Vol. 41 No. 2 情報処理学会論文誌 Feb. 2000

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

丸山冬彦† 小川宏高† 松岡聡†

機械語命令列から同じ意味のソースプログラムを復元するデコンパイルという技術は古くから知られており，主に，リバースエンジニアリングのための手段の一つとして利用されてきた．実際に，Java とそのバイトコードに関しても，いくつかの処理系が提案されているが，これまで提供されてきた処理系では，Java 言語には無い goto を挿入するなど，Java 言語の文法を逸脱した結果を出力することがある．また，デコンパイルのアルゴリズムがアドホックで，応用の利かないものであるため，我々の OpenJIT コンパイラが要求するような，任意のバイトコードから正しいソース構造を復元するデコンパイラフロントエンドとして用いることができない．そこで，我々は Java バイトコードから適切な Java 言語の制御構造を復元するための効果的なアルゴリズムを新しく考案した．アルゴリズムの基本となる考え方は，メソッドのコントロールフローグラフに対するドミネータツリーを用いるものである．これはブロック構造が完全な入れ子になる言語の場合，制御構造を表す任意のプログラム片はドミネータツリーにおいて，ただ一つのサブツリーをなすという性質に基づいている．この一般性により，アルゴリズムは Java 以外の言語に適用することも可能である．OpenJIT での予備的な実装による評価では，他のデコンパイラが制御構造の復元に失敗するプログラムであっても，我々のアルゴリズムは適切にそれを復元し，かつ，実行速度は同程度であることを示した．

An Effective Decompilation Algorithm for Java Bytecodes

Fuyuhiko Maruyama,† Hirotaka Ogawa† and Satoshi Matsuoka†

The technique called decompilation that reads sequences of machine code and generates the corresponding source program has been known for some time, and utilized primarily for reverse-engineering. For Java and its bytecode, although there have been several proposals of decompilers, most generate outputs that are inappropriately extend the Java language, such as insertion of gotos not present in Java. Moreover, the decompilation algorithms are somewhat ad-hoc and difficult to extend of verify its applicability, which is a hindrance to our our OpenJIT compiler which requires a decompiler frontend to recover the correct source structure from arbitrary bytecode. Instead, we have devised an new and effective alghorithm for decompilation, with emphasis on properly recovering control structures. The key idea is to base the algorithm around the dominator tree of the control flow graph of a method. This is based on the observation that, for a properly-nested block-structured language, each part of program representing a control structure corresponds to just a single subtree in the dominator tree. As such, the algorithm is general enough to be applied to other languges besides Java. The evaluation of our preliminary implementation in OpenJIT show that our algorithm properly recovers control structures where other exisiting decompilers fail, and with relatively equivalent execution speeds.

1. はじめに

コンパイラの生成した機械語列から同じ意味を表すソースプログラムを復元する，デコンパイルという技術は古くから知られており，主にリバースエンジニアリングの一手法として利用されてきた．近年では Java 言語 [4] とその実行環境である Java 仮想機械 [11] (JVM) のバイトコードに対してデコンパイル技術を適用するデコンパイラがいくつか発表されている．

一方，現在我々が研究開発を進めている OpenJIT コンパイラシステム [6] では，仮想機械によるプログラムの実行を高速化するために用いられる Just-In-Time (JIT) コンパイラに自己反映計算の概念を導入した開放型 JIT コンパイラを実現し，主に「性能の可搬性」[3] の実現や DSM[10] に代表される並列プログラミ

† 東京工業大学 情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

ングスタイルの Java への高可搬な適用などを目指している．OpenJIT では，バイトコードより高度な中間表現上のプログラム変換の支援，および，一般的なコンパイラの技術[1]の JIT コンパイラへの適用を目的として，Java 言語の抽象構文木を用いており，そのために，Java バイトコードのデコンパイラを必要とする．しかし，従来のデコンパイラは，ディスアセンブラ程度の解析しか行わ(え)ないものや，制御構造を十分に復元できないものであり，我々の目的には適わない．

そこで，Java バイトコードをデコンパイルするための新たなアルゴリズムを考案し，それに基づいた OpenJIT 用のデコンパイラモジュールを開発した．本アルゴリズムはプログラム解析におけるドミネータツリーと Java のソースコードの対応に着目した新しいものであり，従来のものと比較してより正確な復元が行える，アドホックでないなどの特徴を持つ．

2. 準　　備

本論文では，一般的なコンパイラ技術で用いられる用語を使用する．より包括的，かつ，正確な定義は，コンパイラの解説書[1]に譲るとして，本節では，提案するアルゴリズムの説明のために最も重要なドミネータツリーについて，簡単に説明する．

フローグラフの，先頭ノードからノード B へ至る全てのフローがノード A を通るとき，ノード A はノード B をドミネートするという．このとき，ノード A をノード B のドミネータという．これが「ドミネートする」と「ドミネータ」の定義である．この定義を基に，コントロールフローグラフに含まれるノード間の階層的なドミネート関係をツリーとして表現したものが，ドミネータツリーである．

図 1 の例で説明すると，左のコントロールフローグラフにおいて，先頭ノードであるノード 1 からノード 2 へ至る全てのフローはノード 1 を通るので，ノード 1 はノード 2 をドミネートする．また，先頭ノードからノード 6 へ至る全てのフローはノード 2 を通るので，ノード 2 はノード 6 をドミネートする．図 1 の左のコントロールフローグラフの全てのノードについて，ノード 1，ノード 2 およびノード 6 について説明したようなドミネートする/される関係をツリーとして表したものが右のドミネータツリーである．

