JAVA バイトコードにおける データ依存解析手法の提案と実現

A Method of Data Dependence Analysis of JAVA Bytecode

誉田 謙二 † Kenji KONDA
大畑 文明 † Fumiaki OHATA
井上 克郎 †‡ Katsuro INOUE

† 大阪大学 大学院基礎工学研究科
Graduate School of Engineering Science, Osaka University
‡ 奈良先端科学技術大学院大学 情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology

概要

プログラムの依存関係解析は，プログラム理解，テスト，デバッグなど，ソフトウェアの開発から保守に至るまでその利用範囲は広く，様々な言語に対する解析手法が提案されてきた．しかし，その多くは通常の手続型言語を対象としたものであり，バイトコードのようなスタックマシンを前提としたプログラムの解析法の提案は少ない．本稿ではJAVA バイトコードにおけるデータ依存関係を定義し，その解析手法の提案および手法の実現を行った．

1 まえがき

プログラム依存関係解析は，コンパイラ最適化のバックエンドとしてだけでなく，プログラム理解，プログラムデバッグ (保守) など，その利用範囲は大きい．これまで提案されている依存関係解析手法は，高級言語のソースコードや 3 番地コードを対象としてきた [1]．しかし，**Java** バイトコード (*Java Bytecode* ，以下，バイトコード) [3] は，スタックマシン上での実行が前提となっており，従来手法を適用しただけではスタックを介したデータ依存関係を考慮することはできない．また，バイトコードにおける依存関係に関する研究としては [4, 5] があるが，依存関係の定義が述べられているだけで，実際の抽出手法については知られていない．

本研究では，バイトコードにおける単一スレッド内のデータ依存関係を定義し，その抽出手法を提案する．また，提案手法に基づき，バイトコードを入力とするデータ依存解析ツールを試作し，アルゴリズムを実際に動作させてみた．

以降，2. でバイトコードおよびそこに存在するデータ依存関係について説明する．3. でデータ依存解析手法の提案およびその実現について述べる．最後に 4. でまとめと今後の課題について述べる．

2 バイトコードとデータ依存関係

2.1 バイトコード

JAVA で書かれたソースプログラムは，JAVA コンパイラによりクラスファイルに変換される．クラスファイルはクラス単位に生成され，対応するクラスに関連したデータおよびバイトコードの命令列を含む．そして，**Java** バーチャルマシン (*Java Virtual Machine* ，以下，JAVAVM) がクラスファイルを読み込み，バイトコードを解釈，実行する．

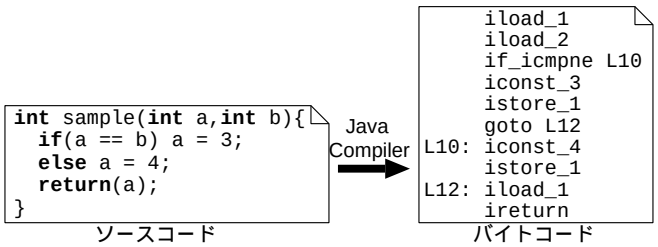

図 1: バイトコードへの変換例

例えば図 1 のように，メソッド`sample` はコンパイラによりソースコードからバイトコードに変換される．バイトコードはおよそ 200 種類の命令から構成されており，表 1 に代表的な命令とその動作を挙げる．またバイトコードにはデータを格納可能な変数として，フィールド，ローカル変数，スタックの 3 種類がある．

2.2 データ依存関係

命令 s で定義されたローカル変数 v の値が命令 t で参照されるとき，s から t に v に関するデータ依存関係 (*Data Dependence Relation*) が存在するという．図 2 では，`istore_1` によりスタックトップの値 x がローカル変数 1 に代入 (定義) され，`iload_1` によりローカル変数 1 の値がスタックトップにプッシュ (参照) されている．このとき，`istore_1` と`iload_1` の間にローカル変数 1 に関

表 1: バイトコードの代表的な命令

命令	動作
pop	スタックトップからデータを取り除く
iconst_n	定数 n をスタックに積む
iload_n	ローカル変数 n の値をスタックに積む
istore_n	スタックトップからデータを取り除き， それをローカル変数 n に格納する
getfield	フィールドの値をスタックに積む
putfield	スタックトップからデータを取り除き， それをフィールドに格納する
iadd	スタックトップからデータを 2 つ取り除き， それらの加算結果をスタックに積む
idiv	スタックトップからデータを 2 つ取り除き， それらの除算結果をスタックに積む
if_icmpne Ln	スタックトップからデータを 2 つ取り除き， それらが等しくなければ ラベル Ln に制御移動する

