豊富な情報を基にしたpointcutを記述できるアスペクト指向言語

中川清志[†] 千葉滋[†]

本稿では，豊富な情報に基づいたpointcutが可能なアスペクト指向プログラミング（AOP）言語，Joshについて述べる．AOPでは，weaverと呼ばれる処理系がプログラムを合成するが，合成個所を指示するものがpointcutである．しかしながら既存のAOP言語では，pointcutで参照できる情報が限られているため，ユーザが望む合成を記述できない場合がある．たとえばAspectJでは，pointcutを記述する際にメソッドのシグネチャなどの情報しか参照できず，メソッドの中身に関する情報は参照できない．本稿ではまずこの問題点を指摘し，次にJoshでは，豊富な情報に基づいたpointcutを定義できることを示す．Joshではpointcutのための指定子を，ユーザがJavaで新たに定義できるように設計されており，そしてその際に，リフレクションを用いてより豊富な情報を参照できる．そのため，AspectJのようにあらかじめ決められたpointcut指定子の組合せしか使えない言語よりも，Joshのpointcut記述力は高い．また本稿ではweaveにかかる時間，および実行時のオーバヘッドをベンチマークテストで計測する．その結果をAspectJと比較し，性能劣化は小さいことを示す．

An Aspect-Oriented Programming Language for Pointcuts Using Various Program Information

Kiyoshi Nakagawa[†] and Shigeru Chiba[†]

This presentation proposes Josh, which is our new Aspect-Oriented Programming (AOP) language that enables to describe pointcuts using various program information. In AOP, pointcuts specify the join points where programs are combined by a weaver. In general, the description of pointcuts needs various information of the programs. However, since the information available in pointcuts is limited in existing AOP languages, there is a case that the users cannot describe aspects that they need. This presentation first points out this problem and then shows our Josh language, in which it is possible to define a new pointcut that specify join points by using various information. Josh allows the users to define a new pointcut designator in Java. Thus the implementation of the pointcut designator can use reflective computing provided by Josh for accessing various information. Therefore Josh has more powerful expressiveness than other AOP languages like AspectJ, which provides only the limited kinds of pointcut designators. Also this presentation shows the weaving time and run time overhead that we measured by benchmark tests, and they were compared with AspectJ's. The result showed that overheads due to Josh is relatively small.

1. はじめに

オブジェクト指向などの従来のプログラミング技法では，ロギング，同期，永続性などの処理群を上手にモジュール化するのが難しいといわれている．これらの処理群は複数モジュール間に散らばってしまい，横断的関心事（crosscutting conern）といわれる．アスペクト指向プログラミング（AOP）[9]とは横断的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る．基本モジュールとアスペクトを合成して両方の機能を持つプログラムを作り出す処理を，weaveと呼び，その処理系をweaverと呼ぶ．その際に合成個所を特定する記述のことを *pointcut* と呼び，プログラム内の様々な個所を特定するのに使う．

数多くのAOP言語が提案されており，その中の代表的なものがAspectJ[8]である．AspectJは広く使われており完成度も高いが，課題はまだ残っている．まずpointcutの記述に使う，pointcut言語が不十分であるといえる．このためユーザの意図した個所に，アスペクトをweaveできない場合がある．また汎用的なインタータイプ宣言（旧イントロダクション）が記述できず，同じようなプログラムを繰り返して書かなければならない．これを解決するためのパラメータつきインタータイプ宣言[6]という，C++のテンプレートのような機能が提案されているが，AspectJにはそのよ

† 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

うな機能はない．

これらの問題に対処するために，本稿では AOP 言語 *Josh* を提案する．Josh は pointcut 言語を拡張して記述力を高められるのが特徴である．pointcut は pointcut 指定子と呼ばれる記述子の組合せで構成されるが，指定子はそれぞれ固有のアルゴリズムを実装している．そして各指定子は，そのアルゴリズムに基づいた振舞いをする．AspectJ ではこの pointcut 指定子の数が限られているため，pointcut の記述力が低いといえる．Josh では，ユーザが独自のアルゴリズムを実装した pointcut 指定子を新たに定義できる．定義に使う言語は Java であり，この際に，リフレクションを使ってプログラム内の様々な情報を参照し，複雑なアルゴリズムを実装した pointcut 指定子を定義できる．定義には手がかかり技術がいるが，他のプログラマが再利用することは容易である．さらに Josh にはインタータイプ宣言にも汎用性のための機能が備わっており，インタータイプ宣言の対象を pointcut 言語で指定できる．また対象に応じて，異なったメソッドやフィールドが追加されるようなインタータイプ宣言を記述可能である．

以後，2 章で AspectJ の問題点をあげる．3 章で Josh を提案し，続いて 4 章では Josh による解決方法を示す．5 章では性能を評価し，6 章で関連研究について議論する．最後に 7 章で本稿をまとめる．

2. 現在の AspectJ の問題点

この章ではまず AspectJ について簡単に説明し，AOP に必要な概念を説明する．そして次に，現在の AspectJ（バージョン 1.1）の問題点を示す．

2.1 AspectJ のプログラミングモデル

AspectJ において，アスペクトとは横断的関心事をモジュール化する単位であり，アスペクトフィールド，アスペクトメソッド，インタータイプ宣言，アドバイスで構成される．アスペクトフィールド，アスペクトメソッドはアスペクトに属するフィールドやメソッドのことである．インタータイプ宣言は，古くはイントロダクションと呼ばれており，新たなフィールドやメソッドを別のクラスに追加する．

アドバイスは，プログラム内のある演算のある実行点で，新たなコード断片を実行することを指示する機能である．これは *pointcut* とアドバイスボディの対で定義される．アドバイスボディは，新たに実行されるコード断片であり，その実行点を特定する記述が pointcut である．pointcut で特定された実行点がアドバイスの合成個所として特定される．アドバイスには before，after，around の 3 種類があり，コード断片を実行するタイミングが異なる．それぞれ pointcut で特定された実行点の前，後，もしくはその代わりに実行する．アドバイスのプログラム例は，

```
before() : call(int Point.getX()) {
  System.out.println("before call getX");
}
```