3. デコンパイラの構成

多くのオブジェクト指向言語の場合と同様，Java のプログラムは複数のメソッドから構成される．しかし，

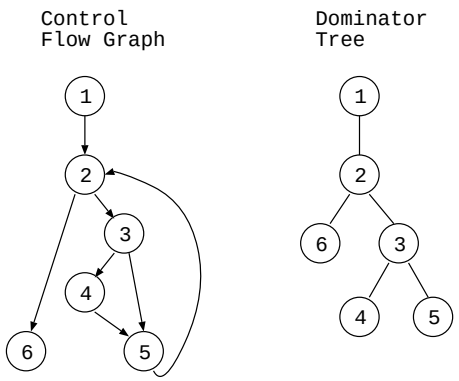


図 1　コントロールフローグラフとドミネータツリー
Fig. 1　control flow graph and dominator tree

クラスファイルの定義により各メソッドが明確に分離されているため，デコンパイル処理はメソッド単位で行うことができる．

そこで，我々のデコンパイラはメソッドのバイトコードを入力とし，抽象構文木を出力するものとして，以下の手順で処理を行う．

(1) メソッドのバイトコードを JVM の命令列に分解し，同時にベーシックブロックのリーダとなるべき命令に印をつける．
(2) ベーシックブロックのリストとしてコントロールフローグラフを構築する．
(3) コントロールフローグラフに対応するドミネータツリーを構築する．
(4) ベーシックブロック毎のシンボリック実行☆により，ベーシックブロックをまたがない(部分)式や文を復元する．
(5) 複数のベーシックブロックと条件分岐により実現される，Java 言語の&&や||のような演算子による式や条件式を表すコントロールフローを見つけ出し，それらの式を復元する．
(6) 制御構造を復元する．
(7) 結果を抽象構文木として出力する．

これらの内 (1) から (5) および (7) は単純であるか，既知の手法を用いて実現できる．そこで本稿では (6) の「制御構造の復元」を主題とする．

4. 制御構造の復元

我々は本論文において，効率的に制御構造を復元する新しいアルゴリズムとして，直接コントロールフ

☆ バイトコード命令を後置記法の式と見なし，計算を行うかわりに式を生成することにより，式の抽象構文木が得られる[9]．また，この手順をシンボリック実行と呼ぶ．

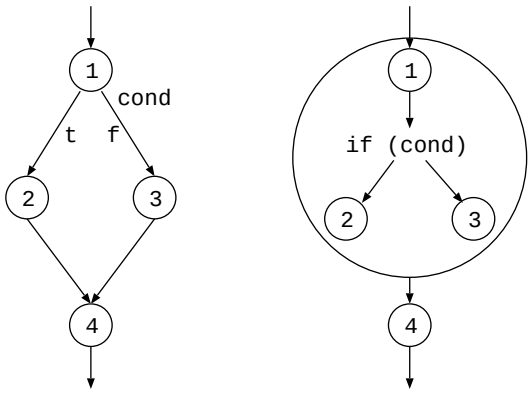


図 2 if 文の (部分) コントロールフローグラフ
Fig. 2 (partial) control flow graph of if statement

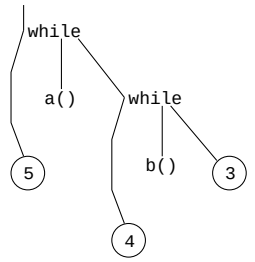


図 4 while 文を含む抽象構文木の例
Fig. 4 Small example of AST contains while statement

ローグラフを解析する代りに，対応するドミネータツリーを用いる方式を提案する．

Java バイトコードのデコンパイルを実現する際に問題となるのは，Java 言語の持つ制御構造の内，JVM の持つ命令を用いて表されている if-else 文，do-while 文，while 文，for 文，switch 文，break 文，continue 文の構造の復元である．本節ではそのためのアルゴリズムを提案する．以降，単に制御構造という場合，これらの構造を指すものとする．また，Java 言語の持つ制御構造としては他に try-catch 文がある．しかし，この構造はバイトコードとは独立した情報を用いて複数のプログラム片を組み合わせることにより表現されている．つまり，try-catch 文の構造は，クラスファイルの定義によりはじめから構造化されている．よって，本アルゴリズムでは取り扱わない．

4.1 既存の手法

制御構造を復元するため従来から知られている方針は，それぞれ弱点を持っている．単純な方針では，バイトコード (命令列) のパターンマッチングを用い，用意したパターンの数に応じた複雑さのプログラムを復元できる．しかし，ある程度複雑な構造を持つバイトコードを復元するためには，この方針は実際的ではなく，いわゆる「構造化されたコントロールフロー」の復元が必須である．

構造化されたコントロールフローの復元のための基本的な考え方は，コントロールフローグラフの部分グラフを単純化していくことである．例えば，図 **2** の左の部分グラフは，if-else 文を表すグラフであり，これを右のように作り替える．これにより，元は条件分岐を含む複数のノードからなっていた部分グラフを，if-else 文を内包する一つのノードと見なし，全体のグラフを直線的なグラフと解釈することができるようになる．また，Tarjan のアルゴリズム [8] として，この作り替えの規則を正規表現を用いて表すことにより，一般的に取り扱う方法も提案されているが，入れ子になったループ構造を取り扱うことが難しい☆．

他にも，Krakatoa[12] では，Ramshaw の提案した Pascal プログラムから goto を取り除くアルゴリズム [5] を拡張して Java プログラムに適用する手法が提案されている．Krakatoa では，まず Ramshaw のアルゴリズムを用いて，バイトコードを文法的には正しいが，不自然な Java プログラムへと変換し，その後幾つかの規則を用いることにより，自然なプログラムへと再変換するという手順で制御構造を復元する (図 **3**)．この手法は有力だが，アルゴリズムが複雑で，かつ，一度不自然な形からパターンマッチを行っているので，潜在的に効率が悪い．

4.2 我々のアルゴリズムの基礎

ドミネータツリーをコントロールフロー解析に適用可能な理由は，Java 言語の抽象構文木とそれを実現するバイトコードに対するドミネータツリーの類似性による．例えば，図 **4** は，次の Java プログラムに対する (簡略化した) 抽象構文木であり，同じプログラムに対応するドミネータツリー (図 **5**) を比較すると，両者がほぼ対応していることが判明する．

```
while (a()) {
  while (b())
    block3;
  block4;
}
block5;
```

