情報処理学会論文誌　プログラミング　Vol. 4　No. 2　67–76 (Mar. 2011)

2つのメソッド呼び出しに関わる最適化を可能にするアスペクト指向言語

伊尾木　将之[†1]　　千　葉　　滋[†1]

本稿では，2つのメソッド呼び出しを1つのアドバイスで置き換え可能なアスペクト指向言語を提案する．2つのメソッド呼び出しに関わる関心事として，たとえば2回のデータベース（DB）アクセスを1回に減らすなどの，DBの最適化処理がある．しかし，2つのメソッド呼び出しの間に他のコードが存在し，そのコードと2つのメソッド呼び出しの間に依存関係がある場合，既存のアスペクト指向言語ではうまく対応することができない．本稿が提案するアスペクト指向言語 Dualcut では，そのような場合であっても，2つのメソッド呼び出しと間のコードが一定の条件を満たせば，1つのアドバイスで置き換えることが可能である．本稿では，その条件を先頭末尾連続可能と呼ぶ．

An Aspect-oriented Language for Optimization over Two Method Calls

Masayuki Ioki[†1] and Shigeru Chiba[†1]

This paper proposes an aspect-oriented language named Dualcut. Dualcut can substitute an advice for two method calls. An example of concern cutting across two method calls is optimization, which replaces two database (DB) accesses with one merged access. Existing aspect-oriented languages cannot allow such optimization. If two DB accesses (i.e., method calls) are not adjacent but surround a code fragment depending on those accesses, existing aspect-oriented languages do not allow developers to write a single advice substituting for the two accesses. On the other hand, Dualcut allow them if the code fragment satisfies a certain condition.

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

1. はじめに

既存のアスペクト指向言語では，メソッド呼び出しなどの実行点（ジョインポイント）の振舞いを変更することができ，典型的な応用例として最適化の実施が知られている．しかし，既存のアスペクト指向言語が扱えるジョインポイントは1度に1つだけであり，そのため2つのメソッド呼び出しが関わるような最適化を実施することは難しい．

既存のアスペクト指向言語では対応が難しい最適化例として，2回のデータベース（DB）アクセスを1回に減らす最適化が考えられる．DBアクセスにDAO（Data Access Object）コンポーネントを用いた場合，2回のDBアクセスはDAOメソッドの2回の呼び出しになる．DBアクセスを1回に減らす最適化とは，2つのDAOメソッド呼び出しで実施する処理を1つのDAOメソッド呼び出しに置き換えることである．しかし，このような最適化は，既存のアスペクト指向言語ではうまく実装できない．最適化対象の2つのメソッド呼び出しの間に他のコードが存在し，それらのコードと2つのメソッド呼び出しが依存関係を持つ場合があるからである．

本稿では，2つのメソッド呼び出しを1つのアドバイスで置き換え可能なアスペクト指向言語としてDualcutを提案する．DualcutはAspectJ[1)]など既存のアスペクト指向言語では対処できない場合でも，一定の条件を満たしていれば，2つのメソッド呼び出しだけをアドバイスで置き換えることができる．本稿では，この条件を先頭末尾連続可能と呼んでいる．先頭末尾連続可能とは，プログラムの意味を保存したまま2つのメソッド呼び出しを連続して実行（順次実行）することが可能かどうかを判断する条件である．

以下では，2章で1度に2つのメソッド呼び出しを扱えない既存のアスペクト指向言語の問題を指摘し，3章でDualcutの概要を示す．4章では先頭末尾連続可能について述べ，5章で実験について述べる．そして6章で関連研究について述べ，7章でまとめを述べる．

2. Motivating Example

2つのメソッド呼び出しに関わる最適化の例として，2つのテーブルを検索する処理を考える．また，DBアクセスにはDAO（Data Access Object）の利用を想定する．DAOとは，DBなどの永続化機構に対するアクセスを抽象化したAPIを提供するデザインパターンである．DAOをサポートするツールは数多く存在し（Hibernate[2)]，iBatis[3)]など），WEBアプリケーションなどDBを使用するアプリケーションで利用されることが多い．

今，**図1**のようにDBに部署表（Dept表）と従業員表（Emp表）が存在し，これらの表

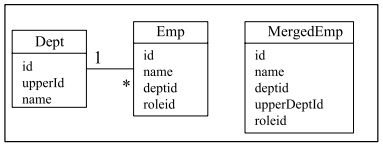

図 1　テーブル構造
Fig. 1　Table schema.

リスト 1　最適化対象コード
List 1　DB access code.

```
1  void members(int deptId,DeptDao deptDao,EmpDao empDao, String roleName) {
2    Dept[] depts=deptDao.selectUnderBy(deptId);
3    String logMessage="depts count: "+depts.length;
4    int roleId = Util.nameToId(roleName);
5    Emp[] emps=empDao.select(depts,roleId);
6    Log.log(logMessage);
7    return emps;
8  }
```

から指定部門の部署に所属する指定ロールの従業員一覧を取得したいとする．具体的には，Dept 表から指定された部門の部署一覧を取得し，次に，先ほど取得した部署一覧に所属し，かつ指定されたロールに該当する従業員一覧を Emp 表から取得する．これを実行しているのが**リスト 1** のコードである．リスト 1 中の，deptDao，empDao は，それぞれ Dept 表，Emp 表へのアクセスを抽象化した DAO であり，deptDao の selectUnderBy メソッドから指定部門の部署一覧を取得し，empDao の select メソッドから目的の従業員一覧を取得している．また，empDao の select 時には 4 行目で得られた roleId を使用している．

