ユーザー定義されたプログラム解析を利用するアスペクト指向プログラムのコンパイル手法

A Framework for Defining and Compiling User-Defined Program Analysis in AOP

青谷 知幸 † 増原 英彦 †

Tomoyuki AOTANI Hidehiko MASUHARA

aotani@graco.c.u-tokyo.ac.jp masuhara@acm.org

† 東京大学大学院総合文化研究科

Graduate School of Arts and Sciences, The University of Tokyo

本研究では, AspectJ 言語に対するコンパイルの枠組である SCoPE の改良を提案する. SCoPE は静的な条件ポイントカットを自動的に見つけ出し, 静的に評価することができる. 改良は束縛時検査の高速化と, Java バイトコード操作ライブラリと SCoPE との連携から成る. これによって, (1) ユーザーが Java バイトコードの複雑な静的解析を行う静的なポイントカットを記述できるようになり, (2) これを実用的な時間でコンパイルすることが可能となった.

1 はじめに

アスペクト指向プログラミング (AOP) とは, ロギングや永続化などの, モジュール間に跨って存在する関心事をアスペクトとしてモジュール化するプログラミング技法であり, ジョインポイント (プログラム実行時に起こるメソッド呼び出しなどのアクション), ポイントカット (ジョインポイントを指定する手段) およびアドバイス (ポイントカットで指定したジョインポイントに対してログ取りを追加するなどの影響を与える手段) からなる. 例えば, AspectJ 言語による以下のアスペクト:

```
1 before(): execution(* *(..)){
2 writeLog(thisJoinPoint);
3 }
```

は, `execution(* *.*(..))` というポイントカットによって, 関数の実行というジョインポイントを指定し, `before` アドバイスによって, そのジョインポイントの直前に関数の実行履歴を記録することを指定している.

ポイントカットはアスペクト自体のモジュール性を確保するために重要であり [10], call ポイントカットのようにメソッドの名前を直接に指定するのではなく, プログラムの性質を記述することによってポイントカットを記述できるようにすることが提案されている. このような表現力の高いポイントカットには, プログラムの静的解析によってコントロールフローを予測する pcflow ポイントカット [12] や, クラスの構造を調べる has/hasfield ポイントカット [6] がある. また, このような特別なポイントカットを逐次定義するのではなく, ユーザーが自由に定義できるような言語拡張 [7–9] が提案されてきた.

表現力の高いポイントカットを実現するための方法として, 我々は AspectJ 言語に対するコンパイルの枠組み SCoPE を提案した [1]. SCoPE には Josh [7] などのユーザーが新しいポイントカットを定義できる類似の機構と異なり, AspectJ 言語を拡張しないという特徴がある. SCoPE は条件ポイントカットと自己反映計算を用いて定義されたポイントカットをコンパイル時に評価する. このため, SCoPE のユーザーは実行時オーバーヘッドを導入しない様々なポイントカットを定義できる. 例えば, has/hasfield ポイントカットや, メソッドの名前に対して正規表現との適合を調べるポイントカットは, SCoPE でコンパイルすることによって, 実行時オーバーヘッドを導入せずに済む. だが, 命令レベルの解析が必要な pcflow などのポイントカットは, 定義できなかった.

そこで, 本研究では SCoPE を拡張し, ユーザーによって定義されたプログラムの解析メソッドを利用するポイントカットを実用時間内にコンパイルできるようにする方法を提案する. この拡張は (1) 静的に評価可能な条件ポイントカットを発見するための手段である束縛時検査の高速化と (2) バイトコード解析ライブラリとの円滑な連携のための枠組みの構築から成る. これによって, ユーザーが Java バイトコー

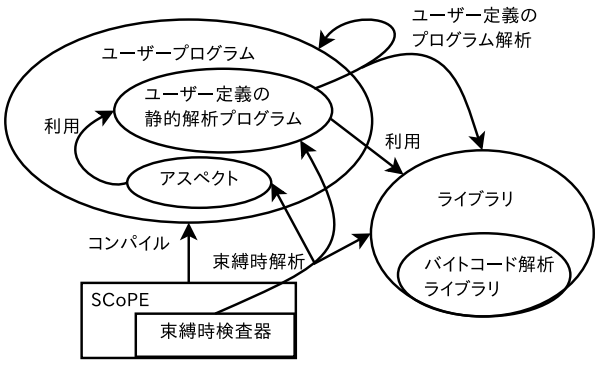

図 1: SCoPE とユーザープログラムの関係

ドの複雑な静的解析を行う静的なポイントカットを簡潔に記述できるようになり, これを実用的な時間で発見することが可能となった.

以降, 本稿は以下のように構成される. まず 2 節で, 本稿に登場するプログラムの種類とそれらの関係について概観する.3 節では, デメテル則 [15] を検査するポイントカットを例に, 条件ポイントカットと自己反映計算を用いて表現力の高いポイントカットが記述できることを示す. 4 節では, 束縛時検査による静的条件ポイントカットの発見における問題点とその解決を述べる. また, 5 節ではバイトコード解析ライブラリを円滑に使用するための枠組みを説明する. 6 節では, 本稿で提案する拡張によって静的に評価できるようになったポイントカットを例示するとともに, 束縛時検査が十分に高速であることを検証する. 7 節では関連研究を議論し, 8 節にまとめと今後の課題を置いて本稿を締めくくる.

