社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－SE－152 (5)
2006／5／17

Javaプログラム理解支援のための不変性解析

水野 良太† 桑原 寛明† 山本 晋一郎‡ 阿草 清滋†

†名古屋大学大学院情報科学研究科
‡愛知県立大学情報科学部

本稿では，Javaプログラムの理解支援を目的として参照の不変性解析手法を提案する．不変な参照は，その参照が指すオブジェクトの状態を変更する操作が適用されないため，オブジェクトの状態変化を追跡する際に着目する必要がない．本研究では，クラス，インタフェース，メソッド，メソッドの参照型の戻り値，インナクラスのコンストラクタ，参照変数といったJavaプログラムの構成要素間に不変性依存関係を定義する．不変性依存関係は，ある要素の不変性が別の要素の不変性に依存することを表す．不変性依存関係をたどることで不変性解析を行い，不変な参照変数を特定する．Javaプログラムに対し不変性解析を行い，不変な要素とそうでない要素を色分けして示すソースコードブラウザを実装した．実行例を示して本手法の有用性を評価する．

Immutability Analysis for Java Program Understanding

Ryota Mizuno† Hiroaki Kuwabara† Shinichiro Yamamoto‡ Kiyoshi Agusa†

†Graduate School of Information Science, Nagoya University
‡ Faculty of Information Science and Technology, Aichi Prefectural University

This paper proposes a method of immutability analysis to help understanding Java program. Immutable references can be ignored when we trace the transition of the state of objects, because it is impossible to change the state of object through them. In this research, we define an immutability dependency relation between the constructs of Java program: class, interface, method, return value which is of reference type, constructor of inner class, and reference variable. An immutability dependency relation expresses that immutability of constructs depends on immutability of other constructs. We find immutable reference variables, and methods by analyzing immutability following immutability dependency relations. We implement source code browser which shows immutable and mutable elements by different color. We show the usability of our method using a simple example.

1 はじめに

近年，ソフトウェアの巨大化は著しく，開発の効率化が望まれている．コードの再利用をスムーズに行うことができれば，生産性を向上させることができる．既存のソフトウェアを再利用する場合，プログラマは機能を拡張したり，不要な機能を省くなど，ソフトウェアを改変する必要がある．ソフトウェアの改変を行うために，プログラマは対象のソフトウェアの動作を理解する必要がある．また，発見されたバグへの対応や機能の拡張など，ソフトウェアの保守を行う際にも，対象のソフトウェアの動作についての理解が要求される．

オブジェクト指向プログラムの動作を理解するためには，プログラム内に出現するオブジェクトの状態変化を追跡する必要がある．しかし，一般にプログラム中には多数のオブジェクトが出現するため，オブジェクト全ての状態変化を追跡することは困難である．実際には，プログラム中には実行中に状態が変更されないオブジェクトが存在する．それらのオブジェクトは，状態変化を追跡する必要がない．つまり，プログラムの実行中に状態が変化しないオブジェクトを特定することで，状態変化を追跡するべきオブジェクトの数を減らすことができる．これにより，プログラムを理解するためのコストを削減できる．

本研究ではJavaプログラムを対象とし，Javari[1]における不変性の定義に基づいて変数，メソッド，メソッドの戻り値，コンストラクタ，クラス，インタフェースといった構成要素間にある不変性の依存関係を定義する．Javariは，参照の不変性を明記できるようにしたJava言語の拡張である．不変な参照とは，参照しているオブジェクトの状態を変更するた

めに使われていない参照である．構成要素間の不変性は相互に依存しており，単純に決定することは不可能である．本研究でどのような依存関係が存在するか明らかにする．さらに，不変性依存関係に基づいて，Javaプログラムの不変性依存グラフを定義する．不変性依存グラフを用いて，プログラム中に出現する不変な参照変数および不変なメソッドを特定する不変性解析手法を提案する．

本論文の構成を以下に示す．2章では，Javaプログラムにおけるオブジェクトの状態を定義し，不変性の定義を述べる．3章では，2章で述べた不変性の定義をもとに不変性依存関係を定義する．不変性依存関係をグラフで表し，プログラム中に出現する要素の不変性を解析する手法を示す．4章では，ツールの実装および実行例について述べる．最後に，本論文のまとめと今後の課題について述べる．