以上の処理に対して，2 回の DB アクセスを 1 回の DB アクセスで置き換える最適化を実施する．具体的には，Dept 表と Emp 表を融合した読み込み専用の表（MergedEmp 表）を用意しておき，この MergedEmp 表を 1 回検索することで同等の結果を得るようにする．コードとしては，MergedEmpDao を作成し，DeptDao と EmpDao へのメソッド呼び出しを MergedEmpDao への select メソッドの呼び出しに置き換える．このような最適化は，アプリケーション開発・保守の現場ではよく行われるが，どのような最適化が良いかはアプリケーションのデータ量や利用状況によって異なるため，随時最適化処理を変更できるほうが良い．このような処理には，アスペクト指向のように最適化処理を容易に追加・削除可能な機構が望ましい．

しかし，2 つのメソッドが関わる最適化を既存のアスペクト指向言語で行うことは難しい．たとえばリスト 1 のコードに対して，AspectJ を使って最適化を行うとした場合，2 行目の DeptDao と 5 行目の EmpDao のメソッド呼び出しをそれぞれポイントカットし，どちらかのポイントカットで MergedEmpDao のメソッド呼び出しを行うことになるだろう．しかし，2 つのメソッド呼び出しの間のコードが問題になり，どちらの場合もうまくいかない．仮に 2 行目の DeptDao のメソッド呼び出しを MergedEmpDao へのメソッド呼び出しに置き換えると，検索の条件の 1 つである roleId を知ることができないため，実際には検索処理を実行することができない．また，5 行目の EmpDao のメソッド呼び出しを MergedEmpDao へのメソッド呼び出しに置き換えると，3 行目のログメッセージの構築処理を実行することができない．このように 2 つのメソッド呼び出しとその間のコードに依存関係が存在する場合，既存のアスペクト指向言語ではうまく対応することができない．

3. Daulcut

我々は，2 つのメソッド呼び出しに対して 1 つのアドバイスで置き換え可能なアスペクト指向言語として，Daulcut を開発した．

3.1 概　　要

Dualcut では，連続すると見なせる 2 つのジョインポイントを 1 つのアドバイスで置き換えることができる．連続すると見なせるという条件を「先頭末尾連続可能」と我々は呼んでおり，この条件に関しては次章で詳しく述べるが，直感的には次のような意味である．つまり，2 つのジョインポイントがコード上並んで実行（順次実行）されているか，あるいは 2 つのジョインポイントの間に他のコードがある場合，プログラムの意味を保存したままコード移動を行って 2 つのジョインポイントが並ぶような変換結果が存在する場合を連続と見なすという条件である．また，Dualcut で扱うジョインポイントはメソッド呼び出しだけであり，これ以降説明のしやすさから，特に断らない限り「ジョインポイント」よりも「メソッド呼び出し」と明示的に記述する．

具体的には，次のような挙動をとる．2 つのメソッド呼び出しが順次実行のように並ぶ場合，その 2 つのメソッド呼び出しが指定された 1 つのアドバイスに置き換られる．2 つのメソッド呼び出しが並んでいない場合，2 つのメソッド呼び出しとその間のコードを 1 つのコード領域として，まずこのコード領域が先頭末尾連続可能を満たすかどうかをチェックされる．コード領域が先頭末尾連続可能を満たさない場合，アドバイスの置き換えは実施せず，満たす場合のみこのコード領域を指定されたアドバイスで置き換える．ただし，一般

にアドバイスは置き換えたい 2 つのメソッド呼び出しに相当する処理しか実行しないため，そのコード領域全体をそのままアドバイスで置き換えるとプログラムの意味が変わってしまう．そこで，プログラムの意味が変わらないようにアドバイスの前後に 2 つのメソッド呼び出しに囲まれるコードと同等の処理を追加し，アドバイスを拡張する．そして，この拡張されたアドバイスを用いてコード領域を置き換える．

先の 2 章の例では，2 行目の DeptDao のメソッド呼び出しと 5 行目の EmpDao のメソッド呼び出しで囲まれるコード領域（2 行目から 5 行目までのコード列）に対して先頭末尾連続可能がチェックされる．満たしていると判断されると，3 行目のログメッセージ構築および 4 行目の roleId 取得処理がアドバイスの前後に追加される．roleId 取得はアドバイスの前，ログメッセージ構築処理はアドバイスの後に実行されるようにアドバイスが拡張される．最後に，この拡張されたアドバイスで 2 行目から 5 行目までのコード領域を置き換える．実際にアドバイスを拡張して正しく置換できるかを判定するのは，最初の先頭末尾連続可能のチェックである．後で述べるように，この条件が満たされれば，つねに置換が可能である．

3.2 仕　　様

Dualcut でのアスペクトの例を**リスト 2** に示す．このアスペクトでは 2 つのポイントカット（selectDept と selectEmp）を定義し，1 つのアドバイス（optimize）を定義している．Dualcut で定義できるポイントカットは，AspectJ の call に相当し，ポイントカット定義の「：」より右辺でどのメソッド呼び出しをポイントカットするのか指定している．また，左辺の pointcut からポイントカット名までの「noex」や「nose」などはポイントカットに対してユーザが表明することのできるユーザヒントである．表明することが可能なユーザヒントは以下の 4 つである．詳しくは後述するが，これらの情報は先頭末尾連続可能の判定時に使用される．

(1)　メソッド呼び出しで例外発生がないことの表明（noex）
(2)　メソッド呼び出しで副作用がないことの表明（nose）
(3)　メソッド呼び出しの実行結果が引数にのみ依存することの表明（argsonly）
(4)　メソッド呼び出しの実行結果が null ではない表明（notnull）