2 本論文に登場するプログラムの種類とその関係

この節では, SCoPE とそのコンパイル対象であるユーザープログラムとの関係 (図 1) について説明する. SCoPE はユーザープログラムをコンパイルする. ユーザープログラムは, アスペクトとそれ以外のベースプログラムからなる. アスペクトはユーザー定義の静的解析プログラムを利用する. ユーザー定義の静的解析プログラムは, ユーザープログラムとライブラリを解析する. SCoPE の束縛時検査器は, アスペクトと, アスペクトが利用している静的解析プログラムおよびライブラリを検査する.

```
1  pointcut violation(): call(* *(..))
2   && (!if(Supplier.isPreferredSupplier(
3      ejp, tjp)));
4  after(): violation(){
5   printErrorMessageWithTheLocation(tjp);
6  }
```

図 2: デメテル則の違反を検出するアスペクト

3 例: デメテル則の検査

本節では, 条件ポイントカットと自己反映計算を用いたポイントカットの例として, デメテル則 [14, 15] のクラスフォームを検査するポイントカットを取り上げる.

この節において, ユーザープログラムはデメテル則の検査の対象となるプログラムである. アスペクトはデメテル則の違反を検出すると, 違反箇所のソースコード上の位置を出力するプログラムであり, 静的解析プログラムはユーザープログラムに対してデメテル則を検査する.

3.1 デメテル則のクラスフォーム

ここでは, 検査項目である, デメテル則とそのクラスフォームを簡単に紹介する. デメテル則とは, クラス同士の依存関係を小さくするためのプログラムの作法である. デメテル則のクラスフォームとは, デメテル則を静的に検査できるようにしたものである. デメテル則のクラスフォームは, クラス c のメソッド $c.m$ の定義中に現れるメソッド呼び出し式の対象となるオブジェクトのテキスト上の型 T が, 以下の 5 つの条件のいずれかを満たすとき, 成立する.

1. T は c の派生クラスである
2. T は c のフィールドの型である
3. T は $c.m$ の引数の型である
4. T は $c.m$ の戻り値の型である
5. T は $c.m$ の定義の中に現れるインスタンス生成式 `new C(...)` のクラス `C` である

3.2 条件ポイントカットによるクラスフォームの検査

図 2 に, このポイントカットを用いて実行時にデメテル則の違反箇所を検出するアスペクトを示す. このアスペクトは, デメテル則のクラスフォームに違反す

```
 1  boolean isaPreferredSupplier
 2    (JoinPoint.StaticPart ejp,
 3    JoinPoint tjp){
 4   return
 5   isSubTypeOfThisType(ejp,tjp)
 6    ||isMemberVarType(ejp,tjp)
 7    ||isArgType(ejp,tjp)
 8    ||isReturnType(ejp,tjp)
 9    ||isPreferredAcquaintance(ejp,tjp);
10  }
```

図 3: Supplier.isPreferredSupplier

るメソッド呼び出しが起こると, 実行時にその違反箇所のソースコード上の位置を出力する.

図 2 の 1 行目から 3 行目までは violation という名前のポイントカットを定義している. 1 行目の call(**(..)) は任意のメソッド呼び出しに適合する. 2 行目から 3 行目は条件ポイントカット if(e) を用いてメソッド呼び出しの中でもデメテル則を満たさないものに適合するポイントカットを定義している. 条件ポイントカットは, 任意の Java の式が記述できる. ここで, Supplier.isPreferredSupplier(ejp,tjp) は現在のジョインポイント ejp, tjp を使ってメソッド呼び出しをしている. tjp および ejp は, ジョインポイントを表す特別な変数である[1].

4 行目から 6 行目まではデメテル則に違反するメソッド呼び出しが終わった後に, 違反箇所を表示するアドバイスを定義する. 4 行目の after は, 後に続くポイントカットに適合するジョインポイントの直後にアドバイスを実行することを宣言する. このようなアドバイスを after アドバイスと呼ぶ. 5 行目はアドバイスで実行される処理を定義する. ここでは, メソッドを呼び出したソースコード上の位置を出力する.

図 3 の isPreferredSupplier(ejp,tjp) では, デメテル則の条件 1 から 5 の項目を検査する. このうち, 条件 5 を除く項目は, ジョインポイントオブジェクトに対する自己反映計算と Java Reflection APIs を用いることで検査可能である. 例えば, 条件 3 は図 4 に示す関数によって検査できる.

条件 5 は, Java Reflection APIs だけでは定義することができないが, バイトコード解析ライブラリを利用することによって解決する. これについては, 5 節で述べる.

[1] tjp は thisJoinPoint, ejp は thisEnclosingJoinPoint の略である

```
 1  boolean isArgType
 2   (JoinPoint.StaticPart ejp,
 3    JoinPoint tjp){
 4   Class supplierT=
 5    tjp.getSignature().getDeclaringType();
 6   CodeSignature msig=
 7    (CodeSignature)ejp.getSignature();
 8   return
 9    Arrays.asList(msig.getParameterTypes())
10    .contains(supplierT);
11  }
```

図 4: *T* が *c.m* の引数の型であることを検査する関数

4 束縛時検査を用いた静的条件ポイントカットの発見

SCoPE は条件ポイントカットに対して, Java バイトコードの上で束縛時検査を行うことによって静的条件ポイントカットを自動的に見つけ出す. このため, 束縛時検査は高速に, より多くの静的条件ポイントカットを見つけ出せることが望ましい. 本節では, まず静的条件ポイントカットと, それを見つけ出す束縛時検査を概観した後, 束縛時検査を行う際の問題点を挙げ, 我々の取った解決策を示す.

4.1 静的条件ポイントカット

4.1.1 定義

静的条件ポイントカットとは, 条件ポイントカット if(e) のうちで, 静的ポイントカットと同様に, ジョインポイントシャドウの上で e を評価することのできるものである.

ポイントカットの評価とは, 与えられたジョインポイントに対してそのポイントカットが適合するか否かを決定することである. AspectJ 言語の実行モデルでは, ジョインポイントはプログラムの実行中に起こるアクションであるため, ポイントカットはプログラムの実行中に評価される.