のようになる．これは Point クラスの getX メソッド呼び出しの前に，メッセージを出力することを指示する．

AspectJ の仕様では明確に述べられていないが，インタータイプ宣言もまた，pointcut とインタータイプボディの対であるといえる[10]．たとえば，

```
int Point.getX() { return x; }
```

というインタータイプ宣言は，Point クラスのみに getX メソッドが追加される．もし Point の代わりに Point+ となっていれば，Point の全サブクラスに getX メソッドを追加する．つまり Point や Point+は，メソッドの追加先を特定しているので，インタータイプ宣言中の pointcut であると見なせる．インタータイプボディは残りの部分である．

その他のアスペクトの機能に，コンテクスト引き渡しがある．アドバイスでは pointcut で特定された演算の静的・動的な情報を，ボディの中で参照することができる．たとえば，メソッド呼び出しを pointcut で特定したときに，その引数をボディの中で参照するといったことであり，AspectJ の args や target にはこの機能がある．本稿ではこれを**コンテクスト引き渡し**と呼ぶ．thisJoinPoint にもコンテクスト引き渡しの機能があり，args などのように pointcut 部分に明記することなく使用できる．AspectJ のインタータイプ宣言には，コンテクスト引き渡し機能はない．

2.2 AspectJ の記述力の限界

現在の AspectJ では記述できない pointcut が存在する．また，インタータイプ宣言に関してもコンテクスト引き渡しができないため，汎用性が低くなっている．

2.2.1 pointcut 記述の問題点

ここでは図形エディタの例[7]を用い，pointcut 記述に関する問題点について述べる．図形エディタの実装には，直線を表す Line クラス，長方形を表す Rectangle クラスなどが含まれ，それらは FigureElement のサブクラスである（**図 1**）．これらの図形クラスのオブジェクトの状態は，Screen オブジェクトが画面に描画する．このような実装では，図形オブジェクトの外観が変化したとき，再描画のために，必ず Screen オブジェクトの repaint メソッドが呼ばれなければならない．すると repaint が，各図形オブジェクトの redraw

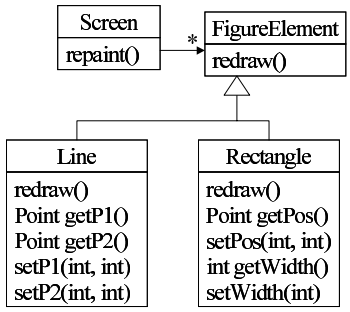


図 1 図形エディタ
Fig. 1 Figure editor.

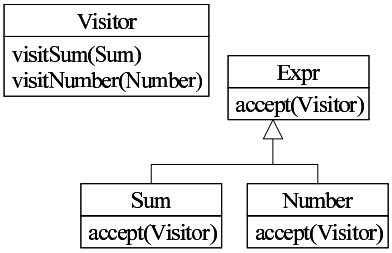


図 2 算術式の木構造
Fig. 2 The tree structure representing expressions.

メソッドを呼び，図形の外観の変化を画面に反映することになる．

しかし，このような再描画処理を実現するには，各図形クラスについて，setWidth のような図形の外観を変更するすべてのメソッドの中に，忘れずに repaint メソッドの呼び出しを書かなければならない．このようなメソッドの呼び出しは，各図形クラスのメソッド内に散らばってしまうことになり，横断的関心事となってしまう．

このような関心事は，以下のようにアスペクトとしてモジュール化できるとよい．まず Rectangle クラスの setWidth メソッドのように，図形の外観変化を起こすメソッド呼び出し実行点のすべてを抽出するように，pointcut を定義する．次に pointcut で抽出されたメソッド呼び出しが起こる個所で，Screen の repaint を呼ぶように，アドバイスボディを定義する．このアスペクトの記述の要点は，図形クラスの外観変化を起こすすべてのメソッドを抽出するように pointcut を定義することである．

このような pointcut の定義のためには,「図形の外観変化を起こすメソッド呼び出し」などといったルールに基づいた pointcut 記述ができると有効である．このルールを詳細に見ると，以下のようになる．各図形クラスの redraw メソッドは，それぞれのクラスのフィールドを読み込みそれを反映するように描画する．つまり図形の外観変化を起こすメソッドとは，それらのフィールドに書き込みを行うメソッドである．このルールに基づいて pointcut を記述できればよい．

しかしながら AspectJ でそのような pointcut を記述することは煩雑であり，また直感的な記述は難しい．そのためには，プログラマは外観変化を起こすメソッドをすべて数え上げるか，もしくはそのようなメソッドはすべて"set"で始まるというような決まりを作り従わなければならない．確かに AspectJ には,「"set"で始まるメソッドすべて」,「返り値が"int"型のメソッドすべて」などといった，シグネチャに関する条件を提示した pointcut の記述も可能である．しかしながら，そのような決まりは煩わしく，またたとえプログラマがそのような努力をしたとしても，抽出の必要があるすべてのメソッドが過不足なく特定されているかを保証できない．

これは AspectJ の pointcut 言語を構成する pointcut 指定子が限られているからである．pointcut 指定子は，それぞれ固有のアルゴリズムに基づいて pointcut 先を具体的に特定するものだが，指定子の種類は AspectJ 言語が仕様で与えているものだけであり，限られている．そのためプログラマが独自のアルゴリズムに基づいて，pointcut 先を特定したいとしても，そのアルゴリズムに一致する指定子がない場合には，pointcut を記述できない．AspectJ には，先ほど示したようなルールを表現する指定子はないが，将来的にそのような pointcut 指定子が言語仕様として追加されるかもしれない．しかし，単純に指定子を増やしていくのには問題がある．そのような新たな pointcut 指定子の多くは，一般に特別な場合にしか使われないが，多数の指定子は言語自体を複雑にしてしまうからである．