Dualcut のアドバイスはユーザの指定した 2 つのメソッド呼び出しに対して適用される．そのためアドバイスは，アドバイス名に続く括弧内に 2 つのポイントカットをとる．ポイントカットは先頭，末尾の順に書く．例の場合，selectDept が先に現れるポイントカットで，selectEmp がその後に現れるポイントカットである．アドバイスは，2 つのメソッド呼び出しを置き換えるため，2 つのメソッドと同様の返り値をアドバイスで返す必要がある．そのため，Dualcut のアドバイスの return 文は返り値が 1 つではなく 2 つである．その順番はポイントカットの順番と一致させる．また，ポイントカットそれぞれの実行点でのコンテキスト情報（引数の値や，メソッドが呼び出されるオブジェクト）は，ポイントカット名に .target，.args をつけることによって参照できる．リスト 2 では，7 行目と 8 行目で引数の値を取得している．たとえば selectDept.args[0] はそのポイントカットが選択するメソッドの第 1 引数を表す．Dualcut では AspectJ の proceed 相当の機能は提供しないが，このコンテキスト情報から proceed に近い処理を実行することができる．

リスト 2　アスペクト定義
List 2　Aspect.

```
1  aspect DaoOptimize {
2    pointcut noex nose argsonly notnull selectDept : Dept[] DeptDao.selectUnderBy(int);
3    pointcut noex nose argsonly selectEmp : Emp[] EmpDao.select(Dept[], int);
4
5    advice optimize(selectDept , selectEmp){
6      MergedEmpDao mergedDao = new MergedEmpDao();
7      int upperDeptId=(Integer)selectDept.args[0];
8      int roleId=(Integer)selectEmp.args[1];
9      MergedEmp[] mergedEmps = mergedDao.select(upperDeptId,roleId);
10     Dept[] depts; Emp[] emps;
11     for(MergedEmp mergedEmp : mergedEmps){
12       // depts と emps の構築
13     }
14     return depts,emps;
15   }
16 }
```

3.3 実　　装

Dualcut は，JasstAddJ[4] を用いて実装された独自の Java 風のアスペクト指向言語である．現在の実装では，アドバイスの織り込み対象となるコード領域に対して，1 つ目のメソッド呼び出しと 2 つ目のメソッド呼び出しが順次実行となるように領域内のコードの配置変えを行う．そして隣り合った 2 つメソッド呼び出しのコードをアドバイスの呼び出しコードに置き換える．また，実装は Java のサブセットをもとにしており，**図 2** に示すように文として変数宣言文，空文，if 文，return 文などを許し，式は 3 項演算子などを禁止する．

2 つのメソッド呼び出しのうち，1 つ目のメソッド呼び出しを H，2 つ目のメソッド呼び出しを T，H と T の間の領域内のコードの各文を S として以下に示す規則でコードの配置換えを行う．この規則は，H を後ろに下げる規則と T を前に上げる規則に大きく 2 つに分

文　$S :=$;(空文) | E; | return E; | $T\ \mathbf{v} = E$; | if(E){ S+ }else{ S+ } .
式　$E := O_1\ E \mid E\ O_2\ E \mid$ new T((E (, E)*)?)
　　| E.**method**((E (, E)*)?) | E.**field** | $\mathbf{v} = E \mid V$.
$V := L$(リテラル) | **v**(変数) | T(型) .
ただし，O_1 は単項演算子，O_2 は 2 項演算子.

図 2　Java のサブセット
Fig. 2　Subset of Java.

かれる．そして，この 2 つの規則を交互に適用していき，H と T が隣り合うか，あるいは H/T がメソッドブロックの末端に到達し，どの規則でも H/T を動かせないときに終了する．この 2 つは対照的な規則であるため，ここでは H を後に下げる規則のみを示す．また，規則の条件に現れる *pdg* 関数は先頭末尾連続可能の判定時に作られる PDG[5]（Program Dependence Graph：プログラムの依存関係を表現するグラフ）上のパスの集合を返す関数である．H を後に下げるには以下の規則を終了するまで繰り返し適用する．

$$\frac{\Gamma \vdash pdg(H,S) = \phi}{\Gamma \vdash \cdots\ H, S \cdots \longrightarrow \Gamma \vdash \cdots\ S, H\ \cdots}.$$

$$\frac{\Gamma \vdash pdg(H,S) \neq \phi, \quad H' : H, S}{\Gamma \vdash \cdots\ H, S\ \cdots \longrightarrow \Gamma \vdash \cdots\ H'\ \cdots}.$$

最初の規則は，H と次の文 S との間に依存関係がない場合，H と S を交換する．次の規則は，依存関係がある場合，S を H に追加し新たな H（規則中では H'）と見なすことを示す．また，この H' は T の規則の適用時にはグループ化が解除され，元の H，S に戻される．次に文 S が if 文である場合の規則を示す．

$$\frac{\Gamma \vdash pdg(H,E) = \phi, \quad S : if(E)\{\cdots\} else\{\cdots\}}{\Gamma \vdash \cdots\ H, S, \cdots \longrightarrow \Gamma \vdash \cdots if(E)\{H, \cdots\} else\{H, \cdots\}, \cdots}.$$

H と if 文の条件式に依存関係がない場合，if 節と else 節に H を挿入する．H と if 文の条件式に依存関係がある場合，先ほどと同様 H を拡張して，新たな H を考える規則を使う．H が if 文を抜け出すには以下の規則で行う．

$$\frac{\Gamma \vdash H_3 : if(f)\{H_1\} else\{H_2\}}{\Gamma \vdash \cdots\ if(E)\{\cdots H_1\} else\{\cdots H_2\} \cdots \longrightarrow \Gamma \vdash \cdots boolean\ f; if(f = E)\{\cdots\} else\{\cdots\}, H_3 \cdots}.$$

if 節と else 節の両方の末尾に H がある場合，新たな if 文を作成し，その if 文を H（規則中では H_3）と見なす規則である．また，H が if 節にしかない場合は else 節の末尾に空文の H が存在すると見なして同様の規則を行う．else 節にしか H がない場合も同様である．また，先ほどの H' と同様に H_3 は T の規則の適用時にはグループ化が解除される．