だが, ポイントカットの中にはプログラムを実行する前に評価することが可能なものもある. このようなポイントカットを静的ポイントカットと呼ぶ. 静的ポイントカットは, ソースコード上の場所に応じてコンパイラが作成するジョインポイントシャドウ [16] の上で評価できる. 例えば, call ポイントカットや execution ポイントカットは静的ポイントカットである.

条件ポイントカット `if(e)` は, 既存の AspectJ の処理系においては動的ポイントカットとして扱われている. これは, 任意の Java の式が `e` として与えられるためである. だが, このような条件ポイントカットも, 与えられる `e` に依っては, ジョインポイントシャドウの上で評価することが可能である. 例えば, 3 節で示した, 条件 1 から条件 5 は全てソースコードの上で容易に調べることができる. このため, デメテル則を検査するポイントカット `if(Supplier.isPreferredSupplier(ejp,tjp))` は, 条件ポイントカットとして定義されているが, 実際はジョインポイントシャドウに対して評価できるはずである. このような, 静的ポイントカットと同様に評価できる条件ポイントカットが, 静的条件ポイントカットである.

4.1.2 恩恵

静的条件ポイントカットをコンパイル時に評価してしまうことによって, 実行時オーバーヘッドを削除することができる. 例えば, 3 節で示したデメテル則を検査するポイントカットは自己反映計算を多用するため, `Supplier.isPreferredSupplier(ejp,tjp)` の評価には自己反映計算のオーバーヘッドが付きまとう. このような条件ポイントカットを静的条件ポイントカットとしてコンパイル時に評価することによって, プログラムの実行速度を大きく改善できる[2].

4.2 束縛時検査

束縛時検査とは, 式やメソッドがプログラムを実行する前にあるジョインポイントシャドウの下で評価できるか否かの検査である. 束縛時とは式やメソッドが評価できる時のことであり, 静的 (プログラムの実行前) を表す S と動的 (プログラムの実行時) を表す D で構成される. 以降, 図 4 で示した `isArgType` 関数を用いて, 束縛時検査の方法を簡単に説明する.

`isArgType` 関数の束縛時は,

- 引数 `ejp` および `tjp` の束縛時が全て S
- 2 行目から 8 行目までの式の束縛時が全て S

であるとき S となる. 式の束縛時が S であることは, 例えば, 5 行目の `ejp.getSignature()` では,

- 変数 `ejp` の束縛時が S
- `JoinPoint.StaticPart.getSignature()` の束縛時が S

であるときに成立する. また, 変数の束縛時が S であるとは, 4 行目の `msig` を例にとれば, `ejp.getSignature()` の束縛時が S であるときに成立する.

以上の条件の 1 つでも満たされなかった時, 関数 `isArgType` の束縛時は D となる. この時, 束縛時検査は束縛時が D となる式や変数を見つけた時点で終了する. 例えば, 5 行目の `ejp.getSignature()` の束縛時が D と判った時点で `isArgType` の束縛時も D と決定し, 以下の 6 行目から 8 行目までの式の束縛時は調べない. この点で, 部分計算 [5, 11] で用いられる, プログラムのどの部分がコンパイル時に計算可能かを調べる束縛時解析と異なる.

4.3 束縛時検査を用いた静的条件ポイントカットの検出

SCoPE は条件ポイントカットに対して束縛時検査を行うことによって静的条件ポイントカットを見つけ出す. 束縛時検査では, 条件ポイントカットからメソッド呼び出し関係で到達可能な式の全てが静的に評価可能であることを調べる. 具体的には, 条件ポイントカット `if(e)` について, 以下の項目を検査する.

- `e` からメソッド呼び出し関係で到達可能な全てのメソッドの束縛時が S となるメソッドである
- `e` に現れる自由変数は, ジョインポイント変数 (`thisJoinPoint`, `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart`) か, int や float などのプリミティブ型や String 型などの値の変更できない final なクラス変数である.
- `e` がジョインポイントオブジェクトの動的なフィールドの値を読み書きしない

ここで, メソッドの束縛時が静的であるとは, メソッドを定義する式が以下を満たすことである.

- `e` に現れる自由変数は, ジョインポイント変数か, int や float などのプリミティブ型や String 型などの値の変更できない final なクラス変数である.
- ジョインポイント・オブジェクトの動的なフィールドの値を読み書きしない
- 呼び出されるメソッドの束縛時は全て S.

[2] Java Grande Forum により提供される sequencial benchmarks のプログラムと, 条件 5 を除いたデメテル則の検査するアスペクトをコンパイルして実験した結果,2%から 370%の実行速度の改善が見られた [1]

```
 1  aload_1
 2  invokestatic \
 3   getSupplierType(JoinPoint.StaticPart);
 4  astore_1
 5  aload_0
 6  invokeinterface \
 7   JoinPoint.StaticPart.getSignature(),1;
 8  checkcast CodeSignature;
 9  invokeinterface \
10   CodeSignature.getParameterTypes(),1;
11  invokestatic
12   Arrays.asList(Object[]);
13  aload_1
14  invokeinterface \
15   List.contains(Object),2;
16  ireturn
```

図 5: isArgType をコンパイルした結果のバイトコード

例えば，メソッド isArgType(図 4) は，ジョインポイントオブジェクトの静的なフィールド情報にしかアクセスせず，大域変数も現れないため，束縛時検査によって静的と判定される． このため，if(isArgType(ejp,tjp)) という条件ポイントカットがあった時, これは束縛時検査によって静的条件ポイントカットと判定される.