2.2.2 インタータイプ宣言の汎用性

次に，木のノードを巡回する処理を実装することを考える．もし Java でこの関心事を実装するならば，Visitor パターン[4]に沿って，**図 2** のように Visitor クラスと，木の全ノードに accept メソッドを用意するだろう．accept メソッドは，引数で与えられた Visitor オブジェクトの visitXX メソッド（XX は木のノードクラス名）を呼ぶ．

もし AspectJ で実装するならば，Visitor クラスと accept メソッド群は，元のクラス定義から分離され，アスペクトとして独立にモジュール化されるだろう．しかしながらプログラマは各ノードクラスに accept メソッドを追加するインタータイプ宣言を，繰り返して書かなければならない．もしクラス数が 10 ならば，

10 通りの accept メソッドを定義しなければならず，またそれらのメソッド定義にはほんの少しの差しかない．たとえば以下のようなプログラムになる．

```
void Sum.accept(Visitor v) {
    v.visitSum(this);
}
void Number.accept(Visitor v) {
    v.visitNumber(this);
}
```

この例の 2 つのインタータイプ宣言の違いは，accept メソッドを宣言しているクラスと，引数 v が呼ぶメソッドだけである．

このような冗長性をなくすには，汎用的な accept メソッドを定義し，繰返しを避ければよい．

```
void Expr+.accept(Visitor v) {
  Class nodeClass = this.getClass();
  String name = "visit"
              + nodeClass.getName();
  Method m =
    v.getClass().getMethod
    (name, new Class[]{nodeClass});
  m.invoke(v, new Object[]{this});
}
```

このインタータイプ宣言は Expr クラスのすべてのサブクラスに，accept メソッドを追加する．しかし，この accept メソッドは Java のリフレクション[11]を使っているため，性能が著しく低い．さらに，このメソッドは複雑で理解が難しい．

3. Josh

前章での問題に対処するために，この章では豊富な情報に基づいた pointcut が可能なアスペクト指向言語 Josh を提案する．現在の Josh は，AspectJ のすべての機能には対応してないが，ユーザによる新たな pointcut 指定子の定義ができ，またインタータイプ宣言でも pointcut，およびコンテクスト引き渡しができるなどと，独自の機能を提供している．

3.1 Josh の設計

この節では具体的に Josh のプログラムコードについて説明する．Josh の文法は AspectJ をもとにしており，大部分は類似している．以下のプログラムは Josh で書かれたロギングアスペクトの例である．

```
aspect Logging {
  before : call("void Point.set*(..)") {
    System.out.println("Point.set* was called");
  }
}
```

これは Point クラスの set で始まるメソッドが呼ばれる直前に，ロギングメッセージを出力する．このプログラムと AspectJ との差は，call の引数がダブルクォートで囲まれているところである．これは現在の実装上の理由によりこのような仕様になっているだけであり，本質的な違いはない．

アドバイス記述は類似しているが，Josh のインタータイプ宣言は AspectJ のそれと異なっている．Josh では pointcut とボディを明確に分けて記述する．たとえば，以下のようになる．

```
intertype : same("Point") {
  int getX() { return x; }
}
```

これは getX メソッドを Point クラスに追加する．same 指定子は，あとに続くブロック内で宣言されたメソッドを，どのクラスに追加するかを特定している．intertype 句は，あとに続くブロックがインタータイプ宣言であることを示している．このように AspectJ と異なる文法を用いるのは，インタータイプ宣言でもアドバイスと同じような強力な pointcut を利用できるようにするためである．詳細は 3.3 節で述べる．

またインタータイプ宣言には次のような機能もある．

```
intertype : same("Point") :
  implements("java.lang.Comparable");
```

これは Point クラスが java.lang.Comparable インタフェースを継承するように，クラス定義を変える．先ほどと違い，same と implements の間が :（コロン）になっている．2 つ目のコロンに続く指定子は場所を特定するものでなく，ボディの一部である．

Josh には AspectJ の args のように実行時のコンテクスト情報引き渡しのための指定子はない．その代わりに，特別な宣言をせずにアドバイスボディの中では使える変数が用意されている．この機能は Josh の基盤となっているシステム，*Javassist* により提供されている．

たとえば，メソッド呼び出しやフィールド参照のときには，$0 がターゲットオブジェクトを表し，$1, $2, . . . はメソッド呼び出しの第 1 引数, 第 2 引数, . . . を表す．これらは AspectJ の args と類似の機能を実現する．これ以外にも特別な変数が使えるが，詳細は文献 3) に譲る．

$1，$2 などの使用例を以下に示す．

```
before : call("void *.move(int, int)") {
  if ($1 < 0 || $2 < 0)
    System.err.println("wrong arguments.");
}
```

この before アドバイスは，move メソッドの引数が負の場合にエラーメッセージを出力する．この場合の $1 と $2 の型は int であるが，これらの型は対象となっている joinpoint の引数に合うように自動的に処理される．

さらに around アドバイスのボディ内では，$proceed

を使用できる．これは AspectJ の proceed と同じである．around アドバイスはある特定の演算をとらえ，その演算の実行の代わりにアドバイスボディを実行する．もし，本来実行するはずだった演算を実行したい場合には，$proceed を呼べばよい．たとえば以下のように使う．

```
around : call("void *.move(int, int)") {
  if ($1 < 0 || $2 < 0)
    throws new Exception();
  else
    $_ = $proceed($1, $2);
}
```

このアドバイスは引数が負でないときは，本来の演算である move メソッド呼び出しを，本来の引数で実行する．$_は演算の返り値となっており，$_に代入した値が，pointcut で特定された演算の結果として返る．また$proceed の引数には，$$という特別な変数も渡せる．これは全パラメータをリストにしたもので，$1，$2, . . . を並べたものと同じである．そのため $proceed($1, $2) は$proceed($$) と同じ命令になる．$$は，自動的に適切な引数を代入してくれるので，パラメータ数を考える必要がない．そのため，たとえば異なる引数を持つメソッド呼び出しを pointcut で特定したときに，同じアドバイスを適用することができる．これは再利用性の高いアドバイスを記述するのに役立つ．