最後の if 文を抜け出す規則は必須ではない．変換規則全体の見通しを良くするために導入しているだけであり，最初の 3 つの規則の H を下に下げる規則と T を上に上げる規則だけでもよい．

3.4 制　　限

Dualcut は，図 2 の Java のサブセット内で記述されたプログラムに対して，先頭末尾連続可能の条件を判定し，アドバイスの置き換えを行う．図 2 の Java のサブセット以外の文法を使用しているプログラムに対しては，先頭末尾連続可能の条件が成り立たないと判断し無視する．これは，現在の先頭末尾連続可能がループや try-catch などの構文に対応していないためである．

2 つのメソッド呼び出しの間のコードが，実行時間やメモリ使用量の計測処理を含んでいた場合，ユーザ側で注意する必要がある．たとえば，次のような foo メソッドの実行時間を start，end メソッドのペアで計測するコードがあったとする．v=H();start();foo(v);end();T(). この場合，H，T メソッドと start，end メソッドの間に依存関係がなくさらに foo，T メソッド間にも依存関係がないと，Dualcut によって start();v=H();T();foo(v);end(); のように変換され，H，T の並びがユーザ指定のアドバイスで置き換えられる．このような場合，foo メソッドだけの実行時間を計測できない．これは，先頭末尾連続可能の条件の判定時に実行時間やメモリ使用量に関する依存関係を判断できないからである．仮に Dualcut で実行時間やメモリ使用量に関する依存関係をユーザが直接指定するような機能を提供すれば，この限界は解消されうる．しかし，最適化ではそもそも実行時間やメモリ使用量の変更を意図するため，この制限を解消するための機能を提供していない．

4. 先頭末尾連続可能

Dualcut では，2 つのメソッド呼び出しとその間のコードが先頭末尾連続可能を満たす場合のみ，アドバイスの呼び出しで置換する．ここで，先頭末尾連続可能の定義を示す．先ほどと同様に，ユーザの指定する 2 つのメソッド呼び出しを，それぞれ 1 つ目を H，2 つ目を T とおく．また，プログラムの CFG（Control Flow Graph）と PDG（Program Dependence Graph）が作成されているとする．CFG は，プログラムの制御の流れを表現した有向グラフであり，これに対し PDG は，プログラムの依存関係を表現した有向グラフである．

定義 1　プログラムの CFG と PDG があって
CFG 上 H から T への空でないパスが存在し，かつ

PDG 上 H から T へのパスが空であるか，または
空でない場合その距離が 1 であるとき H と T が先頭末尾連続可能という．

CFG についての条件は単純に，H の後に T が実行される可能性を述べている．PDG についての条件は，H と T がまったくの無関係であるか，あるいは，H と T の間に依存関係が存在する場合，直接的な依存関係しかないということを述べている．

この条件が満たされる場合，H と T で挟まれたコードは，H の前かあるいは T の後で実行してもよいコードだけである．そのため，アドバイスの前後に間のコードと同様の処理を追加した拡張アドバイスを作ることがつねに可能である．

4.1 CFG と PDG のノード

Dualcut が用いる CFG と PDG のノードについて述べる．Dualcut が用いる CFG と PDG では，1 文を 1 ノードと見なす．ただし，以下に示す正規化が実施された後のプログラムの文をノードとする．また，PDG に対してはさらに「変数が陽に現れない形式」という形式に変換したプログラムの文をノードとする．

4.1.1 正　規　化

Dualcut は 2 つのメソッド呼び出しを扱うため，1 つのノードに 1 つのメソッド呼び出ししか含まないような形式が適している．そこでプログラムを以下のような制限が課せられた，より単純な文しか持たない形式に正規化する．具体的には**図 3** のような文法のみを許す形式に変換する．正規化を実施すると，たとえば，メソッド呼び出しの引数部分にメソッド呼び出しが存在するような文や，メソッドチェーンを用いるような文は複数の文に変換される．

4.1.2 変数が陽に現れない形式

変数が陽に現れない形式とは，プログラムの中から変数がすべて削除された形式である．これは，文間の依存関係を解析しやすくするためと余分なデータ依存関係を排除するために実施する．変数をすべて削除するため，ローカル変数の別名などを意識する必要がなくなる．また，余分なデータ依存関係とは，変数によって引き起こされる関係であり，たとえば**リスト 3** の 2 行目と 3 行目に存在するデータ依存である．このデータ依存は定数伝播によって削除可能であり，リスト 3 を**リスト 4** のように変換しておくと，2 行目と 3 行目のデータ依存関係をなくすことができる．

Dualcut では「変数が陽に現れない形式」を得るために定数伝播をすべての変数に適用する．しかし，素朴な定数伝播ではすべての変数を削除できない．分岐によって値が変わる変数と，メソッド呼び出しなどのように副作用をともなって値が決まる変数は削除できない．

文　S' := ;(空文) | E'; | return **v**; | T **v** = E'; | if(**v**){ S'+ }else{ S'+ } | **v** = E'; .
式　E' := O_1 V | V O_2 V | new T((V (, V)*)?)
　　| V.**method**((V (, V)*)?) | V.**field** | V .
　　V := L(リテラル) | **v**(変数) | T(型) .
　　ただし，O_1 は単項演算子，O_2 は 2 項演算子.

図 3　正規化後の文と式
Fig. 3　Normalized syntax.