4.4 保守的な束縛時検査とその問題点

静的条件ポイントカットの発見のために, SCoPE は Java バイトコードの束縛時を検査するが, この時, invokeinterface および invokevirtual 命令を使った仮想メソッドの呼び出し式に対する束縛時の決定が問題となる. ここでは, isArgType 関数の束縛時を検査する例について, 束縛時検査の問題点を簡単に示した後, メソッド呼び出し式の束縛時の定義を定式化し, 保守的な束縛時検査の問題を述べる.

4.4.1 例:isArgType の束縛時検査における問題点

ここでは, 束縛時検査におけるでメソッド呼び出し式の問題点とその解決策を, 図 4 で示した isArgType を例に説明する.

isArgType をコンパイルすると, 図 5 のようなバイトコード列を得る. これを上から順に検査することで, isArgType の束縛時検査が行われる.

aload_1 などの式の束縛時を決定することは容易だが, 5,7 および 10 行目に現れる invokeinterface 命令によるメソッド呼び出し式の束縛時を検査することは容易ではない. invokeinterface 命令によって実行時に呼び出されるメソッドは, 実行時の型によって決まる. これらの式の束縛時を調べるには, 実際に呼び出される可能性のあるメソッド全ての束縛時を調べなければならない.

我々はこのようなメソッド呼び出し式の束縛時を, プログラムの実行時にその式によって呼び出される可能性のあるメソッドの集合を求め, それらの束縛時のうち 1 つでも D があれば D, そうでなければ S とする. 呼び出される可能性のあるメソッドの集合は, メソッド呼び出しのターゲットオブジェクトの実行時型となる可能性のあるクラスの集合を決めることによって計算する.

この方法によって, 近似的に束縛時検査を行うことができる. 求めたメソッドの集合と実際に呼び出されるメソッドの集合がの差が小さい程, この近似の精度が上がる. このため, メソッド集合を計算するためのクラスの集合をどのように決定するかが重要となる.

4.4.2 仮想メソッド呼び出し式の束縛時

ここでは, 4.4.1 で述べた, メソッド呼び出し式の束縛時の決定を定式化する. 束縛時 S と D の大小関係は $S < D$ である. 仮想メソッド呼び出し式の束縛時は, その式によって呼び出され得るメソッドの束縛時の最小上界として定義する:

$\mathcal{B}(\text{invkd } c.m, C) =$

$\sup\{\mathcal{B}(c'.m, C) | c' \in C \wedge c' <: c \wedge m \in methods(c')\}$

ここで, invkd は invokevirtual か invokeinterface 命令を表し, $c.m$ はクラス c のメソッド m を表す. また, $methods(c')$ はクラス c' で定義されたメソッドの集合であり, C はメソッド呼び出しのターゲットオブジェクトの実行時型となり得る可能性のあるクラスの集合である. $c.m$ の束縛時 $\mathcal{B}(c.m, C)$ も同様に, メソッドを定義する式の束縛時の最小上界として定義する:

$\mathcal{B}(c.m, C) = \sup\{\mathcal{B}(s, C) | s \in body(c.m)\}$

ここで, $body(c.m)$ は $c.m$ を定義する式の列 $\langle s_1, s_2, ..., s_n \rangle$ である.

4.4.3 素朴なクラス集合とその問題点

コンパイラに与えられた全てのクラスを含む集合 C_{all} は素朴なクラス集合である. コンパイラに与えら

れた全てのクラスとは, コンパイラに渡されたライブラリとソースファイルに含まれるクラスの全部を意味する.

メソッド呼び出し式の束縛時を決定するために, 素朴なクラス集合 C_{all} を用いることは, 束縛時検査の (1) 速度と (2) 精度を低下させる.

プログラムの実行時とコンパイル時とで同じクラス集合 C_{all} が与えられると仮定した時, 条件ポイントカット *pcd* を評価するために使われるクラスの集合 C_{pcd} は $C_{pcd} \subseteq C_{all}$ を必ず満たすため, 常に $\mathcal{B}(\text{invkd } c.m, C_{all}) \geq \mathcal{B}(\text{invkd } c.m, C_{pcd})$ が成立する. だが, C_{all} を用いた束縛時検査は保守的すぎる. これは, $C_{all} \setminus C_{pcd}$ が大きすぎるためである. 検査しなければならないメソッドの集合が大きくなるために, 束縛時検査の対象として不必要なメソッドが増え, その結果, 以下の 2 つの問題を招く.

束縛時検査の速度の低下 束縛時検査の対象となる不必要なメソッドが増えると, 束縛時検査にかかる時間が増える. `isArgType`(図 5) の束縛時を検査するためには `List.contains(Object)` メソッドを呼び出す式 (14–15 行目) の束縛時を調べる必要がある. このメソッド呼び出しのターゲットオブジェクトは, 明らかに `Arrays.asList(Object[])` メソッドを呼び出した結果生成される ArrayList オブジェクトである (11–12 行目)[3]. だが, C_{all} はコンパイラに渡されたライブラリに含まれる全てのクラスを含むため, Vector クラスや LinkedList クラスなどの List インターフェースを実装した全てのクラスが, メソッド呼び出しのターゲットオブジェクトの実行時型の候補とされてしまう. この結果, 不要な Vector.containts(Object) メソッド及び LinkedList.contains(Object) メソッドの束縛時を検査しなければならなくなり, 検査のためにかかる時間が増える.

束縛時検査の精度の低下 束縛時検査の対象となる不必要なメソッドが増えると, 束縛時検査によって発見可能な静的条件ポイントカットの数を減らす危険がある. 何故ならば, 不必要なメソッドの束縛時が 1 つでも D になるとき, メソッド呼び出し式の束縛時も D となり, 条件ポイントカット全体の束縛時が D となるためである.

