2003年度修士論文

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

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

学籍番号 02M37220

中川 清志

指導教官

千葉 滋 助教授

2004年2月6日

概要

本論文では、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 とはロギング、同期、永続性などの複数モジュール間に散らばる処理をアスペクトとしてモジュール化するプログラミング方法である。分離して記述されたアスペクトと、クラスなどの基本モジュール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。その際にアスペクトは weaver に、合成個所と合成内容の 2 つを指示する。

合成個所を指示するものは pointcut と呼ばれ、pointcut にはプログラム内の様々な情報が必要である。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 weave 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。本論文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 指定子の組合わせしか使えない言語よりも、Josh の pointcut 記述力は高い。

また合成内容の指示に関しても Josh は高い記述力を持つ。アスペクトはインタータイプ宣言と呼ばれる、他のクラスに新たなフィールドやメソッドを追加する機能を持つ。Josh では、一つのインタータイプ宣言を複数クラスに同時に適用でき、その際に追加される内容が対象クラスしだいで異なるようにプログラムを記述できる。

さらに本論文では 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などで計測し、その結果を AspectJ と比較する。

謝辞

指導教官の、東京工業大学大学院 情報理工学研究科 数理・計算科学専攻の千葉 滋 助教授には、私が千葉研究室に所属している 3 年間にわたって、非常に親身に指導していただいた。本研究においては、研究の方針やプログラムコード、論文の書き方など実にさまざまな点で助言をいただいた。また、本論文に限らず私が執筆した論文を技術面・文法面の両方に関して精細に添削していただいた。さらに、国内外の会議への参加を積極的に勧めてくださり、多くの研究者との交流の場をもうけていただいた。そこから得た知識や刺激が、本研究を進める原動力となった。ここに深く感謝の意をあらわす。

東京工業大学大学院 情報理工学研究科 数理・計算科学専攻の光来 健一 助手には、本研究について適切な助言をいただいた。また私と同じく千葉研究室に在籍する 佐藤 芳樹 氏と 西沢 無我 氏とは、類似した研究題目を持ち、本研究の題材について深い議論をしていただいた。そして昼夜を問わずともに学んできた千葉研究室の方々には、学びやすい環境を提供していただいた。ここに深く感謝の意をあらわす。

目 次

図 目 次

表 目 次

第1章　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言われている。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) と言われる。アスペクト指向プログラミング (AOP)[12] 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示すのが *pointcut* であり、プログラム内の様々な個所を指定できる。

数多くの AOP 言語が提案されており [1, 15, 16]、その中の代表的なものが AspectJ[11] 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 [8] 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 にはそのような機能はない。

これらの問題に対処するために、本論文では AOP 言語 *Josh* を提案する。Josh は pointcut 言語の記述力が高いのが特徴である。pointcut は pointcut 指定子と呼ばれる記述子の組合わせで構成されるが、AspectJ ではこの pointcut 指定子の数が限られている。また指定子はそれぞれ、固有のアルゴリズムに基づいて pointcut 先を特定するが、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 章で関連する研究と AspectJ をとりあげ、その問題点をあげる。3 章で Josh を提案し、続いて 4 章では Josh の拡張方法を示すとともに問題の

解決法を示す。5 章では性能を評価し、最後に 6 章で本論文をまとめる。

第2章　関連研究

この章ではまずアスペクト指向の概念について説明し、次に AOP 言語の代表的なものである AspectJ について述べる。そして AspectJ を使ったとしても、解決が困難な問題を提起する。

2.1　アスペクト指向

オブジェクト指向とは、プログラムをオブジェクト (もの) としてモジュール化するプログラミング技法であり、それぞれのオブジェクトはある機能を表している。オブジェクト指向は様々な機能をカプセル化する能力に優れており、保守性、拡張性、可読性の高いプログラムを記述できるといわれている。しかしながら、ある種の機能のための処理群は複数オブジェクトに散らばってしまうことがある。これらの処理群は横断的関心事 (crosscutting-concern) と呼ばれ、上手にオブジェクトにモジュール化できないことが問題視されている。これらの機能をを修正するためには、プログラムの多くの個所を修正する必要が出てしまい、その結果一つの機能が一つのプログラムで表現されるという簡潔性が損なわれ、質の低いプログラムになってしまう。

アスペクト指向プログラミング (AOP) は、この問題を解決するためのプログラミング技法である。AOP 言語では複数のオブジェクトに散らばってしまう機能を、オブジェクトとは別の側面から考慮し、それをアスペクトとしてモジュール化して扱える。特にアスペクトはシステマティックな機能を扱うことが多い。それらにはログ処理、同期制御、永続性、例外処理、トランザクションなどがある。

AOP にはこのように、基本モジュールであるオブジェクト (クラス) と、別の側面からの機能を持つアスペクトの二種類のモジュールが存在する。これら別個の意味合いを持つものを合成し、両方の機能を持つモジュールにすることを **weave** といい、その処理系を **weaver** という。この処理によりアスペクトとして分離してモジュール化されていた処理群を、複数のオブジェクトに同時に組み込むことができる。アスペクトには weaver の制御に関するコードが記述されており、具体的にはオブジェクト内のどの個所にどういったコードを合成するかについてが記述されている。

2.2　AspectJ のプログラミングモデル

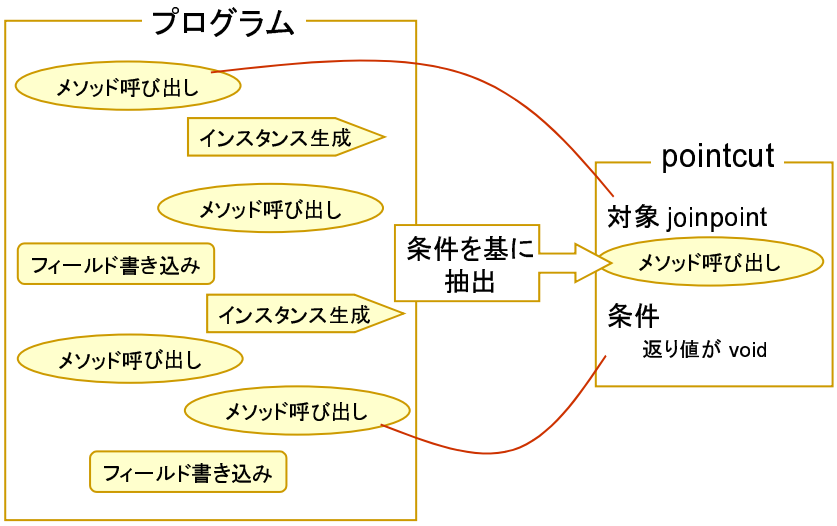
AspectJ は標準的な汎用 AOP 言語の一つであり、Java にアスペクト指向を取り入れて拡張したものである。AOP 言語は数多く提案されているが、その中で AspectJ は代表的なものであり、現在もっとも幅広く使われている。そのため AOP 独自の概念を説明する際には、AspectJ の用語を使うことが多く、本論文も同様である。この節では AspectJ のプログラミングモデルを説明するとともに、AOP を理解するのに必要な概念を説明する。

2.2.1　pointcut

joinpoint とは、プログラム内の演算で、AOP 言語がコードを合成しうる個所のことである。AspectJ にはメソッド呼び出し、フィールド参照、インスタンス生成などがある。joinpoint の種類は言語によって異なる。

joinpoint を抽出する作業のことを *pointcut* といい、コードを合成したい個所を具体的に特定するものである。AOP では、アスペクトはオブジェクトに合成されるが、pointcut はオブジェクト内の個所を weaver に支持する。その個所で何を実行するかに関しては、pointcut は関与しない。これは抽出の対象とする joinpoint の種類と、そこに課す条件の対で表せる [図 2.1]。