リスト 3　コード例 1
List 3　Code 1.

```
1  int x = foo();
2  int y = x + 1;
3  bar(y);
```

リスト 4　コード例 2
List 4　Code 2.

```
1  int x = foo();
2  int y = x + 1;
3  bar(x+1);
```

分岐によって値が変わる変数は，まず SSA（静的単一代入）形式[6] に変換する．SSA 形式とは，変数への代入がコード上 1 度しか現れないプログラム形式であり，コンパイラ内部のコード表現としてよく使用される．SSA 形式では，分岐によって格納する値が変わる変数に対してはファイ関数が適用される．ファイ関数は，分岐で得られる可能性のある値すべてを保持する関数で，評価する際には通ってきたパスに該当する値を返す．つまり，ファイ関数自身に分岐の情報を保持させていることになり，このため定数伝播を行う際に分岐の影響を無視することができる．

次に，副作用などをともなって値が決まる変数は，デルタ関数とイプシロン関数という 2 つの関数を適用する．これは我々が独自に導入した関数である．デルタ関数とは，引数に式をとり，その式の結果をキャッシュして返す関数であり，イプシロン関数はデルタ関数の式を実行する関数である．先のリスト 3 に対してイプシロン関数とデルタ関数を適用した結果が**リスト 5** である．

まずリスト 3 の 2 行目の変数 y は，単純な定数伝播によって不要になるので 2 行目そのものを削除する．次に 1 行目にデルタ関数とイプシロン関数を使用する．デルタ関数の引数

リスト 5　デルタ関数・イプシロン関数が適用されたコード
List 5　Applying delta function and epsilon function.

```
1  ε (δ (foo()));
2
3  bar(δ (foo())+1);
```

は変数 x に代入されていた式であるが，デルタ関数は式を評価しない．デルタ関数の式を評価するのは，イプシロン関数である．デルタ関数は評価した結果をキャッシュし，次に同じデルタ関数が呼ばれた際にこのキャッシュした値が返される．つまり，1 行目のイプシロン関数で foo メソッドが評価され，その値がデルタ関数にキャッシュされ，3 行目ではその値が使用される．また，メソッドの仮引数とフィールドは，デルタ関数の引数にその変数自身が入れられる．

このデルタ関数の返す値がオブジェクトである場合，オブジェクトのうちのフィールドが変更される場合がある．このため厳密には定数ではないが，定数であると仮定して変数をデルタ関数で置き換えてもプログラムの意味は変わらない．また，同一メソッドの呼び出しを区別するためにデルタ関数に番号を振っておく．たとえば foo メソッドが 2 回呼ばれ，それぞれ 2 つの変数に代入されている場合，foo メソッドを持つ 2 つのデルタ関数で置き換えることになる．しかし，δ (foo()) のままではどちらの変数を置き換えているのか不明であるため，δ_0(foo()), δ_1(foo()) のように番号を振って識別する．ただし，本稿中では特に断らない限りこの番号は明示的には記述しない．

4.2　PDG

PDG で表現する依存関係は制御依存関係，データ依存関係，例外依存関係である．PDG は，まず CFG を作成し変数が陽に現れない形式に変換を行い，グラフに依存関係を表すパスを追加していく．最後にそのグラフから依存関係のパス以外を削除して，PDG を得る．

具体的には次のような手順で PDG を構築する．まず「データ依存関係」「例外依存関係」を調べるために，ノードに**表 1** のようなラベルを付ける．たとえば先のリスト 5 では，1 行目の文のノードはイプシロンノードと副作用ノード，例外ノード，ミュータブルノードのラベルが付けられる．そして，3 行目の文のノードにはデルタノードと副作用ノード，例外ノード，ミュータブルノードのラベルが付けられる．

次に，ラベル付けされたノード間に**表 2** の依存関係を追加する．イプシロン，デルタノード間に追加される依存関係はデータ依存 DataWriteRead（DWR）である．これはイプシロンノードでキャッシュされた値をデルタノードで使用するためである．副作用ノードから

表 1　ノード種別
Table 1　Kinds of node.

ラベル名	内容
イプシロンノード（ε）	イプシロン関数が使用されているノード
デルタノード（δ）	デルタ関数が使用されているノード（ただしイプシロン関数の引数は除く）
副作用ノード（S）	コンストラクタ/メソッド呼び出しを実行するノード またはフィールド，配列に代入を行っているノード
ミュータブルノード（M）	ミュータブルなオブジェクト（プリミティブ型や String 型など以外）を 使用しているノードまたは final でない static フィールドを使用しているノード メソッド呼び出しを行うノードもこれに入る． また，配列はミュータブルと見なされるが，配列長はミュータブルとは見なさない． デルタ関数の値がミュータブルな場合も，ミュータブルノードと見なされる．
例外ノード（E）	コンストラクタ/メソッド呼び出しを実行するノード，または Null 例外，0 除算例外，配列アクセス例外が 発生する可能性のあるノード

表 2　ノード種別間依存関係
Table 2　Kinds of dependence.

ノード間	依存関係
ε → δ	データ依存（DWR）
S → S	データ依存（DWW）
S → M	データ依存（DWR）
M → S	データ依存（DRW）
S → E	例外依存
E → S	例外依存
E → E	例外依存

は，副作用ノード，ミュータブルノード，例外ノードへ依存関係が追加される．副作用ノードどうしには DataWriteWrite（DWW）のデータ依存関係が追加される．副作用ノードからミュータブルノードへは DWR の依存関係が追加される．これは副作用ノードによりミュータブルなオブジェクトの値が変更される可能性があるためである．副作用ノードから例外ノードへは例外依存関係が追加される．また，ミュータブルノードから副作用ノードへも DataReadWrite（DRW）のデータ依存関係が追加される．これは，ミュータブルノードで参照しているオブジェクトの値が副作用ノードで変更される可能性があるためである．そして，例外ノードから副作用ノードと例外ノードへの例外依存が追加される．