例えば, 図 5 で挙げた `isArgType` の 14–15 行目における `List.contains(Object)` メソッドの呼び出しは, ユーザープログラムの中に List インターフェースを実装したクラスが存在し, そのクラスの `contains(Object)` メソッドがデバッグ用の出力などの副作用のある処理を行うとき, その束縛時は D となる. このため, `List.contains(Object)` メソッドの呼び出し式の束縛時は D となる. だが, 14–15 行目のメソッド呼び出し式のターゲットオブジェクトの実行時型は 11-12 行目によってインスタンスが生成される `ArrayList` クラスであり, `ArrayList.contains(Object)` の束縛時は S である. よって, 実際の 14–15 行目のメソッド呼び出し式の束縛時は S となる. 本来ならば束縛時が S となるはずの式の束縛時が D となってしまっているため, クラス集合として C_{all} を選ぶことは, 束縛時検査の精度が低下させ, 結果として発見できる静的ポイントカットの数を減らしてしまう危険がある.

4.5 束縛時検査の改善

我々は, (1) 条件ポイントカット *pcd* 毎にクラス集合を計算し, (2) その下で条件ポイントカット *pcd* の束縛時検査を行うことで,4.4.3 節に挙げた問題を解決した. クラス集合の計算には, メソッド呼び出しのターゲットオブジェクトが取りうる実行時型の候補を, 条件ポイントカット *pcd* から呼び出し関係で到達可能なメソッドの内部でインスタンスの生成が行われたクラスに限定する rapid type analysis [4] を利用する. この方法によって, 我々の束縛時検査は, コンパイル時間全体のおよそ 2–4%で終了する[4].

4.5.1 クラス集合の生成

Algorithm 1 集合 C_{comp} を求めるアルゴリズム

$\mathcal{M} \leftarrow \phi$ //検査済みのメソッド名の集合
$\mathcal{I} \leftarrow \{pcd\}$ //メソッド呼び出し式で使われた
　　メソッド名の集合
$C_{comp} \leftarrow \phi$ //new されるクラス名の集合
while $dispatch(\mathcal{I}, C_{comp}) \setminus \mathcal{M} \neq \phi$ **do**
　for all $c.m \in (dispatch(\mathcal{I}, C_{comp}) \setminus \mathcal{M})$ **do**
　　$\mathcal{M} \leftarrow \mathcal{M} \cup \{\text{c.m}\}$
　　$C_{comp} \leftarrow C_{comp} \cup \{c | s \in body(c.m) \wedge s = \text{new } c\}$
　　$\mathcal{I} \leftarrow \mathcal{I} \cup \{c.m | s \in body(c.m) \wedge s = \text{invk } c.m)\}$
　end for
end while

[3] JDK 1.5 の標準ライブラリにおける定義による.

[4] 6.2 節を参照

クラス集合の計算は, 関数呼び出しグラフの生成と平行して行う optimistic rapid type analysis [4] と同様にして行う. このアルゴリズムを Algorithm 1 に示す. ここで, $\mathcal{M}$ は既に検査済みのメソッドの集合を, $\mathcal{I}$ はメソッド呼び出し式において使用されたメソッド名の集合を表し, *pcd* はアスペクトをコンパイルした際にメソッド化された条件ポイントカットである. invk は invokestatic, invokespecial, invokevirtual または invokeinterface 命令を表す. また, $dispatch(\mathcal{I}, C_{comp})$ は現在のクラス集合とメソッド呼び出し式に用いられたメソッド名から, 検査すべきメソッド定義を取得する関数である.

$$dispatch(\mathcal{I}, C) = \bigcup_{c.m \in \mathcal{I}} calls(c.m, C)$$

$calls(c.m, C)$ は, メソッド $c.m$ がオーバーライド可能なメソッドであった場合は現在のクラス集合 C の下でオーバーライドしたメソッドを集め, そうでなければ $c.m$ をそのまま返す関数である:

$$calls(c.m, C) = \begin{cases} \{c'.m | c' \in C \wedge c' <: c \wedge m \in methods(c')\} & \text{if } virtual(c.m) \\ \{c.m\} & \text{otherwise} \end{cases}$$

$methods(c)$ はクラス c で定義されたメソッドの集合を表し, $virtual(c.m)$ は, メソッド $c.m$ がオーバーライド可能なメソッドかを調べる関数である.

4.5.2 条件ポイントカットの束縛時検査

条件ポイントカットの束縛時検査は, クラス集合 C_{comp} を計算する時に生成した $\mathcal{M}$ を用い, $\mathcal{M}$ に含まれるメソッドを, メソッド呼び出し式を無視して検査することで行う:

$$\mathcal{B}(\text{pcd}) = \sup\{\mathcal{B}(c.m) | c.m \in \mathcal{M}\}$$
$$\mathcal{B}(c.m) = \sup\{\mathcal{B}(s) | s \in body(c.m) \wedge s \neq \text{invk } c.m\}$$

上の定義は 4.4.2 で示した定義と等価ではないが, 条件ポイントカットの束縛時は, $C = C_{comp}$ とした時の 4.4.2 の計算結果と完全に一致する.

5 既存のバイトコード解析ライブラリを利用したポイントカットを実現する枠組み

この節では, プログラム解析の結果に基づくポイントカットを簡潔に定義でき, かつ定義されたポイントカットが SCoPE で静的に評価できるようにする枠組みの性質と構成, 振る舞いを述べる.