するデータ依存関係が存在する．

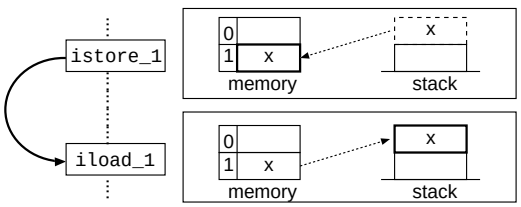

図 2: ローカル変数に関するデータ依存関係

命令 s で定義されたフィールド v の値が命令 t で参照されるとき，s から t に v に関するデータ依存関係が存在するという．図 3 では，putfield Point/x I によりスタックトップの値 y がフィールド x に代入 (定義) され，getfield Point/x I によりフィールド x の値がスタックトップにプッシュ (参照) されている．このとき，putfield Point/x I と getfield Point/x I の間にフィールド x に関するデータ依存関係が存在する．

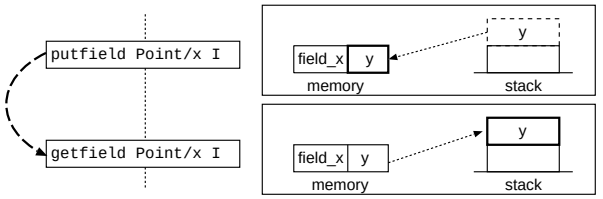

図 3: フィールドに関するデータ依存関係

また，命令 s でスタック上に積まれた値が命令 t で参照されるとき，s から t にスタックに関するデータ依存関係が存在するという．図 4 では，iconst_3 によりスタックにプッシュ (定義) された値が istore_1 によって参照されるため，iconst_3 と istore_1 の間にスタックに関するデータ依存関係が存在する．

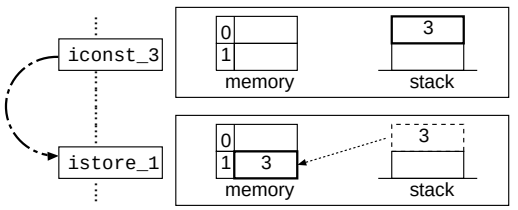

図 4: スタックに関するデータ依存関係

3 データ依存解析手法

本稿で提案するバイトコードに対するデータ依存解析手法は，以下の 2 つのステップで構成される．

Step1 制御フローグラフの構築:

制御フローグラフ (*Control Flow Graph* ，以下，*CFG*) とは，バイトコードに対して制御フロー解析を行い，各命令を節点，命令間の制御の移動を有向辺で表現したものである．バイトコードに対する CFG 構築は，従来の手続き型プログラムに対する手法 [1] を適用することが可能であり，図 5 にアルゴリズムの概略を示す．また，このアルゴリズムを図 1 のバイトコードに適用した結果を図 6 に示す．

アルゴリズム ConstructCFG
入力 バイトコード
出力 CFG
処理 バイトコードに対し CFG を構築する
(1) 各命令を抽出し，CFG 節点を作成 ($\mathcal{N}$: 全節点の集合)
(2) **foreach** n **in** $\mathcal{N}$ **begin**
(3) n の次に実行され得る CFG 節点の集合 $\mathcal{N}'$ を導出
(4) **foreach** n' **in** $\mathcal{N}'$
(5) n から n' に CFG 辺を引く
(6) **end**

図 5: アルゴリズム ConstructCFG

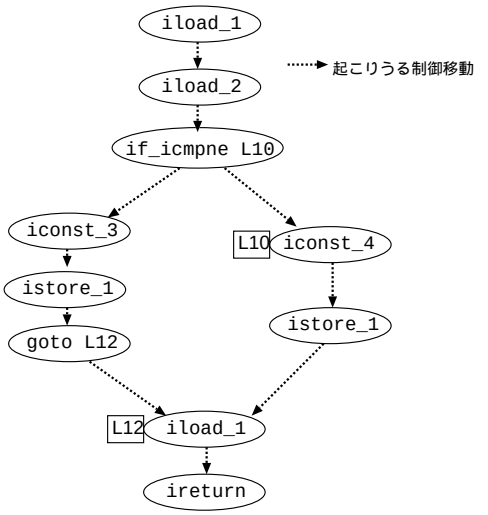

図 6: 図 1 の CFG

Step2 データ依存関係の抽出:

はじめに，データ依存解析に用いる解析フレームを準備する (Phase1)．次に，解析フレームを更新しながら Step1 で構築した CFG の辺をたどることにより，データ依存関係の抽出および CFG へのデータ依存辺の追加を行う (Phase2)．解析は実行可能性のある全ての経路に対して行い，新たなデータ依存関係が抽出されなくなるまで続ける．以下，各フェーズを詳細に述べる．