3.2 Josh コンパイラ

Josh のコンパイラは，Josh 言語のソース（.josh ファイル）から，Java で記述された専用 weaver のソース（Weaver.java ファイル）を生成する（**図 3**）．専用 weaver は.josh で指定されたアスペクト定義が埋め込まれており，この定義に沿って，別の Java ファイル（.class）のクラス定義を変換する．生成された専用 weaver を使って，weave を実行するのが Weaver-Driver である．WeaverDriver の出力は Java のバイトコード（.class ファイル）であり，それらにはアスペクトが合成されている．

3.2.1 Joinpoint モデル

Josh の joinpoint モデルは AspectJ のものとほぼ同等である．以下ではこのモデルについて述べながら，Josh の説明に必要な用語を示す．*Joinpoint* とは，AOP 言語がコードを合成しうる個所のことであり，演算 joinpoint と構造 joinpoint がある．演算 joinpoint とはプログラムの演算の中で合成可能な個所のことであり，メソッド呼び出し，フィールド参照などが含まれる．これらはメソッドボディ中，またはコンストラクタボディ中に現れる．構造 joinpoint とは，プログラムの構造に関するものであり，クラス，メソッド，フィールドなどがある☆．

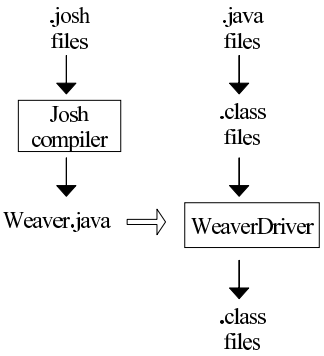


図 3 Josh コンパイラの流れ
Fig. 3 The flow of compilation by Josh.

pointcut とは，コードの合成をしたい個所に対応する joinpoint を抽出するものであり，対象とする joinpoint の種類と，そこに課する条件の対で表せる．pointcut は pointcut 指定子で記述される．たとえば AspectJ の指定子に call というものがある．これは call(int Point.getX()) のように使われるが，対象とする joinpoint はメソッド呼び出しであり，そこに課する条件が括弧内に記述されている．また AspectJ の execution 指定子は，指定したメソッドの始まりや終わりにコードを挿入するものであり，対象 joinpoint はメソッド（構造 joinpoint）である．

また pointcut 指定子には静的指定子と動的指定子の 2 種類がある．これらは，その pointcut が対象としている joinpoint に対して，どのような条件を課すかが異なっている．静的指定子は，プログラムの字句構造を解析して得られる情報が条件となる．たとえばメソッドのシグネチャやクラス名などである．AspectJ の call や within は静的指定子である．一方動的指定子は，実行時の情報が条件となる．たとえばメソッド呼び出しをしたときの，ターゲットオブジェクトの実行時の型などは実行時の情報である．AspectJ の target や cflow は動的指定子である☆☆．インタータイプ宣言に，動的指定子を使うことはできない．なぜなら Java ではクラス構造を実行時に変えることはできないため，実行時情報を条件とするインタータイプ宣言を実装できないからである．

3.2.2 Joinpoint オブジェクト

Java のプログラム内にある各 joinpoint を，Josh

☆ AspectJ の joinpoint はプログラム内の演算のみをさすが，Josh ではプログラムの字句構造を表す場合にも用いる．
☆☆ target は実行時の型に依存するため静的指定子ではない．

では *joinpoint* **オブジェクト**として扱う．これは joinpoint を抽象化したものであり，joinpoint の種類の数だけ，joinpoint オブジェクトの種類もある．Josh ではクラス，フィールド，メソッド，コンストラクタの 4 種類の構造 joinpoint オブジェクトがあり，それぞれ CtClass，CtField，CtMethod，CtConstructor というクラスのオブジェクトで表される☆．構造 joinpoint オブジェクトには，標準の Java リフレクション API に類似した，joinpoint の静的な構造を調べるためのメソッドが提供されている（**表 1**）．

さらにこれらの構造 joinpoint オブジェクトには，joinpoint の構造を改変するためのメソッドがある．たとえば CtClass には addField，addMethod があり，クラスに新たなフィールド，メソッドを追加できる．また setSuperclass，addInterface により，クラスの階層構造を変えることができる．これはインタータイプ宣言の機能に相当する．

演算 joinpoint にも同様に，各演算に対応する joinpoint オブジェクトがある．Josh にはメソッド呼び出し，フィールド参照，インスタンス生成，キャスト式，例外ハンドラ，instanceof 式の 6 種類の演算 joinpoint オブジェクトがあり，それぞれ MethodCall，FieldAccess，NewExpr，Cast，Handler，Instanceof というクラスのオブジェクトで表される．これらのクラスにも，joinpoint の情報を得るメソッドがある（**表 2**）．また replace メソッドを使えば joinpoint の挙動を変えることができ，アドバイスを実現できる．これらの構造・演算 joinpoint オブジェクトを表すクラスは，Javassist により提供されている．これらを利用すれば，joinpoint と直接は関係していないクラスやメソッドの情報を得ることも可能である．

3.2.3 コンパイラによるアスペクト変換

Josh の weaver は各クラスの定義を調べ，joinpoint を探す．Joinpoint を見つけると，それを表す joinpoint オブジェクトを生成し，与えられたすべての pointcut 定義と比較して合成が必要かを決定する．必要ならば，指定されたコードの合成を行う．

Josh コンパイラはこのような処理をする weaver を作成するために，アスペクト内の pointcut を weaver 中の if 文の条件式に変換する．この条件式は，pointcut の条件とその pointcut が対象とする joinpoint オブジェクトを含んでおり，joinpoint オブジェクトの値次第で真偽が変わる．weaver は次々に joinpoint オブジェクトを生成するが，生成したものと，この条件式の joinpoint オブジェクトの種類が一致する場合にのみ，この式は評価される．アドバイスやインタータイプ宣言のボディは，この式が真の場合に実行される．たとえば，

```
intertype : same("Point") {
  int z;
}
```

というコードは，Point クラスにフィールド z を追加するインタータイプ宣言であり，以下のように変換される．

```
if (c.getName().equals("Point"))
  c.addField(CtField.make("int z;", c));
```

ここで c は CtClass クラスの joinpoint オブジェクト

☆ *Ct* とは Compile-Time を意味し，標準の Java リフレクション API にある Class や Field などと区別するために付けられている．

表 1 CtClass クラスのメソッド（抜粋）
Table 1 Methods in CtClass.