従来の SCoPE では Java Reflection API を使ってプログラムのクラスやフィールド, メソッドのような構造レベルの解析を行う条件ポイントカットを静的に評価することが可能だったが, 命令レベルの解析が必要な pcflow [12] や 3 節で述べたデメテル則を調べるポイントカットは定義できなかった. これは, Java Reflection API によって得られるプログラムの静的な性質が, バイトコードを解析して得られる情報のほんの一部に過ぎないことに依る. 我々は, Soot [17] や ASM [2] などのバイトコード解析ライブラリを使った静的条件ポイントカットの記述ができる枠組みを作ることによって, この問題を解決した. バイトコード解析ライブラリを用いることによって, ユーザーはプログラムを解析するメソッドを自由に定義することができ, その解析メソッドを用いた静的条件ポイントカットを記述することが可能となる. 例えば, 3 節における, デメテル則の条件 5 を検査するメソッド `isPreferredAquaintance` は, soot を用いると図 6 のように定義できる. 我々の枠組みは, このような関数を含むポイントカットも静的に評価できるよう SCoPE を拡張する.

5.1 枠組みの性質

5.1.1 互換性の確保

提案する枠組みは, abc [3] や ajc [13] などの既存の AspectJ 言語の処理系と SCoPE とで, 同じソースコードがコンパイルでき, 同じソースコードをコンパイルすると同じ振る舞いをするプログラムを生成することを実現する. このことは, SCoPE が ajc や abc に対して, 条件ポイントカットの評価に関する最適化処理が加わったコンパイラであるという前提を満たす上で重要な性質である. コンパイルされたプログラムが同じ振る舞いをすることは,

1. プログラムを解析するプログラムが同じである
2. 解析の対象となるプログラムが同じである

の 2 点を満たすことで保証される.

静的解析プログラムの一致 静的解析プログラム (図 1) は, SCoPE でコンパイルしたときと ajc や abc でコンパイルした時で変化することはない. これは, SCoPE が静的条件ポイントカットの評価を行う際に, 一度 ajc や abc と同じコンパイルをして Java バイトコードを得, そのバイトコードを用いて条件ポイントカットを評価するためである [1].

静的解析の対象となるプログラムの一致 提案する枠組みは, 静的解析の対象となるプログラムを, ユー

メソッド	動作
abstract void setPaths(String[])	ロードパスを設定する
abstract void loadClass(String)	指定されたクラスをロードする
abstract void loadClasses(String[])	指定されたクラス集合をロードする
abstract void loadAllRequiredClasses()	現在ロードされているクラスを含む推移的参照閉包を計算し, 含まれる全てのクラスをロードする
void initialize(Config)	Config 情報に基づきロードパスの設定, 推移的参照閉包の計算及びその閉包に含まれるクラスのロードを行う.
abstract String[] getBTCAvoids()	束縛時検査を回避するパッケージ名を取得する

表 1: abstract class Connector

```
boolean isPreferredAcquaintance(
  JoinPoint.StaticPart ejp, JoinPoint tjp){
 Type supplierT=
  Scene.v().getSootClass(
   getSupplierType(tjp).getName()
   ).getType(); /*メソッド呼び出しのターゲットオブジェクトの静的型を取得*/
 List preferredAcquaintances=new ArrayList(); /*new された class 名のリスト*/
 MethodSignature msig=
  (MethodSignature)ejp.getSignature(); /*現在実行しているメソッドのシグニチャ*/
 SootMethod method=
  Scene.v().getSootClass(msig.getDeclaringTypeName()).getMethod(
    msig.getName(),
    class2type(Arrays.asList(msig.getParameterTypes())));
 JimpleBody body=
  (JimpleBody)method.retrieveActiveBody(); /*現在実行しているメソッドの body*/
 Iterator it=body.getUnits().iterator(); /*メソッドの body を構成する式 s を逐次取り出す*/
 while(it.hasNext()){
  Object unit=it.next(); /*式 s を取り出す*/
  if(unit instanceof AssignStmt){ /* 式 s が代入式のとき */
   AssignStmt assign=(AssignStmt)unit;
   Value rightExpr=assign.getRightOp(); /*代入式 x=y の y の部分を取り出す*/
   if(rightExpr instanceof NewExpr){ /*代入式が x=new C であったとき*/
    if(((NewExpr)rightExpr).getType().equals(supplierT))
   return true; /*new C の C とターゲットオブジェクトの型が一致した*/
   }}}
 return false;
}
```

図 6: Soot を用いた, デメテル則の条件 5 を検査する関数の定義例

ザーが指定するエントリクラスからの依存関係によって得られる推移閉包に定める. バイトコード解析ライブラリを初期化する際, 推移閉包の計算に使われるプログラムは SCoPE がコンパイル時にポイントカットの評価を行う時と, ajc でコンパイルしたプログラムを実行する時とで変化しない. このため, 同じエントリクラスが与えられた時, 図 1 における解析プログラムの解析対象は一致する.

5.1.2 解析メソッドの独立

提案する枠組みは, プログラムを静的に解析するメソッドを実装する者から, ロードすべきクラスの計算や, クラスをロードする作業を隠蔽する. このため, 解析メソッドの定義を, バイトコード解析ライブラリを使用するための前処理から切り離して定義することが可能となった. このことは, 静的解析メソッドを実装する者の労力を減らす. また, 定義された静的解析メソッドをライブラリ化して再利用することが容易となる.

5.2 枠組みの構成

提案する枠組みは, バイトコード解析ライブラリを初期化するためのプログラム (Connector) と, ajc でコンパイルする際に枠組みとコンパイルされるプログラムとを結びつけるプログラム (InitializingAspect), そして設定ファイルを読み込んで初期化プログラムに送信するプログラム (Initializer) からなる. これらはそれぞれ, バイトコード解析ライブラリを提案する枠組みに適用する者, プログラム解析を利用するポイントカットを利用する者, そして枠組みの提案者の 3 者によって実装される.

5.2.1 Connector

Connector(表 1) は, バイトコード解析ライブラリ固有のプログラムである. 我々の提案する枠組みに新たなバイトコード解析ライブラリを適用する者が, この Connector を実装する.