図 2.1: pointcut の模式図



例えば「int 型の返り値で名前が setX、引数を持たない Point クラスで宣言されている」という「メソッド呼び出し」を抽出したい場合、AspectJ では以下のように記述する。

```
call(int Point.getX())
```

call は pointcut 指定子と呼ばれるものであり、メソッド呼び出し joinpoint を抽出の対象としている。括弧内に条件を記述することにより、プログラム内のどのメソッド呼び出しを抽出するかを特定している。

この他にも AspectJ には数多くの pointcut 指定子が存在する。

call 特定したメソッド呼び出しを抽出

set 特定したフィールド読み込みを抽出

get 特定したフィールド書き込みを抽出

initialization 特定したクラスのインスタンス生成を抽出

execution 特定したメソッド実行を抽出

handler 特定した例外のハンドラを抽出

staticinitialization 特定したクラスの静的初期化子の実行を抽出

AspectJ の pointcut 指定子で抽出できる演算は、以上の 7 種類である。このうち上からの 6 つまでの指定子が対象としている演算を AspectJ における基本 joinpoint と呼ぶことにする。これらの pointcut 指定子は joinpoint の種類を対象としていたが、joinpoint の位置を対象として weave 個所を特定する指定子もある。

within 特定したクラス内の全ての基本 joinpoint、またはそのクラスの静的初期化子

withincode 特定したメソッド内の全ての基本 joinpoint

また、プログラムの実行時の情報を基にした pointcut も存在する。以下の 3 種類がそれに相当する。これらの指定子は、アドバイスという機能において joinpoint の情報を受け渡すためにも使われる。詳細は後述する。

target 特定したクラスのオブジェクトをターゲットとした全ての基本 joinpoint

this 特定したクラスのオブジェクト内で実行した全ての基本 joinpoint

args 特定した引数で実行する全ての基本 joinpoint

さらに AspectJ には、プログラムのフローを意識した pointcut 指定子もある。また、if 指定子は意の式を条件として pointcut をすることができる。以下の 3 種類も実行時の情報を基にする指定子である。

cflow 特定した pointcut のフローの中にある全ての基本 joinpoint

cflowbelow 特定した pointcut のフローの中にある全ての基本 joinpoint、ただし pointcut とで指定された最初の演算は含まない

if 任意の式を含む、

pointcut 指定子には複数の joinpoint を一度に抽出できるようにワイルドカードを使えるものがある。以下がワイルドカードを使った例である。

```
get(* Point.*)
call(public void Point.set*(..))
```

1 つ目は get 指定子を使っておりフィールド読み込みを抽出する。条件は Point クラスで宣言される任意の型で任意の名前のフィールドである。2 つ目は call 指定子を使っておりメソッド呼び出し読み込みを抽出する。条件は Point クラスで宣言される void 型で修飾子が public、名前が set で始まり引数は任意のメソッドである。引数部の [..] は、引数の型も数も任意でよいという意味である。

また pointcut 指定子は論理演算子で組み合わせることができ、&&(論理積)、||(論理和)、!(否定) の 3 種類の論理演算子がある。これは例えば、

```
call(public void Point.set*(..)) && !within(Point)
```

のように使う。先ほどと同じように Point クラスで宣言されているメソッド呼び出しを抽出するが、Point クラス内で起きた演算は抽出しない。

2.2.2　アドバイス

pointcut で抽出された複数または単数の joinpoint において、新たなコード断片を実行することを指示する機能をアドバイスという。この際実行されるコード断片をアドバイスボディと呼ぶ。アドバイスには before, after, around の 3 種類があり、ボディを実行するタイミングが異なる。それぞれ pointcut された joinpoint 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.");
}

before(int newx)
    : call(void Point.setX(int)) && args (newx){
    System.out.println("setting x in Point. " +
                       "new x = " + newx);
}
```

のようになる。1 つ目は 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。まず、before の部分でこのアドバイスの実行タイミングを指示する。次にコロンに続くのが pointcut 部であり、プログラム内のどの個所でアドバイスを実行するかを指示する。最後に中括弧で囲まれた部分がアドバイスボディであり、何を実行するかを指示する。アドバイスボディには任意の Java 言語の式を記述できる。

2 つ目では**コンテクスト引き渡し**という機能を使っている。これはアドバイスボディの中で、joinpoint の実行時の情報を参照するためのものである。まずメソッドの仮引数を定義するように before(int newx) と記述する。そして args を使い、joinpoint のどの情報が newx に束縛されるかを指示する。この場合は call 指定子とともに使っており、メソッド呼び出しの第一番目の実引数値が束縛される。こうして宣言した newx という変数をアドバイスボディ中で使うことができる。AspectJ ではこの他に target と this 指定子のこの機能がある。call の条件であるメソッドの引数の数と args の引数が異なっている場合は、どのメソッド呼び出しも抽出されない。

2.2.3　インタータイプ宣言

既存クラスに新たな要素を追加することをインタータイプ宣言という。追加できる要素には、フィールドメソッド、コンストラクタがある。また既存クラスの階層構造を変えることをインタータイプ宣言といい、親クラスの変更とインタフェースの追加ができる。これは従来イントロダクションと呼ばれていたものと同等である。以下に具体例を示す。

```
private int Point.z = 5;
public int Point.getZ() {
    return z;
}
declare parents : Point implements java.io.Serializable;
declare parents :
    (Point || Line || Rectangle) implements Comparable;
private Strnig Figure+.name = "Figure";
```

1 つ目は、修飾子が private で int 型の z という初期値が 5 のフィールドを Point クラスに追加する。2 つ目は、修飾子が public で返り値が int 型の、引数を持たない getZ という名前のメソッドを Point クラスに追加する。メソッドボディは中括弧でくくられている部分である。3 つ目はクラスの階層構造を変えるもので、Point クラスが java.io.Serializable インタフェースを継承するようにする。4 つ目と 5 つ目は、複数クラスを同時にインタータイプ宣言の対象としている。4 つ目は || で連結された 3 つのクラスが Comparable インタ

フェースを継承するようにし、5 つ目は Figure のサブクラス全てに name というフィールドを追加する。

AspectJ の使用では明確の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。上記の例のインタータイプ宣言対象は、上から 3 つでは Point クラスのみであるが、4 つ目と 5 つ目では複数クラスを対象にしている。4 つ目では Point, Line, Rectangle の 3 クラスが対象であり、それら全てにインタータイプボディを適用する。AspectJ のインタータイプ宣言にはコンテクスト引き渡しの機能がなく、weave の対象となったクラスの情報を参照することができない。また複数クラスを対象としていても、インタータイプボディは同一のものでなければならない。

2.2.4　アスペクト

横断的関心事をモジュール化したもののことであり、関連する pointcut とアドバイスとインタータイプ宣言が集まっている。またアスペクト自体のメンバとなるフィールドとメソッドを宣言することもできる。以下のプログラムが AspectJ のアスペクト全体を記述した例である。