2 Javaプログラムにおける不変性

2.1 オブジェクトの状態

以下ではオブジェクトの状態を次のように定義する．

- オブジェクト o のインスタンス変数 v が基本型ならば，v の値は o の状態である．
- オブジェクト o のインスタンス変数 r が参照型ならば，r の値および r が参照しているオブジェクトの状態は o の状態である．

参照の不変性制約は，参照が指すオブジェクトの状態が変更されないことを表す．

2.2 不変性の定義

本研究におけるJavaの不変性の定義を述べる．不変性の定義はJavari[1]に基づく．

参照変数およびメソッドの参照型の戻り値が参照しているオブジェクトの状態を変更するために使われているかを特定するために，参照の不変性を定義する．不変な参照は，参照しているオブジェクトの状態を変更するために使われない．

次のような例を考える．

```
StringBuffer sb = new StringBuffer("String");
sb.append("Buffer");
```

StringBufferクラスは，内容が変更可能な文字列のクラスである．メソッドappendは引数で渡された文字列を，自身の持つ文字列の後ろに付け加えるメソッドである．2行目においてsbの状態が変更されているため，参照変数sbは不変な参照ではない．

定義1（不変な参照） 参照 r が次の条件を満たすとき，r を不変な参照とする．

- r が参照しているオブジェクトの状態が r を用いて変更されない．
- r を用いて r が参照している配列の要素の値が変更されない．配列の要素が参照型ならば，r を用いて配列の要素が参照しているオブジェクトの状態も変更されない．
- 代入や引数渡しによって r が不変でない参照にコピーされない．

不変な参照を通して，その参照が指すオブジェクトの状態を変更するメソッドを呼び出すことはできない．不変な参照を通して呼び出すことが可能なメソッドを定めるために，メソッドの不変性を定義する．不変なメソッドは，レシーバの状態を変更しない．不変な参照から呼び出すことができるメソッドは不変なメソッドのみである．

定義2（不変なメソッド） メソッド M が次の条件を満たすとき，M を不変なメソッドとする．

- M がインスタンスメソッドである．
- M の本体に出現する，暗黙の `this` を含む `this` への参照が不変な参照である．

同様に，エンクロージングクラスのインスタンスの状態を変更しないインナクラスのコンストラクタを不変なコンストラクタとする．

クラスの不変性を定義する．不変なクラスのインスタンスは外部から状態を変更されない．そのため，不変なクラスのインスタンスへの参照は不変な参照である．

定義3（不変なクラス） クラスCが次の条件を満たすとき，Cを不変なクラスとする．

- Cが継承するインスタンスフィールドを含む全てのインスタンスフィールドが `final` な基本型もしくは `final` かつ不変な参照である．
- Cが継承するインスタンスメソッドを含む全てのインスタンスメソッドが不変なメソッドである．
- Cがインナクラスならば，Cの任意のコンストラクタが不変なコンストラクタである．
- Cの全てのサブクラスが不変なクラスである．

同様に，インスタンスの状態を外部から変更できないようなインタフェースを不変なインタフェースとする．不変なインタフェースは，そのインタフェースを実装する全てのクラスおよび全てのサブインタフェースが不変であるインタフェースである．

3 不変性依存解析

本章では，不変性の定義に基づいて不変性依存グラフを定義する．初めに，クラス，インタフェース，メソッド，メソッドの参照型の戻り値，コンストラクタ，および参照変数の間にある不変性の依存関係を定義する．次に，不変性依存関係を用いて不変性依存グラフを定義する．

3.1 不変性依存関係

不変性の定義に基づいて，クラス，インタフェース，メソッド，メソッドの参照型の戻り値，コンストラクタ，および参照変数の間に不変性依存関係を定義する．

不変性依存関係は有向関係である．直観的には，A から B への不変性依存関係があるとき，A が不変ならば B も不変である．例として，クラス T がメソッド T#m() を持ち，型が T であるような参照変数 var を考える．var.m() において，参照変数 var が不変な参照ならば，var を通して呼び出すことができるメソッドは不変なメソッドのみであるため，T#m() は不変なメソッドでなければならない．よって，参照変数 var からメソッド T#m() への不変性依存関係が存在する．