より正確に述べれば，Java 言語の制御構造に対応するプログラム片は，ドミネータツリーにおいて，必ず一つのサブツリーをなすという性質がある．これ

☆ Java 言語にはラベル付きの break 文や continue 文があるため，潜在的に無限通りの規則を必要とする．

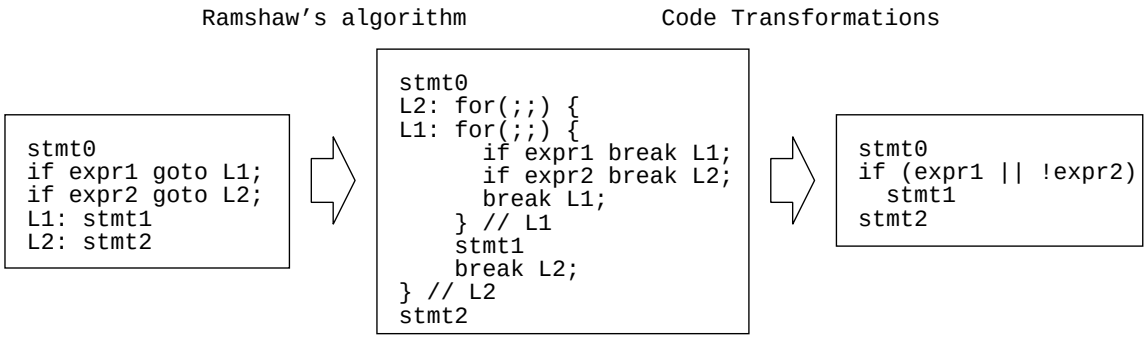


図 3 Krakatoa の方式
Fig. 3 Algorithm of Krakatoa

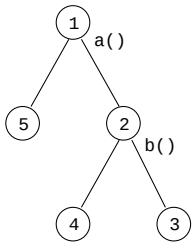


図 5 while 文を含むプログラムに対するドミネータツリーの例
Fig. 5 Small example of dominator tree for program contains while statement

は，制御構造の表す意味に由来する性質であり，例えば，`while(cond)block` という制御構造は，(1)`cond` が false なら終了し，(2)true なら `block` を実行し，(3) (1) からを繰り返す，ことを意味する．この場合，`block` が実行されるのは `cond` が true の場合のみであり，これは，`cond` を評価する (1) の処理が (2) と (3) の処理をドミネートすることを表す．よって，この制御構造全体はドミネータツリーにおいて，必ず，(1) の処理を表すノード以下の一つのサブツリーをなすことになる．while 以外の制御構造についても同様の性質を持つことは明らかである．この性質から図 4 と図 5 のような類似性が現れる．我々のアルゴリズムはこの性質に着目し，ドミネータツリーを主な対象としてコントロールフロー解析を行う．

この場合，重要となるのは，両者の類似性よりも相違点である．抽象構文木の場合，Java 言語の持つ制御構造が制御の流れを定義しているのに対し，ドミネータツリーでは定義されていない．例えば，図 4 では，while という制御構造を表すノードが制御の流れを定義しているのに対し，図 5 ではそれが定義されておらず，その代わりに，対応するコントロールフローグラフにおいて，構造化されていない条件分岐や $4 \to 1$ や $3 \to 2$ といったバックエッジにより，同じ制御構造が表現されている．

我々のアルゴリズムでは，コントロールフローグラフのエッジとして与えられる制御の流れを，等価な制御構造を用いて表現し直すことにより，コントロールフロー解析を実現する．つまり，図 5 の例において，ノード 1 やノード 2 が while ループのヘッダであることやそれぞれのループ本体がノード 2 及びノード 3 を根とするサブツリーであることを識別することにより，ドミネータツリーから抽象構文木を生成する．

4.3 アルゴリズムの概要

我々のアルゴリズムでは，次のように幾つかの段階により，ドミネータツリーから抽象構文木を生成する．

(1) ヘッダ (制御構造の先頭となるノード) に印をつける．ループ構造 (for 文，while 文，do-while 文) の先頭ノードはバックエッジの行き先であることにより判定でき，switch 文の先頭ノードは JVM の tableswitch 命令あるいは lookupswitch 命令の存在で判定できる．条件分岐を持つノードについては，if 文の実現やループ構造の終了判定に現れるため制御構造の先頭となり得るが，この段階ではこれらのどちらであるかを判定できないため，この判定はアルゴリズムの後の段階まで保留する．条件分岐は条件式や論理式の評価のためにも現れるが，それらはこのアルゴリズムの適用前までに取り除いておくことができる．

(2) 前段階で印をつけたヘッダのそれぞれについて，ツリーのルートに近いものから，その構造の中身を表すサブツリーの先頭ノードと，構造から出た直後に制御の移るべき後続ノードを判別し，ヘッダとその中身，そして後続ノードの関係を基に，Java 言語の持つ制御構造の中から適切なものを選択する．具体的には，構造の中身がある場合はその先頭ノードはヘッダに直接ドミネートされているという事実を用いて判別する．

後続ノードについては，ヘッダにドミネートされている場合とされていない場合があり，次のように判別する．

(a) ヘッダ以下のサブツリーの各ノードの持つ全てのエッジについて，行き先ノードがヘッダにドミネートされていないものを集める．

(b) (a) で集めたエッジの内，break 文や continue 文では到達できないものを選別し，もし条件を満たすノードがあれば，そのノードを後続ノードとする．これにより switch 文に現れる，いわゆる fall through な制御の流れを復元する．break 文や continue 文で到達できるノードであるかどうかは，ドミネータツリーにおいて現在注目しているヘッダよりもルートに近い任意の別の構造 (上位の構造) を順に参照することにより判定する．上位の構造は現在注目している構造よりも先に Java 言語の制御構造として復元されるので，あるノードが break 文または continue 文の行き先ノードとして適切かどうかを判定することができる．

(c) (b) で決定できない場合，後続ノードはヘッダにドミネートされている中から選ぶ．switch 文の構造の場合，各 case ブロックからのフローが合流するノードを後続ノードとする．一方，loop 構造の場合，最初にループ内ノードを識別し，次にループ内ノードからループ外ノードへ向かうエッジを集め，更にそれらのエッジの中から合流するエッジを選別する．そのようなエッジが存在する場合，合流するノードを後続ノードとする．

(d) (c) で決定できない場合，後続ノードは (c) において合流するエッジの選別前に集めたエッジの行き先の中から任意に選ぶことができる．それらのノードを後続ノード候補として，次の規則で決定する．