```
01 aspect CounterAspect {
02     public static void print(int i) {
03         Sysetm.out.println(i);
04     }
05     int Point.count = 0;
06     pointcut pc() : call(public * Point.*(..));
07     before() : pc() {
08         count++;
09         CounterAspect.print(count);
10     }
11 }
```

1 行目の aspect という宣言でこれがアスペクトの役割を果たすことがわかる。これはオブジェクト指向の class という宣言に類似している。2-4 行目はアスペクトメンバとしてメソッドを宣言している。AspectJ の処理系はアスペクトと同名のクラスを生成し、その中でこのメソッドが定義される。5 行目がインタータイプ宣言であり Point クラスに count というフィールドを追加する。6 行目で pc という名前の pointcut を定義している。pointcut はこのようにアドバイスと独立して定義することも可能である。7-10 行目がアドバイスであり、Point クラスの public メソッド呼び出しの前にアドバイスボディ実行するように指示する。ボディは、count を 1 増やすコードと CounterAspect 内の print メソッドを呼び出す。この例ではアスペクトの weave 対象は Point クラ

スだけに限られているが、多くのクラスを weave 対象とする場合もある。そのような場合でも、カウントのタイミングや方式の変更時には CounterAspect のみを修正すればよく、weave 対象である Point クラスなどを修正する必要はない。

2.3　AspectJ の記述力の限界

現在の AspectJ(バージョン 1.1) 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。この節ではこれら 2 つの問題点を、例を挙げて指摘する。

2.3.1　pointcut 記述の問題点

ここでは図形エディタの例を使い、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 2.2]。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 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。

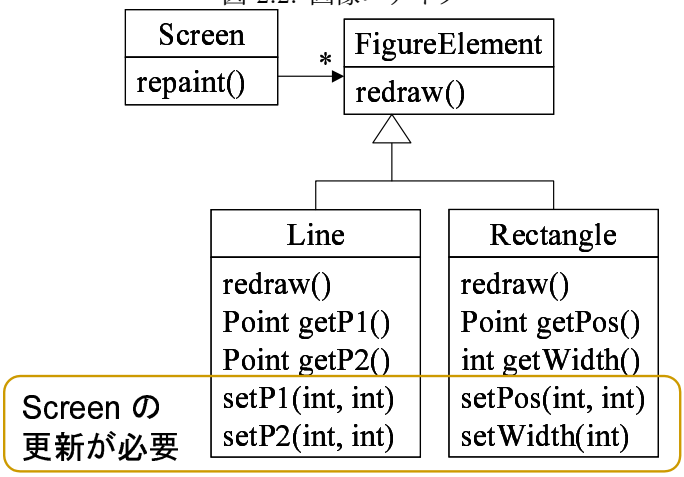
しかし、このような再描画処理を実現するには、各図形クラスの全ての setWidth のような図形の外観を変更するメソッドの中に、忘れずに repaint メソッドの呼び出しを書かなければならない。このようなメソッドの呼び出しは、各図形クラスのメソッド内に散らばってしまうことになり、横断的関心事となってしまう。

AspectJ を使えばこのような関心事は以下のようにモジュール化できる。

```
aspect UpdateAspect {
    pointcut toUpdate() : call(public void Figure+.set*(..));
    after() : toUpdate() {
        // Screen の再描画を指示
    }
}
```

まずプログラマは 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の全てを抽出するように、pointcut を定義する。次に抽出したメソッド呼び出しの後に、Screen の repaint を呼ぶように、アドバイスボディを定義する。この関心事のモジュール化には、全ての

図 2.2: 画像エディタ



図形クラスの外観変化を起こすメソッドを抽出する pointcut を記述することが必要である。

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る。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全て数え上げるか、もしくはそのようなメソッドは全て"set"で始まるというようなルールを作り従わなければならない。そうすれば上記のようなワイルドカードを使った pointcut で記述できる。しかしながら、そのようなルールは煩わしく、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、全てのメソッドが正しく抽出されていることを保証する機能は、AspectJ にはない。

これを解決するためには、より細かい条件に基づいた、メソッド抽出が記述ができればよい。例えばそれは以下のようである。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処理をするメソッドである。この特徴を利用して、メソッドを抽出する条件を記述できればよい。

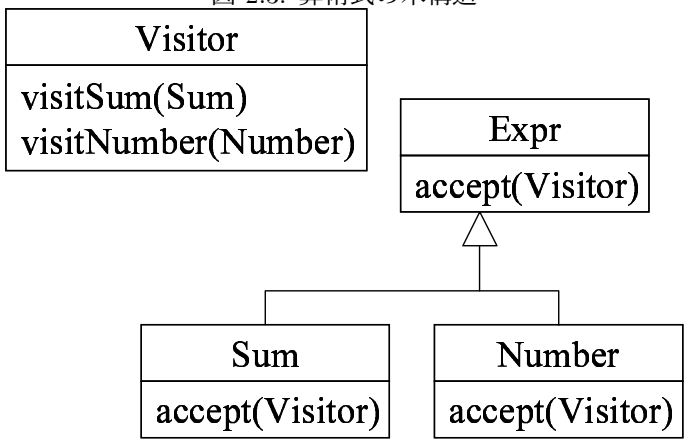
しかしながら AspectJ では、そのような特徴を条件とする pointcut を記述できない。これは AspectJ の pointcut を構成する pointcut 指定子が限られているからである。pointcut 指定子は、それぞれ固有のアルゴリズムに基づいて pointcut 先を具体的に特定するものだが、指定子の種類は AspectJ 言語が仕様で与えているものだけであり、限られている。そのためプログラマが独自のアルゴリズムに基づいて、pointcut 先を特定したいとしても、そのアルゴリズムに一致する指定子がない場合には、pointcut を記述できない。例えば AspectJ には、先ほど示したような条件を満たす指定子はないが、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。

2.3.2　インタータイプ宣言の汎用性

次の例として、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 [6] に沿って、図 2.3 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。

図 2.3: 算術式の木構造



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。例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、繰り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 クラスの全てのサブクラスに、accept メソッドを追加する。ここではまず this オブジェクトのクラス名を手にいれ、それを"visit"とつなげる。次につなげられた文字列を表す Method オブジェクトを手にいれる。そして、this を含む Object クラスの配列をつくる。最後に、引数 v と先ほどの配列を使って Method オブジェクトを invoke する。しかし、この accept メソッドは Java のリフレクション [17] 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。

2.4　pointcut 記述の拡張

AspectJ の pointcut 指定子 if には任意の Java の式を含むことができる。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みなせる。しかしながら if 指定子は、target 指定子のようにプログラムの実行時情報を基にする。そのためコンパイル時に評価される指定子に比べ、実行時のオーバヘッドがある。

また、AspectJ 言語に新たに有用な pointcut 指定子を組み込んだ研究もある。増原らは dflow という新たな指定子を提案した [14, 19]。この指定子はデータの依存関係に基づく pointcut を記述できる。他にも Kiczales は pcflow という新しい指定子の必要性を述べた [10]。新たに指定子を開発していけばとりあえずの問題は回避できるが、しかしながらプログラマの望む指定子が常に実装されているとは限らない。

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 [2, 7]。確かに、論理型言語は複雑な pointcut の条件の記述に役立つが、プログラマは慣れない論理型言語を学ばなければならない。

第3章　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 の全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

Josh の文法は AspectJ をもとにしており、大部分は類似している (第 2.2.4)。以下のプログラムは Josh で書かれたカウンターアスペクトの例である。

