮小する手法について述べる。この手法には次のような特徴がある。

- 他のほとんどの圧縮手法と組み合わせることができる。
- 解凍する必要がなく、圧縮したままで実行することができる。
- コードの転送量だけではなく、実行時のメモリ使用量も減らすことができる。

我々はこの手法を Java に適用しサンプルプログラムを圧縮してみたところ 1 ~ 2 割程度縮小させることができた。我々の手法では、既にコンパイルされた Java クラスファイルの集合を入力として受け取り、サイズを縮小したクラスファイルの集合を出力する。クラスファイルのフォーマットは標準的なものであり、特に拡張はなされていない。また本手法は Java に限られたものではなく、ほとんどのクラスベースのオブジェクト指向言語に適用することができる。

```
class   Point {
    int         x;
    int         y;
    int         GetX() { return x; }
    int         GetY() { return y; }
}
```

図 1: Point クラス.

```
class   Calculator {
    int         acc;
    int         arg;
    void        add() { acc = acc + arg; }
    void        mult() { acc = acc * arg; }
}
```

図 2: Calculator クラス.

2　例

この節では我々のコード圧縮方式について直観的な説明を行う。

図 1 はオブジェクト指向言語の説明の中で頻繁に

```
class   Poinlator {
    int         xcc;
    int         yrg;
    int         GetX() { return xcc; }
    int         GetY() { return yrg; }
    void        add() { xcc = xcc + yrg; }
    void        mult() { xcc = xcc * yrg; }
}
```

図 3: Poinlator クラス.

使われる Point クラスである。このクラスは二つの int 型の変数を instance variable として持ち、いくつかのメソッドが定義されている。一方、図 2 は Point クラスの場合ほど多くはないが、時々利用される [1] 例である。Calculator クラスは計算機、いわゆる電卓の動作を模倣するものであり、一つだけ整数を記憶することができ (acc)、その値に別の値 (arg) を加えたり、掛け合わせたりすることができる。

クラス Point とクラス Calculator は、全く別の目的のために設計されたクラスであり、通常は同じ文脈で使われることはないと予測することができる。しかし、この二つのクラスは instance variable の型が多重集合として偶然等しくなっている。我々のコード圧縮方式はこのような (1) 同じ文脈で使われていない (2) instance variable の型の多重集合が等しい、クラスを探しだし、それを融合するものである。融合の結果、図 3 のようなクラスが生成されることになる。

このようなクラス融合の結果、次のような利点がある。クラスの数が減るために

- クラス管理用のデータ構造が転送コード中から不要になる。
- クラス階層が単純になり、実行時のデータ構造、例えば method dispatch table などが小さくなる。

3 クラスの融合条件とその解析

任意のクラスを融合してしまえる訳ではない。この節ではクラス融合が可能な条件を述べ、クラス融合が可能なクラスを見付ける解析について簡単に説明する。

3.1 クラス融合可能条件

融合可能なクラスとは、直観的には決して混同されることのないクラスである。具体的には以下の条件を満たす時にクラスは融合可能である。

1. instance variable の型の多重集合が等しい。ただしここでは static variable は問題ではない。
2. 仮想関数呼び出しが起こる時にその receiver object の集合の中に同時に含まれない。
3. instanceof 述語によって区別することがない。
4. direct super class を共有しているか、またはどちらか一方が他方の direct super class であるか、またはどちらか一方の direct super class が java.lang.Object である。
5. direct super class の少なくとも一つのコンストラクタが致命的な副作用 (例えばウィンドウを生成する) を持たない。
6. コンストラクタ、静的メソッドを除く、すべてのメソッドが融合可能である。

ただしメソッドが融合可能であるとは以下のいずれかの条件を満たすことを言う。

1. 同じシグネチャ(名前、型) のメソッドはどちらか一方のクラスでのみ定義されている。
2. 同じシグネチャのメソッドは存在するが、どちらかのメソッドの名前を変更することができる。
3. 同じシグネチャのメソッドが存在し、名前の変更もできないが、投げられる例外が一致する。