- ヘッダのベーシックブロックがループ本体の先頭ノードと後続ノード候補内のノードへの条件分岐のみを持つ場合，その後続ノード候補を後続ノードとする．
- ヘッダへのバックエッジと後続ノード候補内のノードへの条件分岐を持つ場合，そのノードを後続ノードとする．
- 前の二つの条件を満たすノードがない場合，後続ノードはないものとする．

次に，現在注目しているヘッダについて，その中身の先頭ノードと後続ノードの関係を基に，Java 言語の持つ制御構造の中から適切なものを選択する．switch ヘッダの場合 switch 文を選び，ループ構造の場合は次の規則を用いて決定する．

- ヘッダのベーシックブロックがループ本体の先頭ノードと後続ノードへの分岐だけである場合，for 文あるいは while 文で表す．この時，バックエッジが一つだけで，かつ，バックエッジを持つノードの中身が Java 言語の式で表せる場合には for 文を選び，バックエッジを持つノードの中身を for 文の update 式とする．それ以外の場合は while 文を選ぶ．while 文の場合，continue 文はバックエッジとして表されるので，バックエッジが複数ある場合は while 文としなければならない．一方，バックエッジが一つだけである場合，潜在的には for 文と while 文のいずれかであるが，仮に continue 文が存在する場合，while 文では矛盾が生じるのに対し，for 文では矛盾とならないため，ここでは for 文とする．
- バックエッジを持つノードがループヘッダと後続ノードへの条件分岐である場合，do-while 文で表す．この時，その分岐条件を do-while の条件とする．
- 前の二つの条件のいずれも満たさない場合，ループの継続条件を true とし，ループ本体の先頭ノードが現在注目しているループヘッダとなるような for 文あるいは while 文で表す．この時，一つ目の条件と同様にバックエッジが一つで，かつ，バックエッジを持つノードが Java 言語の式で表せる場合には for 文を選び，バックエッジを持つノードを for 文の update 式とする．それ以外の場合は while 文とする．

以上の規則を用いて制御構造を選択することにより，ループ内部のノードの持つエッジを必要に応じて continue 文で正しく実現できることを保証する．

(3) 残された条件分岐は全て if 文を表す条件分岐であるから，それらを if 構造を表す抽象構文木表現に作り替える．

(4) この段階で，Java 言語の制御構造を用いて表されていない制御の流れに相当するエッジは break 文あるいは continue 文のものであるから，それらで表す．

(5) 以上の結果を用いて，抽象構文木を出力する．

4.4 アルゴリズムの特徴

本アルゴリズムの計算量は，ドミネータツリーの全探索 ($O(n)$) で見つけた制御構造のヘッダのそれぞれについて，ヘッダ以下のサブツリーの全探索 ($O(n)$) で後続ノード，構造内部の先頭ノードを見つけるため，全体としての計算量は，$O(n^2)$ となる．しかし，これは最悪の場合 (コントロールフローグラフの全てのノードが入れ子となった制御構造を表すプログラム，すなわち，制御構造が n 重にネストされたプログラムの場合) であり，現実のプログラムに適用する場合には，$O(n)$ に近いものと考えられる．

本アルゴリズムでデコンパイル可能なバイトコードは，for 文，while 文および do-while 文がループを表すために使用されている限りにおいて，Java 言語によるソースプログラムからコンパイラによって生成された全てのバイトコードと考えられるが，より理論的，かつ，厳密な議論に関しては，将来の課題とする．

4.5 アルゴリズムの適用例

本節では，実際にアルゴリズムを適用する手順を説明する．例として，次のプログラムを用いる．

```
void daily() {
 lifeCycle:
  for (int i = 0; i < 2; i++) {
    gotoLab();
    do {
      if (emergency())
        break lifeCycle;
      if (tired())
        haveABreak();
      else
        studyHard();
    } while (!hungry());
    if (veryPoor())
      continue;
    gotoRestaurant();
  }
  return;
}
```

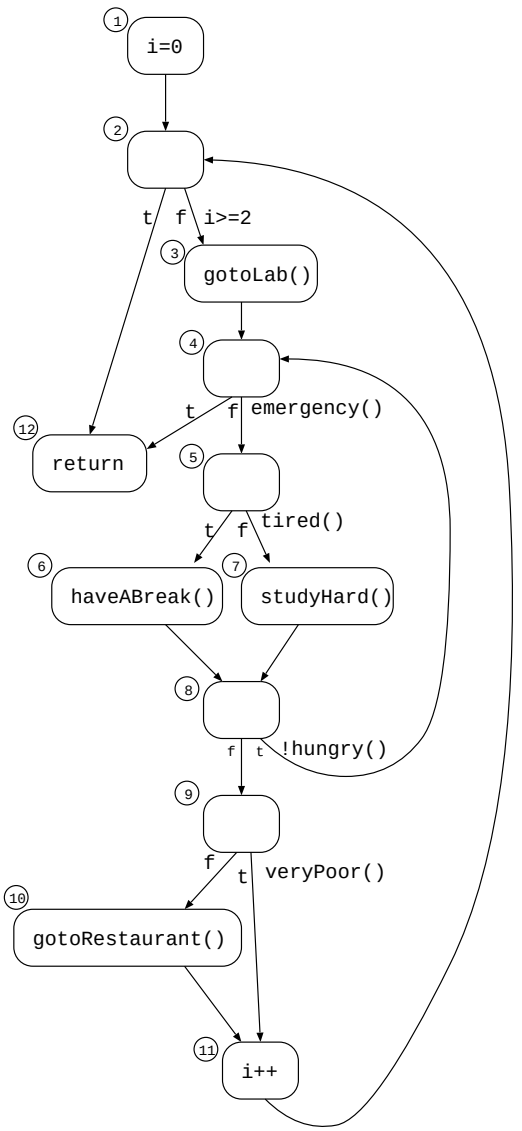


図 6 例で用いるプログラムのコントロールフローグラフ
Fig. 6 Control flow graph of the sample program

このメソッドから得られるバイトコードに対応するコントロールフローグラフは図 6 である．図 6 における枠線内の記述はそのノード (ベーシックブロック) の内容 (抽象構文木表現されたプログラム片) を表しており，条件分岐を持つノードについては右下に分岐条件が記されている．また，各ノードの左上には便宜的に番号が付けられており，以降ではこの番号でノードを識別するものとする．アルゴリズムを適用する前に対応するドミネータツリーも作っておく．