不変性依存関係を，次に示す 14 種類に分けて定義する．

1. 参照変数間の不変性依存関係
2. メソッドの戻り値から参照変数への不変性依存関係
3. メソッドの仮引数の間の不変性依存関係
4. メソッドの戻り値の間の不変性依存関係
5. メソッド間の不変性依存関係
6. メソッド呼び出し式における不変性依存関係
7. インナクラスのクラスインスタンス生成式における不変性依存関係
8. フィールド参照式における不変性依存関係
9. クラスからフィールドへの不変性依存関係
10. クラスからメソッドへの不変性依存関係
11. インナクラスからコンストラクタへの不変性依存関係
12. クラス間の不変性依存関係
13. インタフェース間の不変性依存関係
14. インタフェースからクラスへの不変性依存関係

以下では，var$i$ ($i \geq 1$) を参照型のローカル変数，fi ($i \geq 1$) を参照型のフィールド，mi() ($i \geq 1$) をメソッド，id を識別子，そして args をメソッド呼び出し式の引数部分とする．

参照変数から参照変数への代入式があった場合，定義 1 より，参照変数間の不変性依存関係を定義できる．例として，次のような文を考える．

```
var1 = var2;
```

不変な参照は不変でない参照にコピーされることはないので，var2 が不変な参照変数ならば var1 も不変な参照変数でなければならない．参照変数間の不変性依存関係を次のように定義する．

定義 4 参照変数間の不変性依存関係を次のように定める．

- 任意の代入式 var1 = var2 において，var2 から var1 への不変性依存関係がある．
- 任意のメソッド呼び出し式において，参照変数 var がメソッドの仮引数 arg に渡されているならば，参照変数 var から arg への不変性依存関係がある．
- 任意のメソッド M において，return 文が参照変数 var を返しているならば，参照変数 var からメソッド M の戻り値への不変性依存関係がある．

同様に，メソッドの戻り値から参照変数への代入式があった場合，定義 1 より，メソッドの戻り値から参照変数への不変性依存関係を定義できる．

定義 5 メソッドの戻り値から参照変数への不変性依存関係を次のように定める．

- 任意の代入式において，メソッド M の戻り値が参照変数 var に代入されているならば，メソッド M の戻り値から var への不変性依存関係がある．
- 任意のメソッド呼び出し式において，メソッド M の戻り値がメソッドの仮引数 arg に渡されているならば，メソッド M の戻り値から arg への不変性依存関係がある．
- 任意のメソッド M_1 において，return 文がメソッド M_2 の戻り値を返しているならば，メソッド M_2 の戻り値からメソッド M_1 の戻り値への不変性依存関係がある．

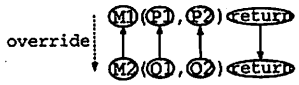

図 1: メソッドオーバーライドにおける不変性依存関係の例

メソッドのオーバーライドにおいても，不変性依存関係がある．クラス D をクラス C のサブクラスとして，次の例を考える．

```
C c = new D();
Obj obj = new Obj();
int var = c.m(obj);
```

C#m(Obj) が不変なメソッドであるならば，C 型の変数から識別子 m(Obj) で呼び出されるメソッドは不変なメソッドである必要がある．同様に，C#m(Obj) の仮引数が不変な参照ならば，C 型の変数から識別子 m(Obj) で呼び出されるメソッドの仮引数は不変な参照である必要がある．

メソッドのオーバーライドにおける不変性依存関係を定義する．直観的には，メソッド M_1(P1,P2) がメソッド M_2(Q1,Q2) をオーバーライドするとき，不変性依存関係は図 1 のようになる．

定義 6 メソッドの仮引数の間の不変性依存関係を次のように定める．任意のメソッド M_1 および M_1 をオーバーライドするメソッド M_2 について，M_1 の仮引数を $P_1, P_2, ...P_n$，M_2 の仮引数を $Q_1, Q_2, ...Q_n$ とすると，$1 \leq i \leq n$ なる全ての i について Q_i から P_i への不変性依存関係がある．

定義 7 メソッドの戻り値の間の不変性依存関係を次のように定める．任意のメソッド M_1 および M_1 をオーバーライドするメソッド M_2 について，M_1 の戻り値から M_2 の戻り値への不変性依存関係がある．

定義 8 メソッド間の不変性依存関係を次のように定める．任意のメソッド M_1 および M_1 をオーバーライドするメソッド M_2 について，M_2 から M_1 への不変性依存関係がある．