```
01 aspect CounterAspect {
02     public static void print(int i) {
03         Sysetm.out.println(i);
04     }
05     intertype : same("Point") {
06         int count = 0;
07     }
08     before : call("public * Point.*(..)") {
09         count++;
10         CounterAspect.print(count);
11     }
12 }
```

まず 1 行目でアスペクトであることを宣言している。2-4 行目はアスペクトメンバのメソッドである。8-11 行目がアドバイスであるが、AspectJ と違い call の引数がダブルクォートで囲まれている。また、before の後ろにコンテクスト引き渡しのための括弧がない。この理由は後に述べる。

アドバイス記述が類似しているが、Josh のインタータイプ宣言は AspectJ のそれと異なっている。5-7 行目が Josh のインタータイプ宣言であり、Point クラスに int 型の count というフィールドの追加を指示する。これは、

["intertype"　":"　pointcut　インタータイプボディ]

という並びになっている。このように Josh のインタータイプ宣言では、pointcut とボディを明確に分けて記述する。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。

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 weave 対象を特定するものでなく、ボディの一部である。

先ほど述べたように、アスペクトにはそれ自身に属するアスペクトメンバを定義することもできるが、定義できるのは現在のところフィールドとメソッドだけである。またアスペクトのインスタンスを生成することもできないため、メンバは static である必要があり、コンストラクタの定義もできない。入れ子クラスなどもサポートしていない。

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*[4] により提供されている。

この特別な変数を使った例を示す。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一引数、第二引数、... を表す。これらは AspectJ の args と類似の機能を実現する。$1, $2 などを使ったプログラムは、以下のようになる。

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

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。$1 と$2 の型は、call 指定子とその引数次第で変わる。pointcut で抽出されるメソッドの引数を考慮して$1 などを使わなければならない。引数が 1 つのメソッド呼び出しに対するアドバイスボディ内で、$2 を使うとコンパイルエラーになる。

別の変数も提供されている。$args は引数の配列であり、型は java.lang.Object である。そのため int のようなプリミティブ型のものは、自動的に Integer 型のようなラッパ型に変換される。$sig(signature) も引数の配列であるが、こちらは java.lang.Class 型のオブジェクトの配列である。$type は java.lang.Class

型のオブジェクトであり、返り値と同様の型になっている。$class も同様に java.lang.Class 型のオブジェクトであり、こちらは演算のターゲットオブジェクトの型である。

さらに around アドバイスのボディ内では、$proceed を使用できる。これは AspectJ の proceed と同じである。around アドバイスはある特定の演算を捉え、その演算の実行の代わりにアドバイスボディを実行する。もし、本来実行するはずだった演算を実行したい場合には、$proceed を呼べばよい。例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 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、例えば異なる引数を持つメソッド呼び出しを pointcut 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 の joinpoint モデル

ここで Josh における joinpoint および pointcut という用語の定義をする。本論文ではこの定義に沿って議論をする。*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かかわるものであり、クラス、メソッド、フィールドなどがある[1]。

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、そこに課す条件の対で表せる。pointcut は pointcut 指定子で記述される。例えば AspectJ の指定子に call と

[1] 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

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。例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。例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[2]。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。

3.2.1　Joinpoint オブジェクト

Java のプログラム内にある各 joinpoint を、Josh 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、リフレクションでいうメタオブジェクトに相当する。リフレクション (自己反映計算) とは、プログラムにそれ自身のモデルを計算対象として与え、プログラムが自身を変更・修正できるようにするための技術である [20, 22]。リフレクションの対象となるのはクラスや関数、レコードなどのデータ構造であり、それらが計算対象としてモデル化されたものをメタオブジェクトという。Java は標準 API(Application Programming Interface) の一部としてリフレクションの機能を提供するプログラミング言語である [17]。しかしながら提供される機構はプログラムの定義を調べることだけである。

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。これらは標準の Java リフレクション API にある Class や Field などと類似しており、構造を調べるためのメソッドが提供されている。[表 3.1]。

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。例えば 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。

[2] target は実行時の型に依存するため静的指定子ではない

表 3.1: CtClass クラスのメソッド (抜粋)

	内観用	
String	getName()	クラス名を得る
int	getModifiers()	修飾子を得る
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
CtConstructor[]	getConstructors()	コンストラクタを得る
	改変用	
void	setName(String)	クラス名を変える
void	setModifiers(int)	修飾子を変える
void	setSuperclass(CtClass)	親クラスを変える
void	setInterfaces(CtClass[])	インタフェースを変える
void	addField(CtField)	フィールドを追加する
void	addMethod(CtMethod)	メソッドを追加する
void	addConstructor(..)	コンストラクタを追加する

表 3.2: CtMethod クラスのメソッド (抜粋)

	内観用	
String	getName()	メソッド名を得る
int	getModifiers()	修飾子を得る
CtClass	getDeclaringclass()	所属クラスを得る
CtClass	getReturnType()	返り値の型を得る
CtClass[]	getParameterTypes()	引数を得る
CtClass[]	getExceptionTypes()	投げうる例外を得る
	改変用	
void	setName(String)	メソッド名を変える
void	setModifiers(int)	修飾子を変える
void	setExceptionTypes(CtClass[])	投げうる例外を変える
void	insertBefore(String)	ボディの前に新命令を挿入する
void	insertAfter(String)	ボディの後に新命令を挿入する
void	setBody(String)	ボディを変える
void	instrument(ExprEditor)	ボディを変える

表 3.3: 演算 joinpoint オブジェクト

joinpoint オブジェクト名	演算の種類
MethodCall	メソッド呼び出し
FieldAccess	フィールド参照
NewExpr	インスタンス生成
Cast	キャスト式
Handler	例外ハンドラ
Instanceof	instanceof 式

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 6 種類の演算 joinpoint オブジェクトがある [表 3.3]。これらのクラスにも、joinpoint の情報を得るメソッドがある [表 3.4]。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。

表 3.4: MethodCall クラスのメソッド (抜粋)

	内観用	
CtClass	getCtClass()	ターゲットのクラスを得る
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を含むメソッドを得る
int	getLineNumber()	ソースコードの行番号を得る
String	getFileName()	ソースファイル名を得る
	改変用	
void	replace(String code)	このメソッド呼び出し演算を置き換える

3.3 Joinpoint オブジェクトの操作による AOP 機能の実現

Josh の joinpoint オブジェクトは Java 言語用の構造リフレクションのためのライブラリ、*Javassist*[3, 21] により提供されている。Javassist は通常の Java のリフレクションを強化したものであり、joinpoint オブジェクトを通してプログラムの定義の内観・改変をできる、つまり AOP と同等の機能を実現できる。この節では Javassist を使い、どのように joinpoint オブジェクトを操作するかを説明する。joinpoint オブジェクトは、プログラム内でプログラムの構造を扱うためのものであり、メタオブジェクトとも呼ばれる。

3.3.1　インタータイプ宣言の実現

インタータイプ宣言とは既存クラスに新たな要素を追加することであった。以下のコードは Javassist を用いたインタータイプ宣言の例であり、順を追って説明する。

```
01 ClassPool pool = ClassPool.getDefault();
02 CtClass cc = ClassPool.getDefault().get("Point");
03 CtField cf = new CtField(CtClass.intType, "z", cc);
04 cc.addField(cf);
```

Javassist を利用する第 1 段階は、JVM にロードするクラスファイル (に対応するクラス) を表す CtClass(compile-time class) オブジェクトを生成し、構造リフレクションを適用できるようにすることである。リフレクションの操作によりクラス構造は改変されるので、これは言い換えると weave 処理をするといえる。

まず 1 行目において ClassPool オブジェクトを入手し、それを使い 2 行目で Point クラスを表す CtClass オブジェクトを入手する。ClassPool オブジェクトは CtClass オブジェクトの入手などに使う。3 行目ではフィールドを表すメタオブジェクトである CtField オブジェクトを生成する。ここでは int 型で z という名前のフィールドを表すメタオブジェクトである。この段階ではこのメタオブジェクトが表すフィールドは、どのクラスにも所属してないが、将来的に所属するクラス cc が 3 つ目の引数になっている。このようにまったく新しいメタオブジェクトを生成することもできる。そしてそのフィールドを 4 行目でクラスに追加する。

ここまでの処理はメモリ上でおこなわれており、実際に処理を反映させるには以下のようにする。