まず，アルゴリズムの第一段階 (前節の (1) に対応) として，ヘッダに印をつける．ドミネータツリー (図 7) の各ノードを探索し，バックエッジを持つノー

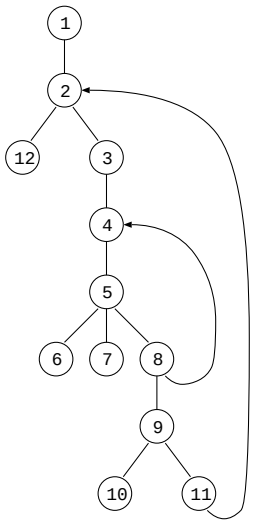


図 7 図 6 のドミネータツリーと
バックエッジの存在により見つかるループヘッダ
Fig. 7 The dominator tree of Fig. 6 and loop headers found by existence of back-edges

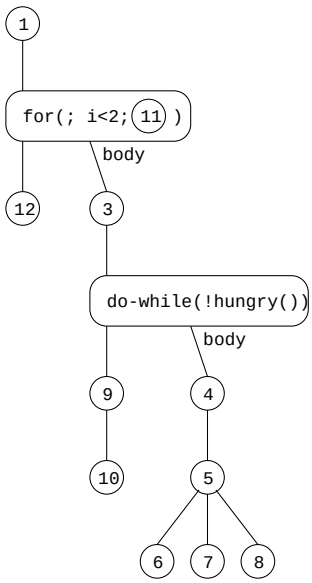


図 8 ループ構造導入後のツリー
Fig. 8 The tree after introduction of loop structures

ドを訪れたときに，その行き先にループヘッダであることを記す．この例の場合，ノード 11 でノード 2 に印をつけ，ノード 8 でノード 4 に印をつける．結局，ノード 2 とノード 4 がループヘッダで，それぞれに至るバックエッジは一つずつであることがわかる．

次に第二段階として，ループヘッダであるノード 2 とノード 4 について，本体の先頭ノードと後続ノードを見つけ，Java 言語による制御構造を選択する．

ノード 2 の場合，ドミネータツリーの子ノード☆ (直接ドミネートしているノード) であるノード 3 とノード 12 の内，ノード 3 だけがノード 2 へのバックエッジを持つノード 11 をドミネートしているので，ノード 3 がループの本体の先頭ノードである．また，後続ノードはノード 12 と決まる．また，この構造には for 文を選択する．この際，ループの終了条件をノード 2 の分岐条件とし，その update 式はノード 11 の中身とする．

ノード 4 の場合，子ノードはノード 5 だけであり，このノードがノード 8 をドミネートしているので，ノード 5 がループ本体の先頭ノードである．こちらの構造の場合，ノード 4 の唯一の子ノードがループ本体に含まれているため，後続ノードはあるとすればノード 5 以下のサブツリーに含まれている．このサブツリーを探索すると，ノード 8 の持つノード 9 へのエッジが，ループ外へのエッジであることがわかる．そこで，ノード 9 を後続ノードとする．また，この構造には do-while 文を選択し，ループの継続条件はノード 8 の分岐条件とする．この場合，ループヘッダはループ本体に含まれることに注意が必要である．この時点での途中結果は図 **8** となる．

第三段階では，ツリーを探索し，残された条件分岐を if 文のヘッダであると見なし，その構造を if 文で表す．ノード 5 の分岐条件は tired() であり，ノード 5 は二つの行き先ノード 6 とノード 7 をドミネートするので if-else 文とする．ノード 9 の分岐条件は veryPoor() である．行き先の一つはノード 10 であるが，もう一つの行き先であるノード 11 (この時点では for 文の update 式を表すノードとなっている) に合流するので，if 文とする．ノード 4 の分岐条件は，emergency() である．ノード 4 は二つの行き先ノード 5 とノード 12 の内，ノード 5 だけをドミネートするので，「`if(!emergency())` ノード 5」のようにしても良いが，ノード 4 からノード 12 へのエッジは何らかの制御構造により (この場合は break 文) で実現されると仮定できるので，ここでは，「`if(emergency())` ノード 12」としておく．この辺りの選択は実装に委ねられている．この時点での途中結果は図 **9** となる．なお，この図において，点線による円で囲まれたノード 12 は，ベーシックブロックの中身をこの位置に挿入するという意味ではなく，ノード 12 へのエッジを表すために便宜的に表記したものであり，アルゴリズ