メソッド呼び出し式に対し，定義 1,2 より，参照変数，メソッドの戻り値，そして，メソッドからメソッドへの不変性依存関係を定義する．例として，メソッド M の本体に含まれる次のような部分式を考える．

```
var1.f1.m1()
var2.f2.m2().f3.f4.m3()
this.f5.m4()
```

1 行目の式では，var1 もしくは f1 が不変な参照ならば var1.f1 は不変な参照となる．不変な参照から

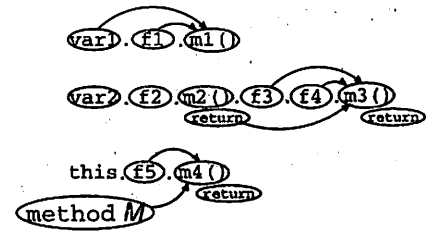

図 2: メソッド呼び出し式における不変性依存関係の例

呼び出せるメソッドは不変なメソッドのみであるから，var1.f1 が不変な参照ならば，m1() は不変なメソッドである．2 行目，3 行目の式についても同様である．上記の式において，図 2 に示す不変性依存関係がある．メソッド呼び出し式における不変性依存関係を定義するために，広義のレシーバを定義する．

定義 9 部分式 $e1$ が，$e2$.id(args) なるメソッド呼び出し式または $e2$.id なるフィールド参照式ならば，部分式 $e1$ の広義のレシーバを次のように定義する．

- 部分式 $e2$ がフィールド参照式ならば，そのフィールドへの参照および $e2$ の広義のレシーバは，部分式 $e1$ の広義のレシーバである．
- 部分式 $e2$ がローカル変数参照式ならば，そのローカル変数への参照は，部分式 $e1$ の広義のレシーバである．
- 部分式 $e2$ がメソッド M の呼び出し式ならば，M の戻り値への参照は部分式 $e1$ の広義のレシーバである．
- 部分式 $e2$ が式 this でありかつ，部分式 $e1$ がメソッドの M の本体に含まれるならば，メソッド M は部分式 $e1$ の広義のレシーバである．

上記以外の場合，部分式 $e1$ の広義のレシーバは存在しない．

前述の式を用いて例を示す．

```
var1.f1.m1()
var2.f2.m2().f3.f4.m3()
this.f5.m4()
```

1 行目の式の広義のレシーバは var1 および f1，2 行目の式の広義のレシーバは m2() の戻り値，f3，および f4，3 行目の式の広義のレシーバはメソッド M および f5 である．

広義のレシーバを用いて，メソッド呼び出し式における不変性依存関係を定義する．

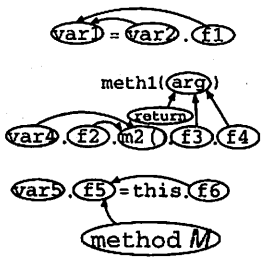

図 3: フィールド参照式における不変性依存関係の例

定義 10 メソッド呼び出し式における不変性依存関係を次のように定める．任意のメソッド呼び出し式 e について，e がメソッド M を呼び出しているならば，e の広義のレシーバからメソッド M への不変性依存関係がある．

インナクラスのインスタンス生成式における不変性依存関係も，メソッド呼び出し式の場合と同様に定めることができる．

定義 1,6 からフィールド参照式における不変性依存関係を定義する．例として，メソッド M の本体に含まれる次のような部分式を考える．

```
var1 = var2.f1
m1(var4.f2.m2().f3.f4)
var5.f5 = this.f6
```

1 行目の式では，var2 もしくは f1 が不変な参照ならば var2.f1 は不変な参照となる．不変な参照は代入や引数渡しによって不変でない参照にコピーされないので，var2.f1 が不変な参照ならば var1 は不変な参照変数である．2 行目，3 行目の式についても同様である．上記の式において，図 3 に示す不変性依存関係がある．

定義 11 フィールド参照式における不変性依存関係を次のように定める．

- 任意の代入式 var=e において，e がフィールド f の参照式ならば，e の広義のレシーバおよび f から var への不変性依存関係がある．
- 任意のメソッド呼び出し式において，フィールド参照式 e によって参照されたフィールド f がメソッドの仮引数 arg に渡されているならば，e の広義のレシーバおよび f から arg への不変性依存関係がある．
- 任意のメソッド M において，return 文がフィールド参照式 e によって参照されたフィールド f を返しているならば，e の広義のレシーバおよび f からメソッド M の戻り値への不変性依存関係がある．