```
cc.writeFile();
```

このようにすればフィールドの追加が実現できる。

フィールドに初期値を与えるには CtField.Initializer クラスを使う。以下のコードは int 型のフィールドを初期値 5 で追加する例である。

```
CtField cf = new CtField(CtClass.intType, "z", cc);
CtField.Initializer init = CtField.Initializer.constant(5);
cc.addField(cf, init);
```

cc は先ほどと同じように CtClass オブジェクトである。このオブジェクトにフィールドを追加する際に CtField.Initializer オブジェクトを同時に渡すと、初期値が設定される。CtField.Initializer 内のメソッドは、全て static で返り値が CtField.Initializer 型の、ファクトリークラスである [表 3.5]。

表 3.5: CtField.Initializer クラスのメソッド (抜粋)

メソッドの種類 初期値
byCall(CtClass methodClass, methodName) static メソッド呼び出しの結果
byCall(CtClass methodClass, methodName, String[] params) static メソッド呼び出し (引数付き) の結果
byNew(CtClass objectType) コンストラクタ呼び出しの結果
byNewArray(CtClass type, int size) 大きさが size で type 型の配列
constant(double d) d で表される実数
constant(int i) i で表される整数
constant(String s) s で表される文字列
byExpr(String source) source で表される任意の Java 式の結果

メソッドの追加も同様にメタオブジェクトを操作しておこなう。以下がその例である。

```
01 String source = "public int getX() { return x; }";
02 CtMethod cm = CtNewMethod.make(source, cc);
03 cc.addMethod(cm);
```

cc は CtClass オブジェクトである。1-2 行目で、メソッドを表すメタオブジェクトを生成している。メソッドのボディにはテキストでソースコードを渡せばよい。CtNewMethod クラスは CtMethod オブジェクトを生成するためのメソッドを数多く提供している。そして 3 行目で cc に追加処理をしている。コンストラクタも同じようにして追加できる。

またインタータイプ宣言には、親クラスやインタフェースを変える機能もあった。これも Javassist で実現できる。

```
01 CtClass inter = pool.get("java.io.Serializable");
02 CtClass superClass = pool.get("Figure");
03 cc.setSuperClass(superClass);
04 cc.addInterface(inter);
```

1 行目で追加したいインタフェースを表すメタオブジェクトを、2 行目で新しく親クラスとなるクラスのメタオブジェクトを入手する。そして 3-4 行目でそれらを処理している。cc はリフレクションの対象、つまり weave 対象のクラスである。

3.3.2　アドバイスの実現

javassist.expr パッケージ内の ExprEditor クラスには、メソッドの振る舞いを変える機能があり、これによりアドバイスを実現できる。それには以下のようにする。

```
ExprEditor editor = new ExprEditor();
cm.instrument(editor);
```

ここで cm は CtMethod オブジェクトである。instrument メソッドは、引数の ExprEditor オブジェクトを使ってメソッドの振る舞いを変える。まず instrument メソッドはメソッド内の演算を一つ一つ調べて、演算 joinpoint オブジェクトのどれかを探す [表 3.3]。次にその joinpoint オブジェクトを ExprEditor の edit メソッドに渡す。edit メソッドは 6 種類の演算 joinpoint 全てを受け取れるように、オーバーロードして定義されている [図 3.1]。edit メソッド内で演算 joinpoint オブジェクトを操作することにより、演算の動作を改変する。この操作がメソッド内の全ての演算 joinpoint に対しておこなわれる。その際

edit メソッドに渡される joinpoint オブジェクトは、その joinpoint の情報を全て持っており、参照可能である。

図 3.1: ExprEditor クラスの概要

```
public class ExprEditor {
    public void edit(MethodCall m) {}
    public void edit(FieldAccess f) {}
    public void edit(NewExpr n) {}
    public void edit(Cast c) {}
    public void edit(Handler h) {}
    public void edit(Instanceof i) {}
}
```

しかしながら ExprEditor クラス自体は、java.awt パッケージのイベントリスナーのアダプタークラスのように何も仕事はしない。オーバーロードされている 6 個の edit メソッドは、全て空である [図 3.1]。そのためユーザは必要に応じてサブクラスを定義して、メソッドをオーバライドしなければならない。ExprEditor は以下のように継承して使う。

```
01 public class SampleEditor extends ExprEditor {
02     public void edit(MethodCall mc) {
03         String name = mc.getMethodName();
04         if (!name.equals("getX"))
05             return;
06         String source =
07             " { System.out.println(\"getX() is caled.\"); "
08             + " $_ = $proceed($$); }";
09         mc.replace(source);
10     }
11 }
```

ここでは 2-4 行目で edit(MethodCall m) メソッドをオーバライドしている。この edit メソッド内では、まず 3 行目で呼び出されたメソッドの名前を得る。次に 4-5 行目で、その名前が”getX”かどうかを調べ、偽の場合はそこでこのメソッドを終了する。6-8 行目でこのメソッド呼び出しを置き換えるコードを作成し、9 行目で実行する。MethodCall や FieldAccess などの演算 joinpoint オブジェクトには、replace(String) というメソッドがあり、演算の挙動を置き換える機能を持つ。replace で置き換えるコードには任意の Java 式を含むことができるが、2 つ以上の式をつなげる場合には中括弧でくくる必要がある。$_ = $proceed($$) は、replace の中で使える特別な命令であり、本来の演算を

実行する。これは AspectJ の proceed に相当する。その他にも replace の中では、$0, $1 や$args などの変数を使用できる。Josh は Javassist のこの機能を利用しているので、第 3.1 で説明したものと同じである。

この edit メソッドは pointcut とアドバイスの役割を果たしている。ここでは MethodCall の名前を調べて、その後の処理をするか否かを決めている。つまり条件と joinpoint を比較して抽出作業をしており、これは pointcut といえる。また replace を使い、抽出された joinpoint のコードを置き換えており、これはアドバイスといえる。上のコード例では$proceed の前に、プリント命令が入っているので before アドバイスである。プリント命令と$proceed の順番を入れ替えれば、around アドバイスになる。SampleEditor クラスは、メソッド呼び出し以外の演算を受け取る edit メソッドをオーバーライドしていないので、その他の joinpoint に関しては何も処理をしない。

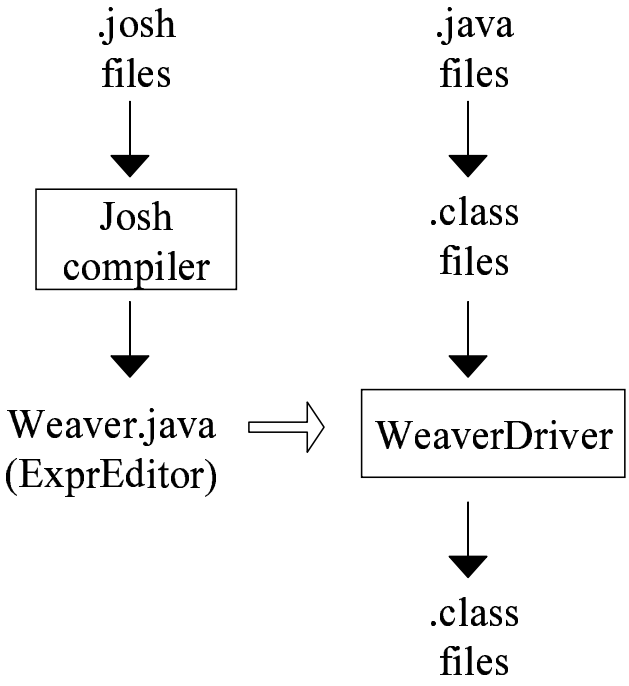
3.4　Josh コンパイラ

構造リフレクションを使えば、AOP の機能を実現することはできることを示した。Josh コンパイラは、Josh 言語で書かれたアスペクトを Javassist を使ったプログラムに変換し、AOP をおこなう。構造リフレクションを使ったプログラミングは強力であるが直接記述するのは困難である。また記述できるとしても、アスペクトの構造が明確になるようにプログラムを書けることは重要である。例えば C のような手続き型言語を使ってオブジェクト指向風なプログラミングも不可能ではない。しかしオブジェクト指向で設計されたシステムを C 言語で実装するのは不自然である。熟練プログラマであったり細かな技法をたくさん用いたりすれば可能かも知れないが、全てのプログラマにそれらを要求できない。また C 言語のコードを見てもオブジェクト指向の設計を読み取れない。そのためアスペクト記述に特化した言語は必要であるといえ、Josh ではアスペクトを一旦 Josh 言語で記述し、それを変換するという手法を採用している。

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 3.2]。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 WeaverDriver 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。

Josh の weaver の実体は Javassist の ExprEditor である。Josh コンパイラは、アスペクト内の pointcut は、それが対象とする joinpoint オブジェクトを受け取る edit メソッド内の if 文に変換される。アドバイスボディは replace メソッドの引数に変換され、pointcut が真のときにのみ実行されるようにな

図 3.2: Josh コンパイラの流れ



る。edit メソッドには次々と joinpoint オブジェクトが渡されるが、それぞれで持っている値が違うので、それらの情報を参照して pointcut ができる。

WeaverDriver は weaver を外部から動かす役割を持つ。その際に weave 中のクラスの情報などを weaver に渡し、weave にはその情報も使われる。WeaverDriver は以下のようにできている。