ここで、メソッドの名前を変更できない場合とは、インタフェースを implements しているか、親クラスのメソッドを override している場合である。(サブクラスによって override されているメソッドは名前が変更できる。)

3.2 解析

条件 2 と条件 3 を除くクラス融合条件はクラス定義から簡単に判定することができる。条件 2 と条件 3 についてはデータフロー解析を行って可能な receiver object の集合を求める必要がある。ただし動的束縛があるので、通常のデータフロー解析とは異なり、解析の進行により、フローグラフに変化が生じる。そのためフローグラフの管理とデータフロー解析を同時に行う必要がある。このような解析については Oxhøj らが提案した手法 [5] があり、我々の実装した手法も原則として彼らのものと同じである。ただし Oxhøj らは context-sensitive なものであったが我々は context-insensitive なものである。また Oxhøj らは例外を利用したデータフローを考慮していないが、我々は Yi [7] の研究を踏まえ、例外についても考慮している。

我々はデータフロー解析を行う前にバイトコード検証を行い、その結果を利用してフローグラフを生成している。これにより動的束縛に関する部分を除いて、フローグラフの作成を 1 パスで行うことができる。バイトコード検証には萩谷らのアルゴリズム [4] を使用した。

この解析を行うためには

1. プログラム全体に含まれるクラスの集合が静的に列挙できる。
2. メソッド呼び出し地点において、実際に呼び出されるメソッドが引数のクラスだけから静的に決定できる。
3. ランタイムシステムから呼び出されるコールバックメソッドの引数として与えられる値の型は静的に決定できる。

という三つの前提が満たされていなければならない。そのためクラスローダーや reflection API を使用することでこの前提が崩れると、安全な解析を行うことができない。すべてのクラスが静的に列挙できるという条件は、クラスが動的にロードされる一般の Java 処理系では実現することが難しいが、KVM の場合では一つのアプリケーションが一つの JAR ファイルとして提供することが要請されており、容易に達成できる。

4 クラス融合アルゴリズム

この節ではクラス融合のアルゴリズムについて簡単に説明する。

クラス A と B が融合可能であるとする。クラス A の構造はそのまま残し、クラス B をクラス A に付加する。クラス B はその結果として消滅する。この時に以下のような方針でクラス A を書き換える。

- クラス B の static field をクラス A に付加する。もし同じ名前の static field があれば名前を変える。プログラムすべてのクラス B のその static field へのアクセスをクラス A の対応する static field に書き換える。
- クラス B の instance variable とクラス A の instance variable の対応を作る。この時にクラス A の親クラスの instance variable と対応することもあるので注意する。この対応に基づいてプログラムすべてのクラス B (およびそのサブクラス) の instance variable へのアクセスを書き換える。またこの時にクラス B のサブクラスの instance variable の名前が衝突する時は、それを書き換える。
- クラス B のコンストラクタをクラス A に付加する。Java ではコンストラクタへの呼び出しは、引数の型によって静的に決定される。したがってクラス B を生成するにはクラス A に埋め込まれたクラス B のコンストラクタを呼び出すようにする。この時にクラス A のコンス

	圧縮前			圧縮後			比率	
名前	クラス数	サイズ	JAR	クラス数	サイズ	JAR	サイズ	JAR
_200_check	18	40707	23327	14	32635	17924	80.2%	76.8%
_201_compress	22	76223	34953	18	68214	29120	89.5%	83.3%
_202_jess	161	455873	197565	111	294434	134835	64.6%	68.2%
_209_db	14	71166	31217	13	68701	27321	96.5%	87.5%
_213_javac	187	622473	289938	143	673596	254568	108.2%	87.8%
_222_mpegaudio	66	181192	89364	54	171346	76368	94.6%	85.4%
_227_mtrt	35	116384	53547	32	110093	45471	94.6%	84.9%
_228_jack	67	191899	96561	50	167552	72257	87.3%	74.8%

図 4: 圧縮効率.