不変なクラスの定義から，クラス宣言における不変性依存関係を定義できる．

定義 12 クラスからフィールドへの不変性依存関係を次のように定める．クラス C で宣言されている任意のインスタンスフィールドと C が親クラスから継承する任意のインスタンスフィールド f について，f が参照変数ならば C から f への不変性依存関係がある．

定義 13 クラスからメソッドへの不変性依存関係を次のように定める．クラス C で宣言されている任意のインスタンスメソッドと C が親から継承する任意のインスタンスメソッド M について，C から M への不変性依存関係がある．

定義 14 インナクラスからコンストラクタへの不変性依存関係を次のように定める．インナクラス I で宣言されている任意のコンストラクタ C について，I から C への不変性依存関係がある．

定義 15 クラス間の不変性依存関係を次のように定める．クラスからその直接のサブクラスへの不変性依存関係がある．

不変なインタフェースの定義から，インタフェース間の不変性依存関係およびインタフェースからクラスへの不変性依存関係を定義できる．直観的には，インタフェースから，そのインタフェースを実装しているクラスおよびそのインタフェースのサブインタフェースへの不変性依存関係がある．

3.2 不変性依存グラフ

不変性依存関係を表現した有向グラフを，不変性依存グラフとし，次のように定める．

定義 16 不変性依存グラフは有向グラフである．不変性依存グラフを構成するノードとエッジを，以下のように定義する．

- ノード
  - クラス
  - インタフェース
  - コンストラクタ
  - メソッド
  - メソッドの参照型の戻り値
  - 参照変数
- エッジ
  不変性依存関係

3.3 不変性解析

定義 1,2 より，代入式など一つの式から参照変数，メソッド，メソッドの戻り値が不変でないことが分かる場合がある．例えば，メソッド M 内のフィールド参照式 this.var.f は，メソッド M または参照変数 var が不変ならば値を代入されない．すなわち，この式が代入式の左辺式ならば，メソッド M は不変なメソッドでなく，参照変数 var が不変な参照でないことが分かる．

さらに，M が不変でないならば，M への不変性依存関係を持つクラス，コンストラクタ，メソッド，メソッドの戻り値，参照変数は不変でない．よって，一つの式から不変でないことが分かる要素を特定し，その要素から不変性依存関係を逆向きにたどることで不変でない全ての要素を特定することができる．

不変性依存グラフのノードに対応する要素の不変性を解析する．不変でないノードを次のように定義する．

定義 17 ノード N が次のいずれかの条件を満たすならば，ノード N は不変でない．

- 次のいずれかの条件を満たす代入式の左辺式 e がプログラム中に存在する
  - e の広義のレシーバが N に対応する *Java* プログラムの要素を含む．
  - e が配列型の変数 arr の要素を参照しておりかつ，arr のノードが N である．
- N が final でないインスタンスフィールドを持つクラスのノードである．
- N が final でも private でもないインスタンスフィールドを持つクラスのサブクラスのノードである．
- N が不変でないノードへの経路を持つ．

定義 17 のうち，1〜3 個目の項目に該当する不変でないノードは，一つの部分式のみから見付けることができる．不変性解析では，一つの部分式のみから不変でないことが分かるノードを全て見付けた後，それらのノードへの経路を持つノードを不変でないノードとする．

不変でないノードのいずれの定義にも当てはまらないノードを不変なノードとする．ノード N が不変なノードならば，N に対応する Java プログラムの要素は不変である．

不変性依存グラフの例を示す．図 4 のプログラムの不変性依存グラフは図 5 のようになる．

```
public class A{
  B b;
  void setB(C c){
    b.setC(c);
  }
  C getB(){
    return b.getC();
  }
}

public class C{
  int i;
}
```

```
public class B{
  C c;
  void setC(C c){
    this.c = c;
  }
  C getC(){
    return c;
  }
}
```

図 4: サンプルのプログラムソース

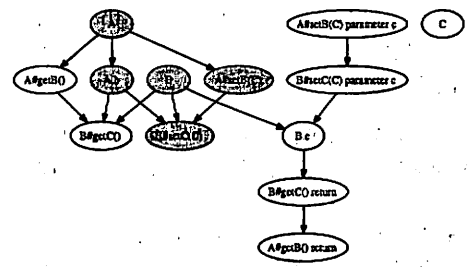

図 5: 図 4 のプログラムに対する不変性依存グラフ

塗りつぶされたノードが不変でないノード，それ以外が不変なノードである．B.c のノードが不変であることから，参照変数 B.c が不変な参照変数であることが分かる．また，A.b のノードが不変なノード B#setC(C) へのエッジを持つことから，メソッド B#setC(C) を実行したときに参照変数 A#b の状態が変化しており，A.b が不変な参照変数でないことが分かる．