```
class WeaverDriver {
    JoshExprEditor editor;
    ClassPool pool;
        :
    public void weave(String className){
        CtClass target = pool.get(className);
            //weave 対象クラス
        CtMethod[] methods = target.getDeclaredMethods();
        JoshContext jc = new JoshContext();
        editor.setContext(jc);

        jc.currentClass = target;
        editor.visitClass(target);   //インタータイプ宣言
        for (int i=0; i < methods.length; i++) {
            jc.currentMethod = methods[i];
            methods[i].instrument(editor); //アドバイス
        }
        target.writeFile();
    }
}
```

WeaverDriver は JoshExprEditor をインスタンスとして持ち、このインスタンスはアスペクトと等価な情報を持つ。JoshExprEditor は ExprEditor のサブクラスであるが、Josh の weave のための機能が追加されている。weave メソッドにクラス名を与えると、そのクラスに weave 処理をする。ここではまず、weave 対象クラスの CtClass オブジェクトと、そのクラスの CtMethod オブジェクトを入手する。次に JoshContext オブジェクトを生成する。このオブジェクトは weave の過程の情報などを参照するためのものであるが、この段階では何も情報を持っていない。JoshContext には、現在 weave 中のクラスやメソッドの情報を与える。そして visitClass と instrument メソッドを呼んで weave する。

3.4.1　具体的なコード変換処理

アスペクトをコード変換すると、ExprEditor のサブクラスのプログラムが生成される。正確には JoshExprEditor のサブクラスである。アスペクト内のアドバイスやインタータイプ宣言は、それぞれ適当な命令に変換される。例えば以下のような 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 文を含む edit メソッドを生成する。

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

call はメソッド呼び出し演算を対象としているため、オーバーロードされている edit メソッドのうち、MethodCall を受け取るものの中に処理される。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。ボディの中の$で始まる特別変数は、バイトコードに変換される際に適切な値に展開される。

within 指定子のように、joinpoint の種類ではなく個所を指定するものもあった。

```
before : within("Point") {
    /* アドバイスボディ */
}
```

例えば上記のような pointcut を使ったアドバイスがあるとする。これは以下のように変換される。within 指定子の対象は演算 joinpoint でない。その場合は全ての edit メソッド内に展開される。ここで jc は JoshContext オブジェクトであり、この例の場合は、この edit メソッドが呼ばれたときにどの CtClass を調査していたかという情報を得ている。

```
public void edit(MethodCall m) {
    if (jc.currentClass.getname().equals("Figure"))
        m.replace(アドバイスボディ);
}
        :
        :
public void edit(NewExpr n) {
    if (jc.currentClass.getname().equals("Figure"))
        n.replace(アドバイスボディ);
}
```

pointcut が 2 つの指定子の論理演算子の合成の場合は、同じ論理演算子の合成となるように変換される。例えば以下の pointcut があったとする。

```
call("void *.move(..)") && within("Figure")
```

これは以下のようなコードに変換される。

```
public void edit(MethodCall m) {
    if ( mc.getMethodName().equals("move")
      && jc.currentClass.getName().equals("Figure") )
```

call の対象はメソッド呼び出し、within の対象はクラス構造であり、Josh コンパイラは、その pointcut が対象としてる joinpoint オブジェクトを自動的に関連づける。

異なる演算 joinpoint を対象とした指定子を、論理積で組み合わせることはできない。なぜなら例えばメソッド呼び出し演算とフィールド参照演算が同時に起きることはないので、処理系がコンパイルエラーを出す。演算 joinpoint を対象とした指定子と、構造 joinpoint を対象とした指定子を論理積で組み合わせることはできる。例えば 3 種類の指定子がそれぞれメソッド呼び出し、メソッド構造、クラス構造を対象としているならば、これらを論理和で組み合わせることもできる。

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、アドバイスボディを条件評価の式でラップする。

```
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。

```
public void edit(MethodCall m) {
    if (jc.currentClass.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 の全てがアドバイスの対象となる。例えば withincode 指定子の対象 joinpoint がメソッドの場合、

```
before : withincode("void Point.init()") {
   /* アドバイスボディ */
}
```

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全ての演算 joinpoint の直前に、そのボディを挿入する。

アスペクト内のインタータイプ宣言も、ExprEditor クラス内の命令に変換される。例えば、

```
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。

```
public void visitClass(CtClass cc) {
    if (cc.getName().equals("Point"))
        cc.addField(CtField.make("int z;", cc));
    }
```

インタータイプ宣言は、演算の改変をするアドバイスとは目的が違うため、edit メソッド内では処理できない。Josh では visitClass というメソッド内でインタータイプ宣言を実現する。cc は weave の対象となっているクラスである。アドバイスと同じように、pointcut の部分が if 文の条件式になっており、cc 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。

その他に、Josh コンパイラはアスペクトと同名のクラスをつくりだす。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ない。また現在の実装上の理由により、アスペクトメンバを参照する場合は完全修飾名でアスペクト名を記述する必要がある。

1 つのアスペクトからは、計 2 つのクラスが作り出されることになる。1 つ目の ExprEditor のサブクラスは、アスペクト名に”Editor”を連結した名前になり、2 つ目の方はアスペクトと同名のソースファイルが生成される。例えば Sample.josh からは SampleEditor.java と Sample.java が生成される。SampleEditor.java は weave のときにのみ使用し、実行時には必要ない。