トラクタに引数の型が同じものがあると区別できなくなるので、新しく付け加えるコンストラクタの引数の型を変更する。この変更には無害な引数の型 (byte, char, short, int) を追加することによって行われる。引数の型を一つまた複数追加した場合、それに応じて引数も追加する必要がある。これは 0 を渡すことによってなされる。このため我々のコード圧縮法ではサイズが増加する場合もありえる。

またクラス B のコンストラクタがクラス B の direct super class のコンストラクタの呼び出しを行っていた場合、それをクラス A の direct super class のコンストラクタの呼び出しに書き換える必要がある。これは、クラス A とクラス B の direct super class が同じだった場合には必要ないが、同じでない場合もありえるのでその時に必要である。ここでもしクラス A の direct super class のコンストラクタが致命的な副作用を持つ場合は、クラス融合自体が失敗する。

- クラス B のメソッドをクラス A に付加する。ここでもし同じ名前、シグネチャのメソッドがクラス A に元々存在するときにはどちらかの名前を変更する。もし変更できない時には、クラスの型を表す instance variable をクラス A に加え、メソッドの内部に入ってから分岐するコードを追加する。これもまたクラスのサイズが増大する要因となる。

- クラス B が implements しているインターフェースをクラス A に implements させる。

クラスが融合できるかどうかはそのクラスの direct super class に依存する。そのためクラスを融合した結果、新しい融合が可能になる時がある。したがって、クラス融合を行うごとに新しい融合が可能かどうか調査する必要がある。クラス融合は、融合可能なクラスがなくなるまで行われる。

5 圧縮効率の測定

この節では実際のプログラムを我々の手法で圧縮した結果について述べる。我々は現実のアプリケーションの例として SpecJVM98 を採用した。SpecJVM98 は PDA 上で実行するにはいささか大きすぎるアプリケーションであるが、以前述べたように PDA などで実行する標準的なアプリケーションがまだ用意されていないためやむを得ず SpecJVM98 を利用した。

図 4 にその結果を示す。これを見ると数個から数十個のクラスを削減することに成功し、その結果およそ 1 ~ 2 割のサイズを減少させることができた。ただし、この実験においては標準的なクラスライブラリ (java. で始まるパッケージ) のクラスは変換の

名前	融合数
_200_check	0
_201_compress	0
_202_jess	1
_209_db	0
_213_javac	52
_222_mpegaudio	1
_227_mtrt	0
_228_jack	17

図 5: 双方の direct super class が java.lang.Object でない融合.

対象ではない。アルゴリズムの節で述べたように我々のクラス融合方式にはサイズを増大させる要因もあり、_213_javac のサイズが増加したのはそのためであると考えられる。このような場合はクラス融合を取り止めることで元のサイズよりも悪くならないようにすることができる。しかし、我々の手法は多くの場合において有効であると言える。

Java においてクラスファイルを圧縮するのにしばしば使われている JAR によって圧縮した結果も図 4 に示す。これによると、我々の手法によって圧縮される比率は、JAR を使うかどうかに関わらずほぼ一定である。この事実は、我々の手法が圧縮のために利用する性質が、JAR が利用している性質とほぼ直交していることを示唆している。しかし _213_javac の場合は、クラスファイルのサイズは我々の手法によって増大しているにも関わらず、それを JAR によって圧縮すると逆に縮小するという結果になっている。この現象に関してはさらに分析が必要である。

6 議論

この論文ではクラス融合可能条件として二つのクラスの instance variable の型が多重集合として等しいこととしているが、吸収されるクラスの instance variable の型が吸収するクラスのそれの部分集合となっていても変換結果は型安全である。しかし、この場合は無駄なメモリの使用量を増大させるため、PDA 上での実行を想定する場合には吸収されるクラスの instance の数が少ないという保証が必要であると思われる。