内観用	
String	getName() クラス名を得る
CtClass	getSuperclass() 親クラスを得る
CtClass[]	getInterfaces() インタフェースを得る
CtField[]	getFields() フィールドを得る
CtMethod[]	getMethods() メソッドを得る
改変用	
void	setName(String name) クラス名を変える
void	setSuperclass(CtClass c) 親クラスを変える
void	setInterfaces(CtClass[] i) インタフェースを変える
void	addField(CtField f) フィールドを追加する
void	addMethod(CtMethod m) メソッドを追加する

表 2 MethodCall クラスのメソッド（抜粋）
Table 2 Methods in MethodCall.

内観用	
CtClass	getCtClass() ターゲットオブジェクトのクラスを得る
CtMethod	getMethod() 呼ばれたメソッドを得る
CtMethod	getMethodName() 呼ばれたメソッド名を得る
CtClass[]	mayThrow() このメソッドが投げる例外を得る
CtBehavior	where() このメソッド呼び出しが起こった場所を得る
改変用	
void	replace(String code) このメソッド呼び出し式を置き換える

であり，weaver が調査中のクラスを表している．if 文の条件式は，c が表すクラスの名前が Point であるかを評価する．インタータイプ宣言のボディに相当する部分は，if 文の波括弧内に挿入されており，joinpoint が条件にあうときだけ実行される．

アドバイスも同様に，weaver 中のコードに変換される．たとえば以下のような Josh コードのアドバイスがあるとする．

```
around : call("void *.move(..)") {
  System.out.println("move");
  $_ = $proceed($$);
}
```

このとき Josh コンパイラは以下のような if 文を生成する．

```
if (mc.getMethodName().equals("move")) {
  mc.replace
    ("{ System.out.println(\"move\");" +
     " $_ = $proceed($$); }");
}
```

ここでは mc は MethodCall オブジェクトである．call はメソッド呼び出し演算を対象としているため，この if 文は今注目している MethodCall joinpoint オブジェクトが条件に合うときにのみ，replace メソッドを実行し，joinpoint の挙動を変える．アドバイスボディは，replace メソッドの引数となっており，バイトコードに変換された後に挿入されて，本来の joinpoint の演算に対応するバイトコードと置き換わる．もしボディの中で$で始まる特別変数を使っている場合は，バイトコードに変換される際に適切な値に展開される．

アドバイスの pointcut が，実行時の情報を条件とした動的指定子の場合もある．その場合，挿入されるコードにも条件評価の式が含まれる．

```
around : target("Point") && within("Display") {
  System.out.println("point");
  $_ = $proceed($$);
}
```

上記の例は以下のような Java コードに変換される．

```
if (c.getName().equals("Display")) {
  mc.replace(
    " if ($0 instanceof Point) {" +
    " System.out.println(\"point\");" +
    " $_ = $proceed($$); }"
}
```

ここで c は CtClass オブジェクトである．挿入されるコード内でターゲットオブジェクトが Point のインスタンスであるかを実行時に調べている．

アドバイスが構造 joinpoint を対象とする場合には，その構造 joinpoint に含まれる演算 joinpoint のすべてがアドバイスの対象となる．たとえば withincode 指定子の対象 joinpoint がメソッドの場合，

```
before : withincode("void Point.init()") {
  /* advice body */
}
```

というアドバイスは，Point クラスの init メソッド内で見つかるすべての演算 joinpoint の直前に，そのボディを挿入する．

そのほかに，Josh コンパイラはアスペクトと同名のクラスを作りだす．このクラスのメンバは，アスペクト内で宣言されたフィールドやメソッドと同じである．現在の Josh では per-object アスペクトのような，アスペクトのインスタンスは使えずに，シングルトンである．

3.3 拡 張 性

Josh ではユーザが，インタータイプ宣言用，もしくはアドバイス用に新たな pointcut 指定子を定義することができる．新たな指定子は通常の Java の static メソッドとして定義する．Joinpoint オブジェクトはプログラム構造を詳細に検査するためのメソッドを提供するので，joinpoint を識別するための複雑なアルゴリズムを実装した pointcut 指定子を定義できる．

Josh コンパイラは pointcut 指定子を，boolean 値を返すメソッド呼び出し命令へ変換する．このメソッド呼び出し命令は，Josh コンパイラがアドバイスを変換するときに生成する if 文の条件式の中に挿入される．

ここで例として paramType1 という簡単な指定子の定義方法を示す．これは，第 1 引数が指定された型と一致するメソッドを抽出する．使用例は以下である．

```
paramType1("ColorPoint")
```

この pointcut 指定子は，第 1 引数が ColorPoint クラスであるメソッドを抽出する．この指定子の定義は，以下のような Java の static メソッドになる（簡易化のためエラー処理は省略してある）．

```
static boolean paramType1(CtMethod m,
        String[] args, JoshContext jc) {
  CtClass parType =
    m.getMethod().getParameterTypes()[0];
  CtClass argType = jc.getType(args[0]);
  if (parType.subtypeOf(argType))
    return true;

  if (argType.subtypeOf(parType)) {
    jc.setIf("$1 instanceof " +
             argType.getName());
    return true;
  }
  else
    return false;
}
```

paramType1 の第 1 引数は，この指定子が weave 対象とする joinpoint オブジェクトである．CtClass や

MethodCall などの他の joinpoint オブジェクトでもよい．第 2 引数は String 型の配列であり，この指定子に渡される引数である．ユーザ定義の指定子は，カンマで区切られた文字列を引数として受け取れる．第 3 引数は，様々な情報を保持している JoshContext オブジェクトである．この paramType1 メソッドは第 1 引数 m が，String の配列で与えられた引数と一致しているならば真を返す．

paramType1 は指定子は動的指定子であり，実行時の第 1 引数の型のチェックを必要としている．つまり静的にアドバイスボディを weave するかどうかを決定できない．そのため paramType1 メソッドは，JoshContext の setIf メソッドを呼び，実行時の型チェックのための条件評価の式がアドバイスボディと一緒に挿入されるようにしている．setIf の引数が，条件評価の式を表す文字列である．アドバイスボディは，この式を条件式とする if 文に包まれて挿入される．アドバイスボディは，実行時にこの条件評価の式が真であるときだけ，実行される．