Connector は対応するライブラリに対して, (1) クラスファイルのロードパスを指定したり必要なクラスをロードする初期化のためのメソッドと, (2) 束縛時検査器に対してバイトコード解析ライブラリを構成するパッケージ名を提示するための束縛時検査の回避のためのメソッドを定義する.

束縛時検査の回避のためのメソッドは, バイトコード操作ライブラリを利用した静的解析プログラムを条件ポイントカットの内部で用いることができるようにする. 一般に, バイトコード解析ライブラリがロードしたクラスファイルのキャッシュ等のために大域変数を使用する. このため, これを用いた条件ポイントカットが SCoPE の束縛時検査器を通らなくなってしまう. 我々の拡張は, Connector から提示されるパッケージに含まれるクラスに対して束縛時検査を行わないようにすることで, この問題を解決した.

```
1  abstract aspect InitializingAspect{
2   private static Reset RESET=new Reset();
3   abstract pointcut entry_point();
4   before(): entry_point(){
5    RESET.run();
6   }
7  }
```

図 7: InitializingAspect のコード

```
1  pointcut entry_point():
2   execution(void *.main(String[]));
```

図 8: main 関数の実行を entry_point として指定するポイントカット

5.2.2 InitializingAspect

InitializingAspect(図 7) は, プログラムの実行時にバイトコード解析ライブラリを初期化するようなアドバイスを定義する抽象アスペクトである. このアスペクトは, `entry_point` に適合するジョインポイントの直前に Initializer を呼び出し, バイトコード解析ライブラリの初期化を行う.

静的解析ライブラリを利用したアスペクト (図 1) を利用するユーザーは, 自分のコンパイルするプログラムが起動するジョインポイントに適合するようなポイントカット (`entry_point`) を定義しなければならない. 例えば, コンパイルするプログラムの中に main 関数を実装するクラスがあり, これがプログラムの実行のエントリとなる場合には, `entry_point` を main 関数の実行を示すポイントカット (図 8) と定義すればよい.

5.2.3 Initializer

Initializer は各バイトコード解析ライブラリに対応して存在する Connector と, InitializingAspect または

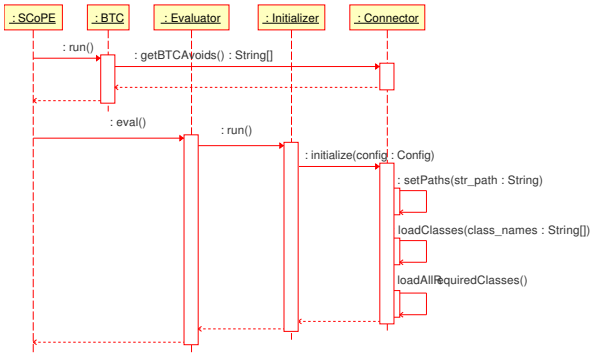

図 9: SCoPE と枠組みの振る舞い

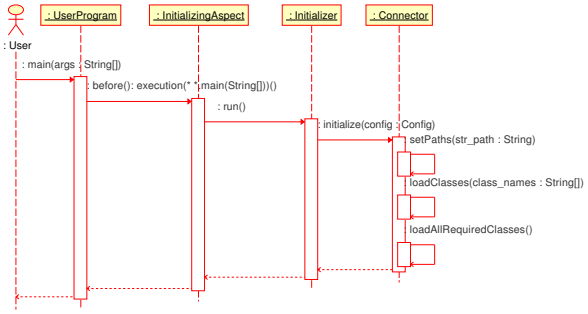

図 10: ajc でコンパイルされたプログラムと枠組みの振る舞い

SCoPE を結ぶプログラムであり, 提案する枠組みの一部として我々が実装し, 提供する.

Initializer はユーザーから与えられる設定ファイルを読み込み, その情報を下に適切な Connector を取り出して, これに設定情報を送信する. 設定情報はクラスのパスと必ずロードされるべきクラス, そして用いるバイトコード解析ライブラリの名前からなる.

5.3 枠組みの振る舞い

5.3.1 SCoPE との連携

SCoPE と提案する枠組みは, 以下のようにして協調動作する (図 9).

1. SCoPE がアスペクトを織り込んだコンパイル済みのバイトコードに対して束縛時検査器 (BTC) を呼び出す.
2. 呼び出された束縛時検査器は, Connector から `getBTCAvoids()` を通して取得できるパッケージに含まれないクラスについて, 束縛時検査を行う.
3. 束縛時が S であった条件ポイントカットを評価するために, SCoPE が静的条件ポイントカットの評価器 (Evaluator) を呼び出す.
4. 呼び出された評価器は, ユーザーから与えられた設定ファイル情報を Initializer に渡す.
5. Initializer は設定情報 (Config) に基づいて使われるバイトコード解析ライブラリに対応する専用の Connector を呼び出す.
6. Connector がライブラリを初期化する.

5.3.2 ajc でコンパイルされるプログラムとの連携

ユーザープログラムは, InitializingAspect と共に ajc でコンパイルすることで, 以下のように提案する枠組みと協調動作する (図 10). ここでは, `entry_point` を, main 関数の実行を表すジョインポイントに適合するポイントカット (図 8) を用いるものとする.

1. InitializingAspect 内に定義された before アドバイスによって, main 関数の実行の直前に Initializer を呼び出す.
2. Initializer がユーザーから与えられる設定情報を読み込み, それに従って適切な Connector を取得し, initialize メソッドを呼び出す.
3. Connector が initialize メソッドを通じて渡された設定情報に基づき, 対応するバイトコード解析ライブラリを初期化する.

6 評価

本節では, 本研究における拡張によって記述できるようになった静的条件ポイントカットの例を挙げると共に, 束縛時検査が高速に行えることを示す.