☆ ループの本体はあるとすれば必ずヘッダに直接ドミネートされている．

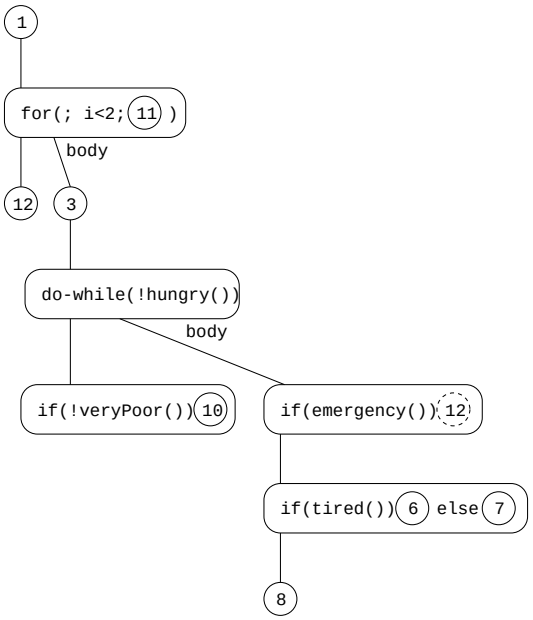


図 9 if 文復元後のツリー
Fig. 9 The tree after recovery of if statement

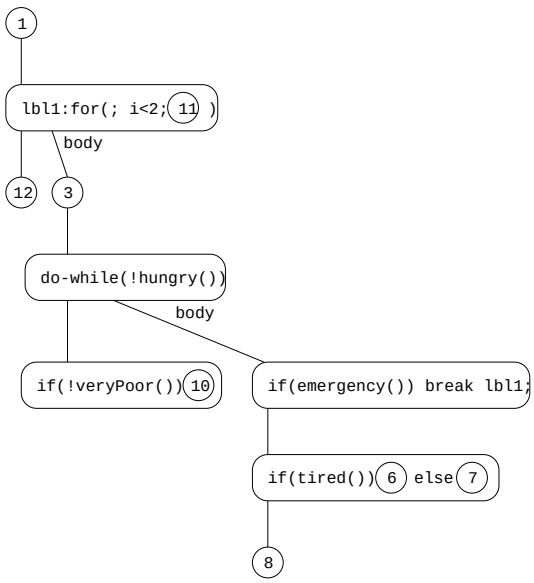


図 10 break 文復元後のツリー
Fig. 10 The tree after recovery of break statement

ムの次の段階で適切に処理される．

第四段階では，ツリーを探索し break 文あるいは continue 文を用いて表すべきエッジを持つノードを探す．この例では，「`if(emergency())` ノード 12」のノード 12 へのエッジを処理する．この際，ノード 12 は，「`for(;i<2;` ノード 11)」の後続ノードであるから，for 文にラベル lbl1 をつけ，「`break lbl1`」で表す．この時点で，結果は図 **10** となる．

第五段階は，抽象構文木の出力である．各ベーシックブロックの内容と図 10 を参照すると，図 **11** のような抽象構文木を簡単に生成することができる．この抽象構文木は，元のプログラムと同じ意味を持つので，これでデコンパイルが正しく行えたことになる．

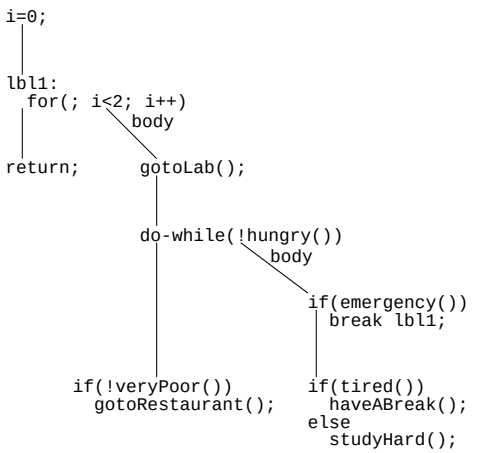


図 11 アルゴリズム適用の結果として得られる抽象構文木
Fig. 11 The AST as a result of application of our algorithm

5. 評　　価

本研究の提案するアルゴリズムを Java 言語により実装したデコンパイラを用い，以下の環境において簡単な評価を行った．なお，既存の Java 用デコンパイラとしてやはり Java 言語で記述された☆mocha-b1[2] を用い，比較対象とした．

- Sun Ultra60 (UltraSparc2 300MHz×2, 256MB)
- 日本語 Solaris 2.6
- JDK-1.1.7B と付属の Java 仮想マシン

5.1 解析結果の比較

最初は，種々のソースプログラムをコンパイルした結果として得られるクラスファイルを我々のデコンパイラと mocha-b1 の両者でデコンパイルし，その結果を比較した．以下にその一例を示す☆☆．

```
public class Test {
  void test() {
  outer:
```

☆ Java 以外の言語で記述された可能性もあるが，少なくとも通常の JVM で実行可能なクラスファイル群として配布されている．

☆☆ 参考までに，両者によるデコンパイル結果の出力を付録 A.1 及び A.2 として添付する．

```
    for (int i = 0; i < 100; i++) {
      switch (i) {
      case 10:
        for (int j = 0; j < i; j++) {
          System.out.print(j);
          if (j == 8) continue outer;
        }
      case 11:
        System.out.println("4");
        break;
      default:
        System.out.println(i);
        break;
      }
      System.out.println("difficult!");
    }
  }
}
```

このプログラムにおいてコントロールフロー解析上の問題となるのは内側の for 文に含まれるラベル付きの continue 文と，switch 文の一つ目の case ブロックが二つ目の case ブロックにつながっている箇所である．

我々のデコンパイラは，このプログラムから得られるバイトコードを正しく処理し，制御構造を適切に復元することができた．一方，mocha-b1 では，コントロールフロー解析をあきらめ，Java 言語にはない goto を含む結果を出力した．

この結果は，我々のアルゴリズムが mocha-b1 の採用しているコントロールフロー解析のアルゴリズムに比べて，より忠実に制御構造を復元できるということを示している．本例だけでなく，これまでの検証では mocha-b1 に復元できて我々のデコンパイラでは復元できないようなプログラムの例は発見されていない．更に，mocha-b1 では goto が生成される他のプログラムについても，我々のデコンパイラでは復元できる．

現在のアルゴリズムで，復元できない制御構造の例としては，Ramshaw が提案しているような，goto を表すだけのために使用される場合の制御構造 (while あるいは for) があげられる．失敗する理由は，この場合の while あるいは for 構造はループを作らないため，バックエッジの存在を利用して制御構造を識別することができないことにある．現在の我々のアルゴリズムでは，失敗は，後続ノードが一意に決定できないという現象として現れるため，この現象を利用してループを作らないような while あるいは for 構造を復元できるようにアルゴリズムを改良することは，将来の課題の一つである．

5.2 速度の比較

速度を比較するために，我々の実装と mocha-b1 とで同等の結果を得られる次のプログラムを 100 回デコンパイルするのにかかる時間を測定した．

```
public class Exam0 {
  void test(int x) {
    while (x < 100) {
      System.out.println("Hello!");
      if (jammed())
        continue;
      if (interrupted())
        break;
      System.out.println("World!");
    }
    System.out.println("bye");
  }
  boolean interrupted() { return false; }
  boolean jammed() { return false; }
}
```

Solaris の time コマンドを用いて計測した時間を表 1 に示す．この結果は，ユーザ時間およびシステム時間のどちらについても mocha-b1 の方が高速であることを示している．しかし，有意な差はシステム時間に対してであるため，この差はコントロールフロー解析の速度を表しているとはいえない．システム時間の大半は I/O 時間だと思われる；この内我々のデコンパイラは mocha-b1 に比べ，使用するクラス数が多いため，その読み込み時間に差が生じている可能性があるが，これは本デコンパイラが OpenJIT の一部として拡張可能なクラスフレームワークとなっているのに対し，mocha はそのようになっていないという，実装上の選択によるものである．一方，デコンパイル目的のクラスファイルの入力部分や結果の出力部分については，同程度の速度が得られるように改善可能である．