インタータイプ宣言

Josh ではインタータイプ宣言用の指定子も新たに定義することができる．Josh のインタータイプ宣言は以下のような文法になっており，AspectJ のそれと異なっていることはすでに説明した．

```
intertype : within("Point"):
  implements("java.lang.Comparable");
```

この宣言のボディは implements であり，これにより Point クラスは java.lang.Comparable インタフェースを継承する．

Josh ではこの implements に代わる指定子も，Java の static メソッドで新たに定義できる．他のユーザ定義の指定子のように，メソッドの引数は joinpoint オブジェクト（構造 joinpoint のみ），String 型の配列，JoshContext である．メソッドの返り値は void である．もしこの joinpoint オブジェクトが，pointcut 指定子のそれと一致しない場合は，コンパイルエラーになる．以下に，このメソッドの実装例を示す．

```
static void addThrows(CtMethod m,
    String[] args, JoshContext jc) {
  CtClass type = jc.getType(args[0]);
  m.addExceptionType(type);
}
```

これは pointcut で抽出されたメソッドの throws リストに，新たな例外を追加する．

インタータイプ宣言のコンテクスト引き渡し

汎用的で再利用性の高いアスペクト記述のために，Josh のインタータイプ宣言はコンテクスト引き渡しの機能を持つ．この機能は，2.2 節で述べられた Visitor パターンの関心事をモジュール化するのに役立つ．

```
intertype : within("Expr+") {
  void accept(Vistor v) {
    v.visit<% josh.getCtClass().getName() %>(this);
  }
}
```

このインタータイプ宣言は，Expr の全サブクラスに accept メソッドを追加する．<%と%>の間には，任意の Java の式を記述できる．この式は String 型に評価されなければならない．評価後の文字列は，コンパイル時に<%と%>の間の部分と置換される．

この式内では，josh という特別変数を使用できる．これは抽出された joinpoint のコンテクスト情報を含む JoshContext オブジェクトへの参照である．上の例では，pointcut により抽出されたクラス名を josh を使って手に入れ，それに応じて挿入する accept メソッドを変えている．Josh コンパイラは，インタータイプ宣言のボディを以下のような String の連結に変換し，その計算結果を weaver が実行時に挿入する．

```
"void accept(Visitor v) { v.visit"
+ josh.getCtClass().getName()
+ "(this);}"
```

上の式は Expr の各サブクラスごとに評価され直すので，各サブクラスには異なるメソッドが挿入される．このインタータイプ宣言は 2.2 節の AspectJ のものに比べて簡潔かつ効率的である．Josh はコンパイル時のリフレクション[2)]を活用してるので，AspectJ を使った 2.2 節の解決法のような実行時ペナルティがない．

3.4 Josh の拡張性の限界

Josh はコンパイル時リフレクションのためのライブラリ，Javasssist[2)]を使って実装されている．そのため Josh の拡張能力やその容易さは，Javassist の能力に依存している．たとえば Josh では，ユーザが新たな種類の joinpoint オブジェクトを定義することはできず，Javassist によりあらかじめ提供されているものしか使えない．またアドバイスの挿入可能個所も，before，after，around のいずれかに限られている．これらの制限をなくしていくためには，Javassist 自体を改良する必要がある．

また現在のところ，メソッドボディを検査・改変する能力には制限がある．たとえば joinpoint 間のコントロールフローやデータフローなどの情報を手に入れることはできない．しかしながら，AspectJ が提供する pointcut 指定子は Josh 上で実現可能である．たとえば，AspectJ の cflow のような機能も実装可能である．cflow は，メソッドの開始時（終了時）に増加（減少）するようなスレッドローカル変数を使えば実装で

きる．Javassist を使ってこのようなコードを必要な場所に挿入すればよい．ただし現在の実装では，cflow の引数に別の pointcut 指定子を含むような場合には対応していない．

4. 応 用 例

本稿の動機付けとなった 2.2.2 項の問題は，3.4 節で解決法を示した．この章では 2.2.1 項の問題について述べる．Josh を使うと，この問題を解決する updater 指定子を新たに定義できる．updater 指定子は以下のように使われる．

```
updater("FigureElement", "redraw");
```

この指定子はメソッド呼び出しに対応する joinpoint を抽出する．メソッド呼び出しが抽出されるのは，呼ばれるメソッドが FigureElement のサブクラスに定義してあり，そのクラスの redraw メソッドが参照するフィールドの値をそのメソッドが変更している場合だけである．このような指定子があれば，2.2.1 項の問題は容易に解決される．

このような joinpoint の抽出は，次の static メソッドで実現できる．

```
static boolean updater(MethodCall mc,
        String[] args, JoshContext jc) {
  CtClass root = jc.getCtClass(args[0]);
  String mname = args[1];
  CtMethod mth = mc.getMethod();
  // skip if the method is redraw().
  if (mth.getName().equals(mname))
    return false;

  Hashtable fields =
    enumerateFilds(jc, root, mname);
  updated = false;
  mth.insturment(new ExprEditor() {
    public void edit(FieldAccess expr) {
      String name = expr.getFieldName();
      if (expr.isWriter() &&
          fields.get(name) == expr.getCtClass())
        updated = true;
    }
  });
  return updated;
}
```

このメソッドには，メソッド呼び出し joinpoint オブジェクト，mc が渡される．mc から得られて mth に代入される CtMethod オブジェクトは，呼ばれたメソッドの構造を表す．