3.4.2　コード変換後の pointcut 指定子

Josh コンパイラは pointcut 指定子を、ExprEditor の edit メソッド内の if 文の条件式に変換する。正確には、pointcut 指定子と同じ名前のメソッド呼び出しへと変換される。call という指定子がコード変換されて実行されることは既に説明したが、正確には以下のように変換がおこなわれる。例えば pointcut のコードが、

```
before : call("void Point.set*(..)") {
    /* アドバイスボディ */
}
```

の場合、以下の if 文が edit(MethodCall mc) メソッドの中に挿入される。

```
if (call(mc, new String[]{"void Point.set*(..)"}, jc)) {
    mc.replace(アドバイスボディ)
}
```

ここで jc は JoshContext オブジェクトである。call メソッドの中身は、mc の情報と pointcut の条件である ”void Point.set*(..)” を比較する。mc と条件が一致する場合には真が返ってきてアドバイスを実行する。

同様に例えば pointcut のコードが、

```
before : set("* Point.*") {
    /* アドバイスボディ */
}
```

の場合、以下の if 文が edit(FieldAccess fa) メソッドの中に挿入される。

```
if (set(fa, new String[]{"* Point.*"}, jc)) {
    fa.replace(アドバイスボディ)
}
```

この場合も set メソッドの中では、fa と条件である ”* Point.*” を比べて真偽を決める。

このように Josh の pointcut 指定子は全て、boolean 型の返り値を持ち、引数は

[joinpoint オブジェクト，　String の配列，　JoshContext]

の並びになっている。そのためユーザが同じシグネチャのメソッドを定義すれば、それを pointcut 指定子としてアスペクトの中で自由に使うことができる。処理系から見ると、call などの標準の指定子とユーザ定義の指定子に区別はない。

within などのように、対象 joinpoint が演算ではなく構造の場合がある。その場合も同じように展開されるが、1 番目の実引数には、対象とする構造 joinpoint オブジェクトが入る。例えば、

```
before : within("Point") {
    /* アドバイスボディ */
}
```

という pointcut は以下のように変換される。

```
if (within(jc.currentClass, new String[]{"Point"}, jc)) {
    mc.replace(アドバイスボディ)
}
```

within の対象は演算 joinpoint ではないので、オーバーロードされている全ての edit メソッド内にこの命令が挿入される (ただしこの例のコードでは仮引数名が mc である)。within 指定子の対象はクラス構造なので、JoshContext オブジェクトを通して CtClass オブジェクトを渡している。within メソッドの中では、jc.currentClass と条件の”Point”を比べて真偽を決める。

第4章　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。この章では pointcut 指定子の実装方法について述べる。

4.1　新たな pointcut 指定子の定義

ここで例として simpleCall という簡単な指定子の定義方法を示す。これは、

```
simpleCall("Point", "hello");
```

のように使う。コンパイラはこれを以下のように変換する。

```
if(simpleCall(mc, new String[]{"Point", "hello"}, jc))
```

この指定子は、メソッド呼び出しを抽出するが、その際にクラス名とメソッド名のみが条件になるものである。通常の call 指定子などと違い、返り値や引数の数などは抽出の条件としていない。これは以下のように定義する。

```
static boolean simpleCall(MethodCall mc, String[] args,
                          JoshContext jc) {
    //上記の使用例の場合 args[0]="Point", args[1]="hello"である
    String calleeName = mc.getMethodName();
    String className = mc.getClassName();
    if ( className.equals(args[0]) &&
         calleeName.equals(args[1]) )
        return true;
    else
        return false;
}
```

pointcut 指定子メソッドの引数は 3 つであり、型も特定のものでなければならない。第一引数は、この指定子が抽出対象とする joinpoint オブジェクト

である。FieldAccess や CtMethod などの他の joinpoint オブジェクトでもよい。第二引数は String 型の配列であり、この指定子が joinpoint を抽出するか否かを決める条件に使われる。pointcut 指定子の使用の際には、カンマで区切って文字列を記述すればよく、処理系が自動的にそれらを配列の中に入れる。第三引数は JoshContext クラスである。このクラスのオブジェクトは weave の過程における様々な情報を保持しており、その情報を基に pointcut をすることができる。これも指定子使用の際には明記する必要なく、処理系が自動的に生成する。

4.1.1　動的指定子と JoshContext

ここで JoshContext の使用例として、動的指定子の定義方法について述べる。動的指定子とは、静的な情報だけでは weave できるかどうか判断しきれない pointcut 指定子である。例えば AspectJ の target 指定子は動的指定子であり、ターゲットオブジェクトの実行時の型によって weave するかどうかが変わる。以下の before アドバイスを getInt() メソッド内のメソッド呼び出しに weave できるかどうかは静的には決められない。なぜなら getInt() メソッド内にある intValue() メソッド呼び出しのターゲットオブジェクトが、Integer オブジェクトであるかは静的にはわからないからである。

```
before(Integer it)
    : call(int Number.intValue()) && target(it) {
        System.out.println(it);
}

int getInt(Number n) {
    return n.intValue();
}
```

AspectJ でこれを weave すると、getInt() メソッドは以下のようなコードになる。

```
int getInt(Number n) {
    if (n instanceof Integer) {
        System.out.println(n);
    }
    return n.intValue();
}
```

アドバイスボディの前に if 文が挿入されており、アドバイスを実行するか否かが実行時に決まる。

Josh でも JoshContext オブジェクトを使えば、動的指定子を定義することができる。ここで例として paramType1 という簡単な動的指定子の定義方法を示す。これは、第一引数が指定された型と一致するメソッド呼び出しを抽出する。例えば以下のように使用し、この pointcut 指定子は、第一引数が ColorPoint クラスのオブジェクトの場合にのみ、その joinpoint を抽出する。

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

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(MethodCall 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 はメソッド呼び出し joinpoint を抽出したいので、の第一引数は MethodCall オブジェクトである。第二引数の String 型の配列に、指定子使用の際の引数が入る。第三引数は JoshContext オブジェクトである。

paramType1 は指定子は動的指定子であり、実行時の第一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をラップして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。

JoshContext オブジェクトの内容をアドバイスに反映するために、実際にはアドバイスは以下の通りに edit メソッド内に挿入される。以下の例では、上部分のアドバイスが、下部分のコードに変換される。

```
before : paramType1("ColorPoint") {
    System.out.println("-- log --");
}

public void edit(MethodCall m) {
  String beforeStmt = "";
  if (paramType1(m, new String[]{"ColorPoint"}, jc)) {
    String code =
        "System.out.println(\"-- log --\");";
    beforeStmt += jc.wrapByIf(code);
  }

  beforeStmt += $_ = $proceed($$);
  m.replace(beforeStmt);
}
```

paramType1 の中で JoshContext.setIf を使って渡しておいた if 式は、ここで wrapByIf メソッドで反映される。結果として、

```
if (obj instanceof ColorPoint) {
    System.out.println("-- log --");
}
```

という命令が抽出されたメソッド呼び出しの前に挿入される。ここで obj は、抽出されたメソッド呼び出しの対象オブジェクトである。

実行時の内容をローカル変数として引き渡していく機能もある。これは expose(String) メソッドを使って実現する。指定子を実装する static メソッド内で expose メソッドを呼ぶと、渡されたにテキストは、後にアドバイスボディの先頭に挿入される。したがって変数宣言をするテキストを expose メソッドに渡すと、その変数をアドバイスボディの中で使用できる。例えば以下のコードは、変数 cname をアドバイスボディの中で使えるようにする。cname のあ値は、この joinpoint を含むクラス名である。

```
jc.expose("String cname = \"" + c.getName() + "\";");
```