制御依存関係は CFG における分岐をもとに，グラフに追加される．あるノード n から制

御フローが分岐しているとき，分岐後枝中のすべてのノードはそのノード n に制御依存する．ただし，分岐後の両方のパスに同一のノードがあり，それらが n からそれらのノードに至るまでのノードに依存しないとき，それらのノードはノード n に制御依存しない．また，現在の Dualcut では try-catch 構文を対象としていないため，例外ノードからの制御依存は考慮していない．

依存関係を計算する際，別名の存在，配列（コレクションも含む）の扱いおよび，手続き間の依存関係に注意する必要があるが，Dualcut では，実装の簡単化のためにこれらの解析を行っていない．しかし，より保守的に依存関係を判断しているためプログラムの意味を破壊するようなことはない．

4.3　ユーザヒント

PDG を構築する際に，保守的に解析を行っているため本来存在しない依存関係をグラフに追加している可能性があり，最適化対象の 2 つのメソッド呼び出しをアドバイスで置き換えることができない状況が発生しうる．このような状況を緩和する手段として，ユーザヒントという機構を提供している．ポイントカットで表明されたユーザヒントを用いると，一部の依存関係を排除することが可能になり，結果としてアドバイスで置き換えられる可能性があがる．

noex（例外発生がない）を表明すると，対象ノードには例外ノードのラベルが付けられない．nose（副作用がない）を表明すると，副作用ノードのラベルが付けられない．このため，これらのラベルに関連する依存関係を排除することができる．argsonly（メソッドの実行が引数にのみ依存する）を表明すると，メソッドの引数の型を調べ，全引数の型がイミュータブル型であれば，そのメソッド呼び出しのノードにはミュータブルノードのラベルが付かない．notnull（返り値がヌルではない）が表明されると，メソッド呼び出しからの返り値を使用している式でヌル例外は起きないと判断され，ヌル例外による例外依存関係を排除することができる．

これらのユーザヒントは，最適化対象メソッドの仕様から判断できると考えられるものを採用している．noex が表明可能かどうかは，Java の throws 節の有無などのメソッドの仕様から判断可能と考えられる．Java では例外は大きく分けて，Exception 系（検査例外）と RuntimeException 系（非検査例外）と Error 系の 3 つに分類される．Exception 系（検査例外）は throws 節で明示的に宣言する必要があるため，throws 節の有無から判断することができる．RuntimeException 系（非検査例外）は通常，入力チェックを行うことによって発生させない仕様にすることが多い．Error 系は，メモリ不足時のような対応が非常に困難な例外であり，多くのアプリケーションではこの例外は無視される．最適化対象メソッドでも無視してよいという判断が可能であれば，noex をポイントカットに表明することができる．nose が表明可能かどうかも，メソッドの仕様から判断可能と考えられる．副作用があるかないかは非常に重要な仕様であり，実際 C++言語では副作用がないことを宣言するための const 修飾子が存在する．argsonly も，メソッドの仕様から判断可能と考えられる．argsonly と似た概念としてステートレスという概念が存在する．ステートレスとは，実行が状態に拠らないことを意味し，Web アプリケーションなどでは重要な概念として知られており，EJB（Enterprise JavaBeans）では Stateless というオブジェクトが状態に依存しないことを宣言するためのアノテーションが提供されている．notnull も，メソッドの仕様から判断可能と考えられる．メソッドの返り値に null を許すかどうかは重要な仕様であり，API ドキュメントに明示的に記述されることもある．実際，ヌルがないことを宣言する機能として NonNull アノテーションという機能を Java に導入することが予定されている．

ただし，ユーザヒントはこの 4 つですべてではない．たとえば，メソッドの返り値が「0 ではない」などのヒントも考えられる．現状の Dualcut では設計の単純さのために，重要であると考えられる 4 つだけを採用している．また，ユーザヒントを間違って指定すると依存関係を破壊することになってしまうので，注意が必要である．

5. 実　　験

Dualcut を用いた最適化が可能であることの確認と，Dualcut によるオーバヘッドの計測のためにマイクロベンチマークを行った．2 章で示した従業員一覧を取得するプログラムに 3 章で示したアスペクトを適用して実験した．実験にはアプリケーションサーバ，データベースサーバとして 2 台の Ubuntu 9.04（Intel(R) Xeon(R) CPU 2.83 GHz * 4，Memory 8 GB）サーバ，JVM 1.6.0，MYSQL 5.0.75 を利用した．

表 3 に実験結果を示す．元の非最適化コードと Dualcut のアスペクトで最適化したコード，そして直接手でコードを修正して最適化したコードについて，実行時間を比較している．非最適化コードでは従業員一覧を取得するのに約 178 ミリ秒程度かかっているのに対し，Dualcut で最適化を実施したコードでは同一の結果を得るのに約 38 ミリ秒に抑えられており，最適化の効果が確認できる．一方，Dualcut による最適化と手で直接実施した最適化には差が見られない．Dualcut の実装にはアドバイス呼び出しなどのオーバヘッドが存在するため，これは JIT やネットワークなどの影響が存在するものと考えられる．

そこで，Dualcut によるオーバヘッドを調べるために，実際には DB にアクセスしない

表 3　最適化結果
Table 3　Result of the experiment.

	平均（ms）	標準偏差（ms）
非最適化コード	$1.78*10^2$	$0.79*10^2$
Dualcut による最適化	$0.38*10^2$	$0.15*10^2$
コードを直接修正した最適化	$0.38*10^2$	$0.13*10^2$

表 4　オーバヘッド計測結果
Table 4　Overhead.