6.1 適用事例

提案する拡張によって, SCoPE でもプログラムの静的解析を利用するポイントカットが, コンパイル時に評価できるようになった. このようなポイントカットの例としては, 以下のようなものが挙げられる.

- デメテル則を検査するポイントカット (約 190 行)
- コントロールフローを静的に解析する pcflow ポイントカット [12](約 130 行)

stage		Monte Calro	Molecular	Search	Euler
1st	compile	30.01	23.18	21.33	32.91
	btc	0.75	0.79	0.96	0.68
	eval	0.32	0.17	0.15	0.24
2nd		10.45	7.84	6.66	15.23

表 2: コンパイル時間とその内訳 (単位は秒)

6.2 束縛時検査の高速性の検証

提案する方法によって, 束縛時検査が高速に行えることを示す. 我々は, abc [3] を拡張して実装した SCoPE コンパイラを用いて, 3 節で示したデメテル則の違反を発見するアスペクトと複数のアプリケーションプログラムと共にコンパイルした時のコンパイル時間を測定した, アプリケーションは Java Grande Forum が提供する sequential benchmarks から, Search, Euler, MD(Molecular Dynamics) および MC(Monte Calro) の 4 つのプログラムを使用した. 計測は Xeon 3.06GHz を 2 基, メモリ 6GB を搭載したマシンで行い, Java 環境としては JDK1.5.0 update 4 を利用した. この結果を表 2 に示す. 束縛時検査のために費される時間は約 0.68–0.96 秒であり, 第 1 コンパイルのおよそ 2–4%, 全コンパイル時間の約 1.4–3.3%の時間で終了する.

今回の計測においては, アスペクトを織り込む対象となるベースプログラムが小さかったが, 束縛時検査に掛かる時間は, ベースプログラムの複雑さには依存しない. より規模の大きなプログラムでは, 束縛時検査にかかる時間の比率は, さらに小さくなるだろう.

7 関連研究

Kiczales は静的にコントロールフローを調べる pcflow ポイントカットを提案した [12]. また, Burke はクラスに特定のメソッドやフィールドが定義されていることを調べる has/hasfield ポイントカット[5]を提案した [6]. これらは個別のポイントカットとして提案されたが, SCoPE を用いれば, 条件ポイントカットと自己反映計算及びバイトコード解析ライブラリによって同等のポイントカットが定義できる.

特別なポイントカットを逐次追加してゆくのではなく, 新しいポイントカットをユーザーが定義できる機構の研究としては, Josh [7] や LogicAJ [9], 関数型クエリ言語やロジックメタプログラミングを用いたポイントカット記述 [8, 18] などがある. これらは, ポイントカットを拡張できる点で SCoPE と方向性を同じくするが, ポイントカットの定義のために XQuery や Prolog などの既存の言語や全く独自の言語を用いている点で, SCoPE とは異なる. SCoPE では, AspectJ 言語以外の言語を必要としない.

[5]JBossAOP や AspectWerkz に実装されている

8 まとめと今後の課題

本稿において我々は, 条件ポイントカットと自己反映計算を用いたポイントカットをコンパイルする枠組み SCoPE に対して, ユーザー定義のプログラム解析を用いたポイントカットも静的に評価できるようにする拡張を提案した. この拡張は束縛時検査の改善と, バイトコード解析ライブラリとの円滑な連携を実現する枠組みの提供からなる. この拡張によって, デメテル則の完全な検査を行うポイントカットや, pcflow ポイントカットが静的に評価できるようになり, より表現力の高いポイントカットの最適化が可能となった. また, この拡張は, 様々なバイトコード解析ライブラリを容易に取り込むことができ, かつプログラム解析を実装するプログラマからクラスファイルのロードや解析対象となるプログラムの特定といった面倒な作業を隠蔽する.

この研究の進展方向としては, 以下の 2 つを考えている.

- ユーザーによって定義された新たなポイントカットを条件ポイントカット `if(e)` の形で記述するのではなく, 直接 `e` と書けるようにする. これにより, 新たなポイントカットを定義するということが明確化される.
- ユーザーによって定義される新たなポイントカットが, ポイントカットを引数に取ることができるようにする. このようにすることで, 新たに定義したポイントカットのモジュール性が向上すると期待する.

謝辞

本稿の作成にあたって, 役立つ助言と訂正を下さった PPP の面々に感謝致します.

参考文献

[1] T. Aotani and H. Masuhara. Compiling conditional pointcuts for user-level semantic pointcuts. In *SPLAT workshop at the 4th international conference on Aspect-Oriented Software Development (AOSD.05)*, 2005.

[2] ASM. http://asm.objectweb.org/.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In P. Tarr, editor, *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM Press.

[5] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 2–11. ACM Press, 1999.

[6] B. Burke. http://docs.jboss.org/.

[7] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. In G. C. Murphy and K. J. Lieberherr, editors, *AOSD*, pages 102–111, 2004.

[8] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *APLAS'04*, Lecture Notes in Computer Science, pages 366–382, 2004.

[9] S. H. Günter Kniesel, Tobias Rho. Evolvable pattern implementations need generic aspects. In *Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*. June 2004.

[10] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proc. of AOSD'03*, pages 60–69. ACM Press, 2003.

[11] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[12] G. Kiczales. The fun has just begun, 2003. Keynote talk at AOSD'03.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[14] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In *Proc. of AOSD'03*, pages 40–49. ACM Press, 2003.

[15] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[16] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proc. of CC2003*, pages 46–60, 2003.

[17] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java bytecode optimization framework, 1999.

[18] K. D. Volder. Aspect-oriented logic meta programming. In *Workshop ion on Object-Oriented Technology*, pages 414–417. Springer-Verlag, 1998.