ハッシュ表 fields には，redraw メソッドが参照するフィールドの一覧が enumerateFilds メソッドにより記録される．詳細は省略するが，このメソッドもプログラマが自分で定義しなければならない．updater メソッドは，このハッシュ表を使い，mth が表すメソッド内で，ハッシュ表に記録されたフィールドの値が変更されているか否かを調べる．1 つでも変更されていれば，updater は真を返し，mc が表す joinpoint が抽出されることになる．

5. 実 験

Josh の性能測定のために，我々は AspectJ との比較実験を行った．この実験では，XML パーザ Xerces [12] にアドバイスを weave し，weave に要する時間と実行時のオーバヘッドを計測した．アドバイスの内容はすべての public メソッド呼び出しの前に，ログ書き出しとカウンタをいれるものであり，実験に使用したアスペクトは，このアドバイスだけを含む．実験の環境は，Sun Blade 1000（Dual UltraSPARC III 750 MHz, 1 GB メモリ）Solaris 8，Sun JDK 1.4.0_01，AspectJ 1.1 b2 である．

AspectJ はバージョン 1.1 からバイトコードレベルでも weave が可能だが，この実験では Xerces のソースと AspectJ のソースを ajc で weave した．一方 Josh はバイトコードレベルで weave するため，すべてのソースコードは通常の Java コンパイラ（javac）で，あらかじめコンパイルされている必要がある．そのため Josh の weave 時間は，通常のコンパイル時間と Josh 処理系による処理時間との和になる．

756 個のファイルからなる Xerces のソースコードをコンパイルすると，894 個のクラスファイルが生成される．それらと 1 個の Josh アスペクトを weave する時間を計測した．またアスペクトを weave した Xerces で，小規模な XML ファイルの DOM ツリーを作り，その処理の時間を計測した．

結果を**表 3** に示す．Josh の行で括弧内に書かれている割合は，AspectJ の時間を 100%としている．オリジナルとはアスペクトを weave しない，純粋の Xerces のことである．まず weave 時間に関して，Josh は AspectJ の 1.4 倍高速だった．ちなみに AspectJ のコンパイラ ajc で，アスペクトを weave せずに Xerces をコンパイルすると 40.3 秒であった．ここから，AspectJ ではアスペクトの weave に処理時間の大部分を費やしていることが分かる．逆に実行時間では Josh は AspectJ の 1.3 倍である．Josh のアドバイスは joinpoint の位置にインライン展開されるが，その際にメソッド呼び出しの引数がすべてローカル変数として保存されるため，引数の数が多いメソッドがあればあるほど，このオーバヘッドが大きくなる．またインライン展開の弊害としてコード長が AspectJ の 1.5 倍となっている．この実験では 15846 カ所と多くの個所に weave

表 3 Xerces への weave および実行時オーバヘッドの計測
Table 3 Weave and execution overheads on Xerces.

	コンパイル+weave（秒）	実行（ミリ秒）	コード長（KB）
オリジナル	36.2（javac のみ）	408	1928
Josh	77.7（69%）	1106（130%）	4269（153%）
AspectJ	112	881	2787

表 4 Java Grande ベンチマークによる性能比較
Table 4 Performance comparison with Java Grande benchmark.

		Euler	Molecular	Monte Carlo	Ray Tracer	Search
weave 時間（秒）	オリジナル	0.3	0.3	0.4	0.3	0.3
	Josh	3.6	3.4	4.4	3.8	3.7
	AspectJ	8.4	5.7	7.1	6.1	5.8
実行時間（秒）	オリジナル	28.0	22.1	22.7	19.1	17.4
	Josh	29.1	22.1	28.3	29.5	20.1
	AspectJ	28.3	22.1	22.9	19.4	20.7
カウンタの回数		2,307	10,818	42,599	5,338,398	71,228,058

表 5 インタータイプ宣言の weave 時間の計測（秒）
Table 5 Weave time for intertype declaration.

対象のクラス数	4	8	16	32	64
Josh	3.0	3.0	3.1	3.3	3.6
AspectJ	4.1	4.3	4.5	5.1	6.2

しているが，さらに規模の大きな weave もありうる．そのような場合にも対応できるようにすることが今後の課題である．

続いて，ベンチマークテストを使い性能を比較した．テストに用いたのは，Java Grande forum☆の sequential ベンチマークである．weave したアドバイスは，すべてのメソッド呼び出しのターゲットオブジェクトを調べ，それがあるクラスのインスタンスの場合にカウントを 1 増やすというものである．Josh で AspectJ の target 指定子と同等の機能を持つものを実装して，実験を行った．結果は**表 4** のとおりになった．Josh は weave 時間が速いが，実行時間のオーバヘッドがあることが分かった．これは AspectJ の weaver が，最適化をしていることが原因である．target 指定子は，実行時のインスタンスの型を調べてアドバイスを実行するかを決める，動的指定子である．そのため Josh では，プログラム内のいたるところに instanceof 命令が挿入される．一方 AspectJ は，不必要な instanceof 命令を除去するような最適化をしている．事実 Josh は，5 種類のベンチマークプログラムに instanceof 命令を合計 659 カ所挿入しているが，AspectJ は 73 カ所しかしていない．

さらに我々は，インタータイプ宣言の性能測定を行った．Josh では，インタータイプ宣言にもコンテクスト引き渡し機能があり，汎用的な記述ができる．ここでは 2.2.2 項で述べた，Expr の全サブクラスに accept メソッドを追加するアスペクトを，Josh と AspectJ それぞれで記述して，weave 時間を測定した．追加される accept メソッドのボディは，インタータイプ宣言の対象のクラスにより異なる．AspectJ の場合は 2.2.2 項のように全サブクラスを明確に記述する必要がある．Josh の場合は以下のような 1 つの汎用的な記述で，Expr の全サブクラスを対象とした weave ができる．

```
intertype : within("Expr+") {
  void accept(Vistor v) {
    v.visit<% josh.getCtClass().getName() %>(this);
  }
}
```

<%と%>の間の式の結果は，インタータイプ宣言の対象となるクラスによって変わる．たとえば Sum というクラスが対象となっている場合は，<%と%>の間の式は，“Sum” という文字列に評価される．