	平均（ms）	標準偏差（ms）
Dualcut による最適化	1.46*10	0.15*10
コードを直接修正した最適化	0.98*10	0.23*10

DAO を利用し再度実験を行った．結果を**表 4** に示す．DB アクセスをともなわないため実行時間が短くなっており，Dualcut による最適化の方が遅くなっている．Dualcut によるオーバヘッドは 5 ミリ秒程度である．これはおおよそ Dualcut のアドバイスの織り込み 1 カ所あたりのオーバヘッドである．

6. 関 連 研 究

6.1 アスペクト指向

メソッド内の複数の文を扱うことができるアスペクト指向言語はいくつか存在する．transactionalcut [7] や regioncut [8]，LoopsAJ [9] は，複数の文を 1 つの塊としてポイントカットすることができる．transactionalcut や regioncut は始点と終点のポイントカットを指定し，その間のコード全体をポイントカットすることができる．LoopsAJ は，for などのループをポイントカットすることができる．しかし，これらの研究はコードのある領域全体をポイントカットするため，始点終点だけではなく間のコードもポイントカットに含まれる．本研究のように間に挟まれたコードの意味を変えずにアドバイスを実行することはできない．

Declarative event patterns [10] や tracematch [11] は，実行時のイベントやイベントの履歴をポイントカット可能なアスペクト指向言語である．tracematch では，実行中のイベントをつねに監視し，関心のあるイベントのパターンが起きたときにアドバイスを実行することができる．このようなシステムでは，離れた 2 つのメソッド呼び出しに対してアドバイスを適用することもできる．たとえば，H の実行後に T が実行されるというイベントを捕まえればよい．しかし，イベントを捕まえられるのは 2 つ目のメソッド呼び出しの実行時であるため，1 つ目のメソッド呼び出しを無効にすることは難しい．Dualcut が可能にするようなアドバイスは実装できない．

dflow [12] は，データフローを表現するポイントカットである．つまり，「あるポイントカットから得られたデータが別のポイントカットで使用される」というようなことを表現できる．dflow を用いれば，2 つの異なるポイントカットの関係を記述することが可能になる．しかし，dflow がアドバイスを適用できるのは 1 つのポイントカットに対してだけであり，Dualcut は離れた 2 つのメソッド呼び出しに対してアドバイスを適用することができる．

6.2 最　適　化

本研究は覗き穴最適化の 1 つである「特殊命令への置き換え」[13] の一種であると見なすこともできる．特殊命令への置き換えは機械語レベルでコード移動を実施し，その結果並んだ複数の命令を対象コンピュータが持つ特殊な 1 つの命令に置き換えるという最適化である．この複数の命令を 1 つの命令で置き換えるという点は本研究と同じである．ただし，本研究では置き換えを高級言語レベルで行っているため，命令単位でなくメソッド単位での置き換えをしている，さらにユーザが指定したメソッドで置き換えるという点が異なる．

また，本研究はジョインポイントの選択にコード移動などを取り入れた点が特徴である．本研究で使用しているコード移動アルゴリズムは，パーコレーション・スケジューリング [14] と同じものである．パーコレーション・スケジューリングは，並列性を上げるためのコード移動アルゴリズムで，特定のコードを前に移動させるアルゴリズムである．ただし，本研究では変換規則の見通し良さのために if 文を抜ける規則を定義している．パーコレーション・スケジューリングの変換規則は Delete，Move-op，Move-cj，Unification の 4 つの変換規則が定義されているが，if 文を抜ける規則は明示的に定義されていない．

コード移動アルゴリズムと依存関係の解析には，パーコレーション・スケジューリング以外にも様々なアルゴリズムが提案されており，それらのアルゴリズムを使用することで現在の Dualcut が対象としていない制御構文にも対応できる可能性がある．

ループ不変式の移動 [13] は，計算結果がループに依存しない式を探し，それをループ外に移動させることによって計算量を減らす最適化である．現状の Dualcut では，ループを超えた計算を対象としていないため，この手法を利用していないがループに対応する場合，ループ不変式を見つけることはループによる依存関係を除去する効果があると思われる．

Lazy Codemotion [15] は，部分冗長性除去に使用されるコード移動アルゴリズムであり，コード移動が可能な箇所の中からなるべく変数の生存期間が短くなる箇所を選びだしてコードを移動させるアルゴリズムである．このアルゴリズムは，Dualcut で利用することも可能

である．H が移動可能な箇所と T が移動可能な箇所を計算し，H と T が並ぶ箇所にコードを移動させるという戦略も可能である．Dualcut では実装の簡単さからパーコレーション・スケジューリングを採用している．

トレース・スケジューリング[16]，ハイパーブロック・スケジューリング[17] はパーコレーション・スケジューリング同様，並列性を上げるためのコード移動アルゴリズムである．各分岐について実行のトレースあるいは推測によってどちらに分岐するかの予測を立て，確率の高い方を優先してスケジューリングし連続したコードに変換するアルゴリズムである．このアルゴリズムも本研究に利用することはできると考えられる．分岐をなくした複数フローを構築し，各フローについて H と T を順に移動させることができる．

また，PDG を拡張した GPDG[18] を用いてパーコレーション・スケジューリングも提案されている．GPDG は本研究と同様 PDG 構築に SSA 形式を利用している．GPDG では，さらに制御依存関係をデータ依存関係として表現している．この制御依存関係をデータ依存関係として表現した PDG は文献 19) で提案されているものである．Dualcut でも GPDG を利用することは可能である．ただし，利用したとしても先頭末尾連続可能の条件には影響がないため，採用していない．

再帰やポインタ操作を含むプログラムに対する PDG の構築アルゴリズムも提案されている[20]．Dualcut では，簡単のために手続き間解析を行っていない．PDG 構築時に手続き間解析を利用することで，余分な依存関係を排除できる可能性がある．

7. ま と め

本研究では，離れた 2 つのメソッド呼び出しに対してアドバイスを適用できるアスペクト指向言語 Dualcut を提案した．Dualcut を用いれば，2 つのメソッド呼び出しの間に他のコードが存在しても，先頭末尾連続可能という条件を満たしていれば，それらに影響を与えずにアドバイスを適用することができる．その結果，従来のアスペクト指向言語ではうまく対応できなかった最適化などを実現できるようになる．