ユーザ時間に関しては，残念ながら mocha-b1 がソースプログラムを含め技術的な詳細が明らかにされていないため，両者の差を検討することは難しい．そこで，我々のアルゴリズムに関してのみ，より詳細な測定を行いこれを検討する．

現在の我々のデコンパイラは，221 個のクラス，1460 個のメソッドから構成されており，これら全てをデコンパイルするのに必要な時間をプロファイラ☆を用いて計測した．なお，予備的な計測により，デコンパイル結果の出力にかかる時間が合計としては非常に大きく，かつ，数多くのメソッドに分散されているために，

☆ JDK 付属の java コマンドで-prof オプションを用いて実行した．

表 1 デコンパイルの処理時間
Table 1 Timings for decompilation

時間	我々のデコンパイラ	mocha-b1
real	0m8.223s	0m8.862s
user	0m3.230s	0m3.060s
sys	0m1.020s	0m0.220s

表 2 デコンパイラの各部分にかかる時間
Table 2 Timings for parts of decompilation

部分	時間 (ms)
デコンパイラ全体	28090
クラスファイルの読み込み	15913
デコンパイル処理全体	11747
アルゴリズム適用前	10068
バイトコードの解析	1068
シンボリック実行	6516
&&と||の復元	1745
アルゴリズム全体	1453
第一段階	356
第二段階	517
第三段階	243
第四段階	305

直接その結果を考察することは難しいと判明したので，表示等に関わる部分を全て取り除いたデコンパイラを用いた．計測の結果，我々のデコンパイラの各部分で費やされている処理時間は表 2 のようになっており，その中で本稿で提案したアルゴリズム (コントロールフロー解析) の実行にかかる時間は，デコンパイル全体の約 12%に過ぎない．このことは，我々のアルゴリズムを用いて従来のデコンパイラよりもより良く制御構造を復元したとしても，それがデコンパイル処理全体の低速化には直接つながらないことを示している．同時にこの結果は，デコンパイル処理の高速化のためには，本論文で提案したアルゴリズム適用以前の部分の処理，特に，シンボリック実行の処理時間の短縮が必須であることも示している．現在の実装では，この部分の処理において小さなオブジェクトの生成破棄を繰り返しており，これが処理時間の増大を引き起こす最も大きな理由である．これに関しては，HotSpot をはじめとする新世代の JVM での最適化により解消すると期待され，それらを用いた更なる評価は今後の課題の一つである．

6. 関 連 研 究

Proebsting 等の Krakatoa[12] は，Ramshaw[5] のアルゴリズムを適用した Java 用のデコンパイラであり，制御構造の復元だけでなく，ベーシックブロック毎の式の復元に関しても詳しい．しかし，Krakatoa の用いる制御構造を復元するアルゴリズムは複雑で，かつ，一度不自然な形からパターンマッチを行っているので，潜在的に効率が悪い．

既存の Java 用デコンパイラとしては，mocha，jasmine，jad，GuavaD などがあげられるが，残念ながらどれも技術的な詳細は不明である．

Agesen の Pep[7] は，Self 仮想機械上で Java バイトコードを実行するシステムであり，初期の実装では if 構造の復元など簡単なコントロールフロー解析を行っていた．しかし，Java バイトコードを，制御構造として closure を用いた Self のバイトコードに変換することを目的する解析であるから，元の Java 言語の制御構造とは異なり，かつ，比較的低レベルな構造への変換を行うのみだった．

7. ま と め

本論文では，Java バイトコードのデコンパイルのために必要な制御構造の復元のための新たな手法として，ドミネータツリーを用いるアルゴリズムを提案した．このアルゴリズムでは，ドミネータツリーと抽象構文木の類似性を用いることにより，比較的単純な処理の積み重ねにより，効果的に復元を完了することができる．予備的な評価の結果から，我々の提案したアルゴリズムを用いることによって，既存の Java 用デコンパイラでは復元不可能だったプログラムを適切にデコンパイルでき，アルゴリズムの有効性が確かめられた．更に，このアルゴリズムの適用自体がデコンパイル処理全体の処理速度に与える影響は小さいこともわかった．また，提案したアルゴリズム自体は，Java バイトコードに限らず，いわゆる構造化言語一般に応用可能であると期待できる．

今後の課題として第一にあげられるのは，OpenJIT への応用を含む，更なる実装の完成である．なぜなら，現在のデコンパイラの実装は評価を目的とする予備的なもので，特に実行速度の面で改良の余地がある．一方で前節で述べたように，新しい JVM を用いることにより，デコンパイラの性能上の特性が変わってくることも考えられるため，この点に関する評価も必要である．

また，デコンパイラをより多くのプログラムに適用し，アルゴリズムの有効性を確認する必要もある．特に，高度に最適化されたバイトコードに対する振る舞いを調査し，アルゴリズムの改良を目指すことは重要な課題である．

アルゴリズムが有効に働いているかどうかを判定するためには，デコンパイル結果が元のバイトコードと同じ意味であることを確認する必要があるのだが，現

在は，自動化されていない．これは，コントロールフローグラフにおいてエッジで表されている制御の流れを，制御構造を復元する段階で，Java の制御構造による表現として正しく表せていることを確認することで自動化できる．そのような実装は，OpenJIT への応用も含め，多量で未知の Java プログラムへのデコンパイラの適用には必須であり，重要な課題である．

OpenJIT への応用については，デコンパイルというコストの高い処理が Java プログラムの実行性能に与える影響を考慮しながら，必要に応じて選択的にデコンパイルを行うという方針も視野に入れた，更なる評価が必要である．

その他，抽象構文木ではなくソースコードを生成する実装も課題としてあげられる．

謝辞 本研究は，情報処理振興事業協会（IPA）の高度情報化支援ソフトウェア育成事業において，テーマ名「自己反映計算に基づく Java 言語用の開放型 Just-in-Time コンパイラ OpenJIT の研究開発」として実施された．