この実験では，weave の対象となる Expr のサブクラス数を変えて計測した．結果を**表 5** に示す．対象クラス数を増やすとそれぞれの weave 時間も増すが，大きな変化はみられない．Josh では，汎用的な記述ができるうえに weave 時間も短いことが分かった．しかしながら Josh では構文解析を簡略化しており，その分が高速化の要因となっているといえる．たとえば Josh では，Expr などを完全修飾名で記述しなければならない．

6. 関連研究

AspectJ の pointcut 指定子 if を使えば，任意の Java

☆ http://www.epcc.ed.ac.uk/javagrande

の式を pointcut 内に含められる．アドバイスボディは，if 式が真の場合にのみ実行されるので，この指定子は pointcut を拡張するための機能とも見なせる．しかしながら if 指定子は動的指定子であり，コンパイル時に評価される静的指定子に比べ，実行時のオーバヘッドがある．そのため，新しい指定子を定義するときは，できるだけ静的指定子にするほうがよい．Josh では，ユーザが静的，動的の両方の指定子を定義できる．

拡張可能な AOP 言語は他にも存在する．たとえば，複雑な pointcut 記述のために論理型言語を採用した AOP 言語も提案されている[1),5)]．Josh が joinpoint オブジェクトとして提供しているすべての情報を，組み込み述語によって提供すれば，そのような論理型の AOP 言語も Josh と同等の拡張性を提供できると思われる．しかしながら，その場合，組み込み述語の数が膨大になってしまい，何らかのモジュール化の機能が必要になるだろう．Josh はオブジェクト指向言語である Java 言語が提供するモジュール化の機能を使って，この点に対応している．

Kiczales は 2.2 節の問題に取り組むために，pcflow という新しい指定子を提案した[7)]．汎用的な指定子を開発していくことは正しい解決方法であるが，それらの指定子では補いきれないこともあると我々は考えている．

7. ま と め

本稿では AspectJ に代表される現状のアスペクト指向言語の問題について述べた．我々が開発した AOP 言語 Josh では，ユーザが Java 言語で pointcut 指定子を新たに定義できる．この機能により，従来は不可能だった joinpoint の抽出が可能になった．Josh はまた，インタータイプ宣言でもコンテクスト引き渡しの機能が使える．これにより，冗長な記述を減らすことができ，同時に再利用性の高いアスペクトが書けるようになった．

Josh は各 joinpoint に直接対応するオブジェクトを提供し，それらを基本要素として，pointcut 指定を Java 言語で記述できるようにしたとも考えられる．Josh の pointcut 指定子は，Java 言語による pointcut 指定の煩雑さを回避するための，syntax sugar にすぎないといえる．一方，AspectJ の pointcut 指定の基本要素は，pointcut 指定子であり，それを論理演算子で組み合わせて記述する．Josh は AspectJ に比べ細粒度の基本要素を提供するので，より複雑な pointcut 指定を柔軟に記述できるようになった．

現在の Josh では，あるアスペクトに対して別のアスペクトを weave するには，一方のアスペクトを weave した後，weaver を再度起動して他方のアスペクトを weave しなければならない．この一連の処理を同時に行う機能は提供されない．また同じ joinpoint に対して複数のアドバイスがあるときに，それらの実行順序を制御する機能は提供していない．これらは実装上の都合によるが，将来的にどのような仕様にしていくかは検討課題である．

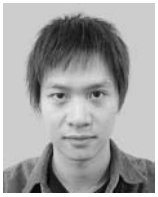
謝辞 本稿の研究の一部は，(独) 科学技術振興機構の戦略的想像研究推進事業「ディペンダブル情報処理基盤」の一環として行われた．

参 考 文 献

1) Brichau, J., Mens, K. and Volder, K.D.: Building Composable Aspect-Specific Languages with Logic Metaprogramming, *Proc. 2nd Int'l Conf. on Generative Programming and Component Engineering*, pp.110–127, Springer-Verlag (2002).
2) Chiba, S.: Load-Time Structural Reflection in Java, *Proc.ECOOP2000*, pp.313–336, Springer-Verlag (2000).
3) Chiba, S. and Nishizawa, M.: An easy-to-use toolkit for efficient Java bytecode translators, *Proc. 2nd Int'l Conf. on Generative Programming and Component Engineering* (*GPCE2003*), pp.364–376. Springer-Verlag New York, Inc. (2003).
4) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
5) Gybels, K. and Brichau, J.: Arranging language features for more robust pattern-based crosscuts, *Proc. Aspect-Oriented Software Development* (*AOSD2003*), pp.60–69. ACM Press (2003).
6) Hanenberg, S. and Unland, R.: Parametric introductions, *Proc. Aspect-Oriented Software Development* (*AOSD2003*), pp.80–89, ACM Press (2003).
7) Kiczales, G.: The Fun Has Just Begun, *Keynote talk at 2nd Int'l Conf. on Aspect-Oriented Software Development* (*AOSD 2003*).
8) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *ECOOP 2001*, LNCS 2072, pp.327–353, Springer (2001).
9) Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C.V., Maeda, C. and Mendhekar, A.: Aspect-Oriented Programming, *ECOOP 1997*, LNCS 1241, pp.220–242. Springer (1997).

10) Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, *Proc. ECOOP2003*, LNCS 2743, pp.2–28, (2003).
11) Sun Microsystems, Inc.: *Java^{TM} Core Reflection API and Specification* (1997).
12) Xerces-2.6.0. http://xml.apache.org/xerces2-j/index.html

(平成 15 年 12 月 20 日受付)
(平成 16 年 2 月 24 日採録)



中川 清志

1979 年生．2002 年東京工業大学理学部卒業．2004 年東京工業大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．アスペクト指向プログラミング，システムソフトウェアに関する研究に従事．

千葉 滋（正会員）

1968 年生．1991 年東京大学理学部情報科学科卒業．1996 年東京大学大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師を経て，現在，東京工業大学大学院情報理工学研究科助教授．博士（理学）．言語処理系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研究に従事．日本ソフトウェア科学会，ACM 各会員．