Phase1 解析フレームの準備

データ依存関係の抽出を行うにあたり，各メソッド内のフィールドやローカル変数，スタック上の各値がどの命令で定義されたかを保持する表 (解析フレーム (*Analysis Frame*) と呼ぶ) を用意する．解析フレームは，変数名およびその値を最後に定義した命令の行番号の対の集合である．図 7 は，バイトコードおよびその 1 ~ 8 行目の命令を解析し終えた状態での解析フレームを示している．フィールド 0 は未定義，ローカル変数 0 は 4 行目の`istore_0` で，stack_2 は 5 行目の`iload_0` で，stack_1 は 8 行目の`idiv` によって定義されたことを表している．stack_0 に関しては，7 行目の`iload_0` により値が定義されるが，8 行目の`idiv` により値が取り除かれるため，未定義となる．

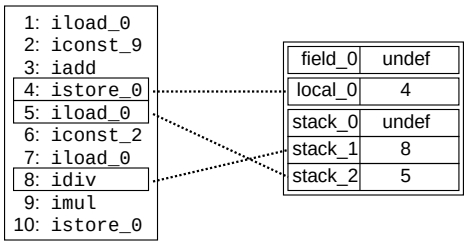

図 7: 解析フレーム

JavaVM の仕様により，特定の分岐から派生した分岐経路が合流する地点でのスタックサイズは経路に依存せず等しい [3]．また，各メソッドの処理に必要なローカル変数の個数や最大スタックサイズはコンパイラによりあらかじめ計算されているため，解析フレームの大きさは静的に決定することができる．

Phase2 データ依存関係の抽出

データ依存関係の抽出は，Phase1 で用意した解析フレームおよび，プログラムの開始節点を入力として，図 8 に示すアルゴリズム AnalyzeDD に従って行われる．このアルゴリズムは再帰的に適用される．

アルゴリズム AnalyzeDD の (8) では，命令の種類に応じて以下の 4 つの処理を行う．ただし (iii) および (iv) は，

```
アルゴリズムAnalyzeDD
  入力 節点 n，解析フレーム f
  出力 データ依存辺が追加された制御フローグラフ
  処理 解析フレームを用いてデータ依存辺を追加する
 (1) if n が CFG の合流点 then begin
 (2)    if f が n の解析フレーム履歴に含まれている then
 (3)       return
 (4)    else
 (5)       n の解析フレーム履歴に f を追加
 (6)    endif
 (7) endif
 (8) f を更新し、データ依存辺を CFG に追加
 (9) n の次に実行され得る CFG 節点の集合 N' を導出
(10) foreach n' in N' begin
(11)    f の複製 f' を作成
(12)    AnalyzeDD(n', f')
(13) end
```

図 8: アルゴリズム AnalyzeDD

(i) や (ii) の処理とともに行われる．

(i) 命令 s で変数 x が参照されるとき
解析フレームの x に対応するエントリが命令 t であるとき，命令 t から命令 s にデータ依存関係が存在することが分かり，対応する CFG 節点間に x に関するデータ依存辺を追加する．

(ii) 命令 s で変数 x が定義されるとき
解析フレームの x に対応するエントリを命令 s の行番号に変更する．

(iii) `iconst_n` や`iload_n` など，スタックへのプッシュを伴う命令のとき
スタックトップを記憶するポインタ sp の値を 1 つ下げる

(iv) `iadd` や`idiv` など，スタックからのポップを伴う命令のとき
解析フレームのスタックトップの変数のエントリを未定義にするとともに sp の値を 1 つ上げる．

図 9 において，`istore_0` は`iconst_6` で定義されたデータをスタックからポップし，ローカル変数 0 を定義している．そのため，これらの命令間はデータ依存関係が存在し，対応する CFG 節点間に辺が追加される．加えて，解析フレームのローカル変数 0 に対応するエントリが`istore_0` の行番号である 3 に，スタックトップに対応するエントリが未定義として更新される．

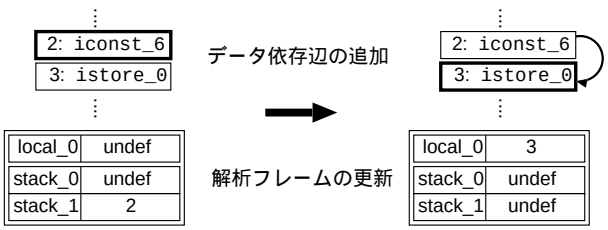

図 9: 解析フレームの更新，データ依存辺の追加

AnalyzeDD の (1) ~ (7) は，CFG にループがある際に解析が無限に続いてしまうのを回避する働きをしている．CFG の合流節点は，AnalyzeDD が適用される毎にその節点での解析フレームを履歴として保持する．このとき履歴に同一のフレームが存在しないかを調べ，存在すればその解析フレームによる解析を停止する．先にも述べたが解析フレームはコンパイル時にその大きさが決定可能であり，またそのエントリの値は行番号の最大値で抑えられるため，AnalyzeDD は必ず停止する．