8. 今後の課題

現在の Dualcut は，2 つのメソッド呼び出し（間に別のコードがあってもよい）の並びという関係のみを扱を扱う．しかし，これは 2 つのメソッド呼び出しの関係の 1 つであり，順次実行以外にも様々な関係が考えられる．たとえば，最初のメソッド呼び出しの実行結果によって 2 つ目のメソッドが実行されるかどうかが決定される，つまり制御依存関係があるようなものもありうる．このような関係は，先頭末尾連続可能の条件を満たすことがないため，アドバイスで置換することはできない．順次実行以外の関係への対応は今後の課題である．また，2 つのメソッドだけではなく 3 つ以上のメソッドへの拡張も考えられる．

2 つのメソッド呼び出しでのデータの関係も現在の Dualcut では対応していない．たとえば，2 つのメソッド呼び出しで使用される引数が同じ変数であるときだけ，あるいは最初のメソッド呼び出しの結果を 2 つ目のメソッド呼び出しで使用しているときだけ，アドバイスの呼び出しで置換するというようなことは，現在の Dualcut ではできない．これは Dualcut の設計を単純にするために採用していない．ただし，アドバイス側の工夫で対応することができる．ポイントカットからコンテキスト情報を取得し，メソッド呼び出しの引数の値の同一性などを確認することにより，本当にアドバイスを実行すべきときか否かを判断できる．しかし，Dualcut はそのような判断を直接支援する機構を提供していない．

さらに，現在の PDG 構築アルゴリズムは単純であり，多くの不要な依存関係を排除できていない可能性がある．別名解析，配列の解析，手続き間解析など，より精緻な依存性の解析も今後の課題である．

参 考 文 献

1) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An overview of AspectJ, *ECOOP '01*, Vol.2072 of LNCS, pp.327–355 (2001).
2) JBoss Inc.: Hibernate. http://www.hibernate.org
3) Apache Software Foundation: ibatis. http://ibatis.apache.org
4) Ekman, T. and Hedin, G.: The jastadd extensible java compiler, *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, *OOPSLA '07*, New York, USA, pp.773–774, ACM (2007).
5) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The program dependence graph and its use in optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, pp.319–349 (July 1987).
6) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, pp.451–490 (Oct. 1991).
7) Sadat-Mohtasham, H. and Hoover, H.J.: Transactional pointcuts: Designation reification and advice of interrelated join points, *Proc. 8th international conference on Generative programming and component engineering*, *GPCE '09*, New York, USA, pp.35–44, ACM (2009).
8) Akai, S. and Chiba, S.: Extending aspectj for separating regions, *Proc. 8th inter-*

national conference on Generative programming and component engineering, *GPCE '09*, New York, USA, pp.45–54, ACM (2009).
9) Harbulot, B. and Gurd, J.R.: A join point for loops in aspectj, *Proc. 5th international conference on Aspect-oriented software development*, *AOSD '06*, New York, USA, pp.63–74, ACM (2006).
10) Walker, R.J. and Viggers, K.: Implementing protocols via declarative event patterns, *Proc. 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, *SIGSOFT '04/FSE-12*, New York, USA, pp.159–169, ACM (2004).
11) Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: Adding trace matching with free variables to aspectj, *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, *OOPSLA '05*, New York, USA, pp.345–364, ACM (2005).
12) Masuhara, H. and Kawauchi, K.: Dataflow pointcut in aspect-oriented programming, *1st Asian Symposium on Programming Languages and Systems*, *APLAS '03*, New York, USA, pp.105–121 (2003).
13) SETHIR. AHO, A.V. and ULLMANJ. D.: *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1985).
14) Nicolau, A.: Percolation scheduling: A parallel compilation technique, Technical report, Ithaca, USA (1985).
15) Knoop, J.R.O. and Steffen, B.: Optimal code motion: Theory and practice, *TOPLAS*, Vol.16.
16) Howland, M.A., Mueller, R.A. and Sweany, P.H.: Trace scheduling optimization in a retargetable microcode compiler, *Proc. 20th annual workshop on Microprogramming*, *MICRO 20*, New York, USA, pp.106–114, ACM (1987).
17) Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Quellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Hwu, W.-M.W. and Lavery, D.M.: The superblock: An effective technique for vliw and superscalar compilation, *The Journal of Supercomputing*, Vol.7.
18) 小松秀昭，古関　聰，深澤良彰：命令レベル並列アーキテクチャのための大域的コードスケジューリング技法，情報処理学会論文誌，Vol.37, No.6, pp.1149–1161 (1996).
19) Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of control dependence to data dependence, *Proc. 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, *POPL '83*, New York, USA, pp.177–189, ACM (1983).
20) 佐藤慎一，植田良一，井上克郎：再帰やポインタを含むプログラムの効率的な依存関係解析法の提案，電子情報通信学会技術研究報告 SS，ソフトウェアサイエンス，Vol.95, No.475, pp.9–16 (1996).

(平成 22 年 9 月 29 日受付)
(平成 22 年 12 月 27 日採録)

伊尾木将之

2003 年大阪大学大学院理学研究科修了．現在，日本 IBM に勤務および，東京工業大学大学院情報理工学研究科に在学中．

千葉　　滋（正会員）

東京工業大学大学院情報理工学研究科准教授．1991 年東京大学理学部情報科学科卒業．1993 年同大学院理学系研究科情報科学専攻修士課程修了．1996 年同専攻より理学博士．東京大学助手，筑波大学講師，東京工業大学講師を経て現職．プログラミング言語およびオペレーティング・システム等，システムソフトウェアの開発に興味を持っている．