4 ツールの実装

本研究では，不変性依存関係を用いて Java プログラムにおける不変性解析を行い，解析結果に基づいてプログラムを HTML 形式で出力し不変な要素と不変でない要素を色分けするツールを実装した．不変な要素と不変でない要素を色分けすることで，追跡するべきオブジェクトとしなくてもよいオブジェクトを明示しプログラムの動作理解を支援する．

4.1 ツールの概要

本ツールは Java プログラムの構文解析に Japid[2] を利用し，その構文解析の結果である J-model を用いて不変性解析を行う．Java プログラムにおける不変性解析を行い，解析結果に基づいて不変な要素と不変

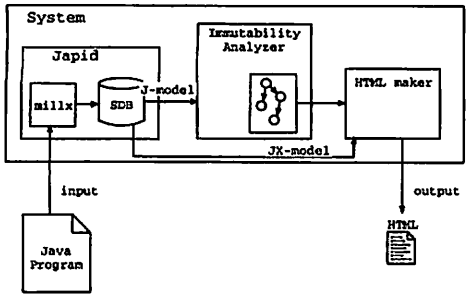

図 6: システム構成図

でない要素を色分けしたプログラムコードを HTML 形式で出力するツールを実装した．実装したツールは以下の 2 つのサブシステムからなる．

- 不変性解析を行うシステム
  解析対象ファイル群の J-model を入力とし，不変性依存グラフを作成して不変性解析を行う．ノードに不変性の真偽値を付加した不変性依存グラフを出力する．
- 解析結果を HTML で出力するシステム
  ノードに不変性の真偽値が付加された不変性依存グラフおよび，対象ファイル群の JX-model を入力とする．不変性依存グラフをもとに，不変な要素と不変でない要素を色分けしたプログラムコードを HTML 形式で出力する．

ツールのシステム構成図を図 6 に，図 4 のプログラムを入力されたときに出力される HTML を図 7 に示す．図 7 において，宣言部分の色が薄くなっているメソッドおよび参照変数が不変なメソッドおよび不変な参照変数である．メソッド `A#getB()` およびメソッド `B#getC()` が，メソッドを呼び出したオブジェクトの状態を変更していないことが分かる．また，`B.c`，メソッド `A#setB(C)` の仮引数 `c` および，メソッド `B#setC(C)` の仮引数 `c` は，参照しているオブジェクトの状態を変更するために使われていないことが分かる．よって，上記の要素はオブジェクトの状態変化を追跡する際に無視することができる．

4.2 評価

Java1.4.2_11 の API に含まれるいくつかのパッケージを入力としてツールを実行した結果は表 1 のようになった．総行数の項目はパッケージに含まれるプログラムの行数の合計，総実行時間の項目は Japid の解析結果を取り込んで不変性解析を行い，出力を完了するまでの時間，不変性解析時間の項目は，実行時間のうち不変性解析の部分のみの時間，そして，不変なメソッドおよび参照変数の項目はそれぞれ，入力されたプログラムに含まれるメソッド，参照変数の総数と，そのうち不変なものの数である．

表 1 の結果から，平均してプログラムに出現するメソッドおよび参照変数の半数以上が不変であることが分かる．よって，本手法を適用することによってオブジェクトの状態変化を追跡する際に調べるメソッドおよび参照変数が半分以下になるため，プログラム理解のためのコストを削減できると言える．

5 おわりに

5.1 まとめ

本論文では，Java プログラムの不変性依存関係と不変性依存グラフを定義した．定義した不変性依存グラフを用いて，オブジェクトの状態を変更するために使われていない参照を特定する不変性解析手法を提案した．提案した手法を用いて不変性解析を行い，メソッド，クラス，参照変数を不変性で色分けして表示するツールを実装した．