図 10 に，図 1 のバイトコードに対する解析結果を示す．図 6 の CFG に対して，ローカル変数，スタックに関するデータ依存関係を表すデータ依存辺が追加されている．

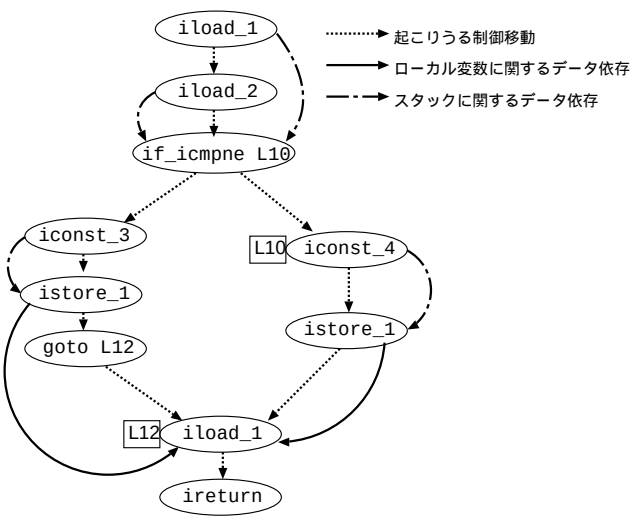

図 10: データ依存解析結果 (データ依存辺を追加した CFG)

3.1 提案手法の実現

本節では，3. で提案したデータ依存解析手法の実現について述べる．

図 11 にツールの構成を示す．バイトコードは通常 2 進形式のデータ列であるため，あらかじめ読みやすい Jasmin 形式 [2] に変換しておく．本ツールは，はじめに Jasmin 形式のバイトコードを構文解析し，CFG を構築する．その後，ローカル変数，スタックに関するデータ依存関係を抽出し，CFG にデータ依存辺を追加する．これによりデータ依存辺が追加された CFG が得られる．

実際に本ツールを用いてバイトコードのデータ依存解析を行い，提案手法が正しく動作することを確認した．本ツールでは，メソッド呼び出し命令がある場合，単にメソッド呼び出し引数を参照される変数，戻り値を定義される変数として扱うため，解析結果の正確性はメソッド間の解析を行った場合より低下する．ただし，抽出すべきデータ依存関係が排除されることはない．また，複数スレッド間にまたがる変数参照や変数定義は解析の対象としていない．

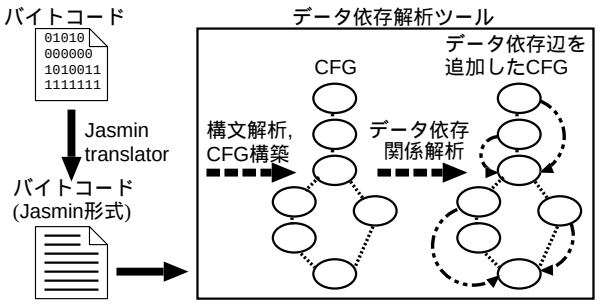

図 11: ツール構成

4 まとめと今後の課題

本研究では，バイトコードに対して，JavaVM の特徴であるスタックを考慮したデータ依存関係を定義し，その抽出手法を提案した．さらに，提案手法のプロトタイプ実装を行いアルゴリズムの動作を確認した．

今後の課題としては，提案した解析手法の効率化や，複数スレッド間におけるデータ依存関係の抽出が挙げられる．また，メソッド間のデータ依存関係の抽出を行うことでプログラム依存グラフ (*Program Dependence Graph* , *PDG*) を構築し，プログラムスライス (*Program Slice*) [6] 抽出などへの発展を考えている．

参考文献

[1] A.V.Aho, R.Sethi, J.D.Ullman: "Compilers Principles, Techniques, and Tools," Addison-Wesley, (1986).

[2] "Jasmin Home Page," `http://www.cat.nyu.edu/meyer/jasmin/`

[3] J.Meyer, T.Downing: "Java Virtual Machine," O'Reilly & Associates, (1997).

[4] J.Zhao: "Dependence Analysis of Java Bytecode," 日本ソフトウェア科学会第 *16* 回大会, pp.317-320(1999).

[5] J.Zhao: "Analyzing Control Flow in Java Bytecode," 日本ソフトウェア科学会第 *16* 回大会, pp.313-316(1999).

[6] M.Weiser: "Program Slicing," IEEE Transaction on Software Engineering, 10(4), pp.352–357(1984).