参 考 文 献

1) A. V. Aho and R.Sethi and J.D. Ullman: *Compilers, principles, techniques, and tools.*, Addison-Wesley (1986).
2) Hanpeter van Vliet: Mocha, the Java Decompiler. `http://www.brouhaha.com/~eric/computers/mocha.html`.
3) 伊藤茂雄, 松岡聡: 複数の Java 処理系における高性能計算の性能評価にむけて, 情報処理学会研究報告 (HOKKE'99), Vol. 99, No. 21, pp. 25–30 (1999).
4) James Gosling and Bill Joy and Guy Steel: *The Java Language Specification*, The Java Series, Addison-Wesley (1996).
5) Lyle Ramshaw: Eliminating go to's while preserving program structure, *Journal of the ACM*, Vol. 35, No. 4, pp. 893–920 (1988).
6) Matsuoka, S., Ogawa, H., Shimura, K., Kimura, Y., Hotta, K. and Takagi, H.: OpenJIT–A Reflective Java JIT Compiler – Short Version for the OOPSLA'98 Reflection Workshop–, *OOPSLA'98 Reflection Workshop* (1998).
7) Ole Agesen: Design and Implementation of Pep, a Java Just-In-Time Translator, *Theory and Practice of Object Systems*, Vol. 3, No. 2, pp. 127–155 (1997).
8) R.E. Tarjan: Fast algorithms for solving Path problems., *Journal of the ACM*, Vol. 28, No. 3, pp. 591–642 (1981).
9) 佐々政孝: プログラミング言語処理系, 岩波講座ソフトウェア科学 5, 岩波書店 (1989).
10) 早田恭彦, 小川宏高, 松岡聡: OpenC++のリフレクション機能を用いた分散共有メモリの実現, 情報処理学会論文誌, Vol. 40, No. 2, pp. 13–22 (1999).
11) Tim Lindholm and Frank Yellin: *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley (1996).
12) Todd Proebsting and Scott Watterson: Krakatoa: Decompilation in Java, *COOTS '97* (1997).

付 録

A.1 我々のデコンパイラの出力

我々のデコンパイラは，抽象構文木を生成するように実装されており，結果として得られる抽象構文木は約 100 個のクラス群により定義された Java オブジェクトとして生成される．以下の出力は，それぞれの抽象構文木オブジェクトの持つデバッグ用出力を流用したものであるため，通常の Java 言語のプログラムとは異なった形式をしている．

```
public synchronized
class Test extends java.lang.Object {
  void test() {
    (= lv1#0 0);
  lbl0:
    for (; (< lv1#0 100) ;
         (= lv1#0 (+ lv1#0 1))) {
      switch (lv1#0) {
      case 10:
        (= lv2#0 0);
        for (; (< lv2#0 lv1#0);
             (= lv2#0 (+ lv2#0 1))) {
          (method
             (((java#0.lang).System).out)
             print lv2#0);
          if (== lv2#0 8) continue lbl0;
        }
      case 11:
        (method
          (((java#0.lang).System).out)
          println "4");
        break;
      default:
        (method
          (((java#0.lang).System).out)
          println lv1#0);
        break;
      }
      (method
        (((java#0.lang).System).out)
        println "difficult!");
    }
    return;
```

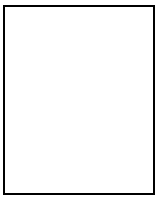
```
  }
  public void <init>() {
    (method super <init>);
    return;
  }
}
```

A.2 mocha-b1 の出力

mocha ではコントロールフロー解析に失敗すると，以下の例のように Java 言語には存在しない goto を含んだ結果を出力する．goto の後に付けられている数字は，バイトコードにおける行き先のアドレスを表しているようだ．

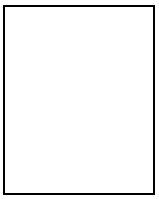
```
/* Decompiled by Mocha from Test.class */
/* Originally compiled from Test.java */

import java.io.PrintStream;

public synchronized class Test
{
    void test()
    {
        int i;
        int j;
        i = 0;
        expression i
        switch
case 10: goto 28
case 11: goto 57
default: goto 68
        j = 0;
        System.out.print(j);
        continue;
        if (j != 8) goto 39 else 41;
        continue;
        j++;
        if (j < i) goto 33 else 57;
        System.out.println("4");
        System.out.println(i);
        System.out.println("difficult!");
        i++;
        if (i < 100) goto 5 else 92;
    }
    public Test()
    {
    }
}
```

(平成 1999 年 5 月 28 日受付)
(平成 1999 年 7 月 19 日採録)



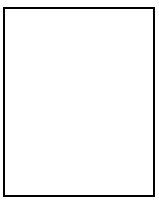
丸山　冬彦

昭和 46 年生．平成 11 年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻修士課程在学中．オブジェクト指向言語，並列・分散システム，広域分散システムなどに興味を持つ．



小川　宏高（正会員）

昭和 46 年生．平成 6 年東京大学工学部計数工学科卒業．平成 8 年同大学大学院工学系研究科情報工学専攻修了．平成 10 年度同博士課程中退．現在，東京工業大学大学院情報理工学研究科数理・計算科学専攻助手．プログラミング言語処理系，オブジェクト指向技術，並列計算機アーキテクチャ，広域分散システムに興味を持つ．ACM 会員．



松岡　　聡（正会員）

昭和 38 年生．昭和 61 年東京大学理学部情報科学科卒業．平成元年同大学大学院博士課程中退．同大学情報科学科助手，情報工学専攻講師を経て，平成 8 年より東京工業大学情報理工学研究科数理・計算科学専攻助教授．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インタフェースソフトウェアなどの研究に従事．現在進行中の代表的プロジェクトは，世界規模の高性能計算環境を構築する Ninf プロジェクト，計算環境に適合・最適化を目指す Java 言語の開放型 Just-In-Time コンパイラ OpenJIT，制約ベースの TRIP ユーザ・インタフェースなど．並列自己反映型オブジェクト指向言語 ABCL/R2 の研究で 1996 年度情報処理学会論文賞受賞．1997 年はオブジェクト指向の国際学会 ECOOP'97 のプログラム委員長を努める．ソフトウェア科学会，ACM，IEEE-CS 各会員．