オブジェクトの状態を変更しない参照を特定することで，プログラムの動作を理解するためにオブジェクトの状態変化を追跡する際に調べるべき参照変数の数を削減できる．また，呼び出したオブジェクトの状態を変更するメソッドを特定することで，オブジェクトの状態変化を追跡する際に本体を調べるべきメソッドの数を削減できる．調べるべき参照変数およびメソッドの数を削減することでプログラムの動作理解にかかるコストを削減できる．そのため，プログラムの動作を理解する際に，不変性依存グラフは有用である．

5.2 関連研究

文献 [4] で定義している手法では，オブジェクトの状態を変更するメソッドが可視であるような参照変数を全て不変でない参照としている．そのため，実際にはオブジェクトの状態を変更していない参照を不変でない参照とすることがある．さらに，ローカル変数やメソッドの仮引数は解析の対象としていない．

JAC[5] では，Javari とは異なった参照の不変性を定義している．JAC は不変性を含むアクセス可能性の階層構造を提供している．しかし，配列への参照やインナクラスの不変性ついては定義がなされていない．

5.3 今後の課題

- モジュール構造による不変性依存関係への影響の考慮

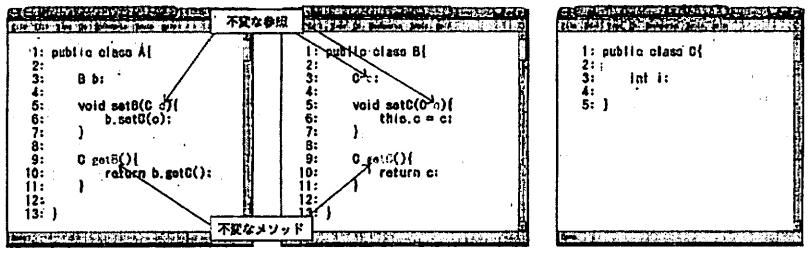

図 7: ツールの出力例

表 1: ツールの実行結果

対象	総行数	総実行時間 (msec)	不変性解析時間 (msec)	不変なメソッド	不変な参照変数
java.util.jar	2387	4547	1102	70/114	132/203
java.util.zip	3729	6295	1384	128/225	95/155
java.math	4788	6700	1830	105/170	109/303
java.util.prefs	4844	7695	1866	83/207	195/312
java.util.logging	5487	7633	1985	86/215	286/486
java.util.regex	6161	21810	4003	131/286	160/368
java.beans および java.beans.beancontext	15283	20895	5381	349/684	1029/1657
java.io	25014	24579	5920	638/1130	647/1143
java.text	25620	87914	6141	341/757	541/1118
java.lang, java.lang.ref, java.lang.reflect	33547	38129	4584	963/1269	980/1418

本研究の手法は入力されたプログラムのみに閉じて不変性解析を行う．入力されたプログラムのみで見た場合に不変であっても外部から変更される可能性のある要素を特定するためには，手法を改良する必要がある．

- 既存のプログラムの特性分析
  本研究の不変性解析手法を用いて，Java のクラスライブラリや，既存のフレームワーク，スタンドアロンアプリケーションにおける不変なメソッド，フィールド，クラスの割合を調べることで特性分析を行うことが考えられる．

謝辞

本研究を進めるにあたり熱心に議論して頂いた阿草研究室の皆様に感謝致します．本研究の一部は文部科学省リーディングプロジェクト e-Society「高信頼 WebWare 生成技術」の助成による．

参考文献

[1] Tschantz, M. S. and Ernst, M. D.: Javari: Adding reference immutability to Java, *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, San Diego, CA, USA, pp. 211–230 (2005).

[2] Yoshinari, H., Shinichirou, Y. and Kiyoshi, A.: A CASE Tool Platform for an Object Oriented Language, *IEICE Trans. on Information and Systems*, Vol. E82-D, No. 5, pp. 977–984 (1999).

[3] Ellson, J., Gansner, E. R., Koutsofios, L., North, S. C. and Woodhull, G.: Graphviz and dynagraph - static and dynamic graph drawing tools, *Graph Drawing Software* (Junger, M. and Mutzel, P., eds.), Springer Verlag (2003).

[4] Porat, S., Biberstein, M., Koved, L. and Mendelson, B.: Automatic detection of immutable fields in Java, *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, p. 10 (2000).

[5] Kniesel, G. and Theisen, D.: JAC - access right based encapsulation for Java, *Software: Practice and Experience*, Vol. 31, No. 6, pp. 555–576 (2001).