我々は本手法によって実行時のメモリ使用量が減少することを主張しているが、実際の減少量は Java のコンパイル方式や実行環境に依存することとなる。しかし、既存の Java の実行環境のソースコードは公開されていない場合も多く、本手法の有効性を検証するのは難しい。本論分では、クラス数が少ないほどクラス管理のためのメモリ使用量は減少するという一般的な傾向を述べる以上の主張はできないが、何らかの有効性の検証法を確立する必要はある。

図 5 は融合した二つのクラスの direct super class が双方とも java.lang.Object でない場合の数を表している。この測定の目的は、ユーザが定義したクラスのサブクラス同士が融合する場合の数を調べることである。その結果_213_javac と _228_jack だけが突出して多いが、他ではほとんどないことがわかる。_213_javac と _228_jack に共通しているのはプログラミング言語の構文木をオブジェクト指向風にクラスを使って定義していることである。この結果は、綺麗なクラス継承木を定義しても実際には使わない場合が多いことを示している。

PDA 向けの Java の仕様として K virtual machine (KVM) と呼ばれるものが Sun Microsystems によって与えられており、NTT Docomo の *i*-アプリも KVM を多少拡張したものを用いている。しかし、我々の知る限り KVM に対する標準的なベンチマークは存在しないため、我々は圧縮の対象として SpecJVM98 のアプリケーションを利用した。また、我々の手法は KVM に特に依存したものではないが、クラスローダをユーザが提供できないことや reflection API の使用を制限されている KVM の性質を利用している。

7 関連研究

Java のクラスファイルを圧縮する研究として Pugh によるもの [6] がある。これは本研究とは異なり、クラスの意味は変更せずに、Java のクラスファイルのデータ構造の性質を利用して圧縮を行うものである。

春名ら [8] は Java のクラスファイルにおいて外部参照が文字列であることに注目し、リンカーに相当するものを Java に与えることによって、クラスファイルのサイズを減少させている。この方式ではクラスファイルのフォーマットは拡張されている。

実行時のコードサイズを縮小させる研究としては Cooper らによるもの [3] がある。この方式の基本的なアイデアはコードの中から同じ命令列を探し出して、その部分をサブルーチンにすることで共有するものである。

クラスの継承関係のみから、仮想関数と宣言された呼び出しが実際には静的に解決できることを分析し、インライニングなどの最適化を行うことは良く行われている [2]。

8 今後の課題

現在の実装では、解析の部分は完全に自動化されているが、クラス融合のアルゴリズムの一部に人間が判断しなければならない部分がある。例えばコンストラクタが致命的な副作用を持つかどうか判断する部分である。将来的にはクラス融合を完全に自動化したい。

我々は、クラスを融合する時に、それらのクラスがしばしば全く同じ命令列を含むメソッドを持っていることを発見した。そのようなメソッドを共有することで更に圧縮効率を高められるのではないかと思う。

参考文献

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *ECOOP'96 Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, July 1996.

[3] Keith D. Cooper and Nathaniel McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, volume 33(5), pages 139–149, 1998.

[4] Masami Hagiya and Akihiko Tozawa. On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines. In *Static Analysis, 5th International Symposium , SAS'98*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32, September 1998.

[5] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In *Proceedings of ECOOP'92, Sixth European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349, July 1992.

[6] William Pugh. Compressing Java Class Files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 247–258, May 1999.

[7] Kwangkeun Yi and Byeong-Mo Chang. Exception Analysis for Java. In *ECOOP'99 Workshop on Formal Techniques for Java Programs*, volume 1743 of *Lecture Notes in Computer Science*, June 1999.

[8] 春名修介, 金丸智一, 吉田力, 和気裕之, and 富 永宣輝. 家電向け仮想マシンアーキテクチャ. 情報処理学会プログラミング研究会, October 1998.