ここで jc は JoshContext オブジェクト、c は CtClass オブジェクトである。このコードはコンパイルされて、アドバイスの最初の部分に挿入される。例えば c が Point クラスを表しているならばアドバイスには、

```
String cname = "Point";
```

というコードが挿入される。

他にも JoshContext には、joinpoint のフローを参照するメソッドがある。演算 joinpoint はメソッドボディ(コンストラクタボディ) に含まれるが、この joinpoint に至るまでに、どのような演算が行われてきたかという情報を JoshContext は持っている。指定子を実装するメソッドは、フローの一番先にある joinpoint しか受け取れないため、この機能は有用である。

4.1.2　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既に説明した。

```
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。

4.1.3　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。コンテクスト引き渡しとは、weave 対象の情報を基にしたボディを記述することであった。この機能は、第 2.3.2 章で述べられた Visitor パターンの関心事をモジュール化するのに役立つ。Josh ではインタータイプ宣言も pointcut 指定子で記述するので、複数のクラスが同時に weave 対象となり、その際に対象クラスごとに異なるボディ

の要素を追加したい場合がある。この機能を使えばそのような問題を解決できる。その例を示す。

```
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 を使って手にいれ、それに応じて挿入する accept メソッドを変えている。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第 2.3.2 章の 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を活用してるので、AspectJ を使った第 2.3.2 章の解決法のような実行時ペナルティがない。

4.2　複雑な pointcut 指定子

本論文の動機付けとなった 2.3.2 章の問題は、4.1.3 章で解決法を示した。この章では 2.3.1 章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 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、2.3.1 章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]);
        // root represents "FigureElement".
    CtClass callee = jc.getCtClass(mc.getClassName());
    if (!callee.subtypeOf(root))
        return false;

    String mname = args[1]; // mname == "redraw".
    CtMethod mth = mc.getMethod();
    // skip if the method is redraw().
    if (mth.getName().equals(mname))
        return false;

    Hashtable fields =
        enumerateFilds(jc, root, callee, 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。まず、この updater メソッドは呼ばれたメソッドがどのクラスに属するかを調べ、FigureElement のサブクラスでないなら偽を返して終わる。次に、mth が redraw メソッド自身かを調べ、そうである場合は偽を返す。これは再帰呼び出しを避けるためである。そして、enumerateFields メソッドを呼び、callee クラスの redraw メソッドが参照するフィールドの一覧を、ハッシュ表 fields に記録する。この際 fields をキャッシュすると効率的である。さらに、updater メソッドは instrument メソッドを呼ぶ。このメソッ

ドは先ほど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に書き込み処理をするか否かを調べる。ひと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。updated は updater メソッドが属するクラスの static フィールドである。

enumerateFields メソッドは redraw メソッドを調べ、FigureElement のサブクラスにあるフィールドの中で、redraw メソッドに読み込み処理をされているものを記録する。以下のコードが enumerateFields メソッドの定義である。root は FigureElement クラスを表す CtClass オブジェクトであり、mname は "redraw" である。また declaring は FigureElement のサブクラスであり、調べたい"redraw" メソッドが属する。

```
public Hashtable enumerateFields(JoshContext jc, CtClass root,
                        CtClass declaring, String mname) {
    CtMethod cm = declaring.getDeclaredMethod(mname);
    Hashtable fields = new Hashtable();
        //ここでキャッシュを使えれば効率的である
    cm.instrument(new ExprEditor() {
        public void edit(FieldAccess expr) {
        if (expr.isReader() &&
            expr.getCtClass().subtypeOf(root))
            fields.put(expr.getFieldName(),
                       expr.getCtClass());
        }
    });
    return fields();
}
```

4.3　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 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。例えば 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。

Javassist には、クラス、フィールド、メソッドを検査する機能と改変する機能がある。検査の機能は標準 Java リフレクション API のものと同等であ

る。改変の機能は Javassist 独自のものであり、新たなフィールドやメソッドを追加することができる。追加には、ソースコード断片を与えればよい。

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。メソッドは CtMethod オブジェクトにより表されるが、このオブジェクトを使えば、メソッドボディのはじめや終わりにコード断片を挿入することができる。メソッドボディ中で投げられる例外を捉えるために、catch 節を追加することもできる。CtMethod オブジェクトには、メソッドボディ内に含まれる joinpoint オブジェクトを得る機能もある。これらの joinpoint オブジェクトの中には、AspectJ では扱えない、instanceof 命令などもある。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いれることはできない。しかしながら、joinpoint オブジェクトは豊富な情報を持っている。例えば MethodCall オブジェクトから、その呼ばれたメソッドの名前、引数、返り値、そしてそのメソッド呼び出し命令をボディに含んでいるメソッドなどを手に入れられる。そのうえ joinpoint オブジェクトには replace メソッドがあり、これは引数として渡したソースコードと、本来の演算を置き換える。

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。例えば 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い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。例えば、AspectJ の cflow 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。

第5章　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[18] 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全ての 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 [表 5.1 の上部分]。実験の環境は、Sun Blade 1000(Dual UltraSPARC III 750MHz, 1GB メモリ) Solaris 8, Sun JDK 1.4.0_01, AspectJ 1.1b2 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、全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 [図 5.1] 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わかる。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全て

表 5.1: Xerces への weave および実行時オーバヘッドの計測

	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

図 5.1: 実験に使ったアスペクト

```
----- Xerces 用 : AspectJ のアスペクト ----------------------
----- 無限ループを防ぐために!within(AJAspect) としている ---
public aspect AJAspect {
    static int count = 0;
    before() : call(public * *.*(..)) && !within(AJAspect){
        System.out.println("public call -- woven by AspectJ :"
                           + AJAspect.count);
        AJAspect.count++;
    }
}

----- Xerces 用 : Josh のアスペクト -------------------------
aspect JSAspect {
    static int count = 0;
    before : call("public * *.*(..)")  {
        System.out.println("public call -- woven by Josh :"
                          + JSAspect.count);
        JSAspect.count++;
    }
}


************************************************************
************************************************************

----- JavaGrande Euler 用 : AspectJ のアスペクト ------------
aspect AJAspectSection3Euler {
    before(euler.JGFEulerBench tarObj) :
call(* *.*(..)) && target(tarObj) {
        euler.JGFEulerBench.mycount++;
    }
}

----- JavaGrande Euler 用 : Josh のアスペクト ---------------
aspect JoshAspectSection3Euler {
    before :
      call("* *.*(..)") && target("euler.JGFEulerBench") {
          euler.JGFEulerBench.mycount++;
    }
}
```

表 5.2: Java Grande ベンチマークによる性能比較

	Euler	Molecular	Monte Carlo	Ray Tracer	Search
コンパイル+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	2,307	10,818	42,599	5,338,398	71,228,058

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 個所と多くの個所に weave 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[9] の sequential ベンチマークである。weave したアドバイスは、全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 [図 5.1 の下部分]。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 5.2] の通りになった。Josh は weave 時間が早いが、実行時間のオーバヘッドがあることがわかった。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 個所挿入しているが、AspectJ は 73 個所しかしていない。

さらに我々は、インタータイプ宣言の性能測定をおこなった。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 2.3.2 章で述べた、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 2.3.2 章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一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。

表 5.3: インタータイプ宣言の weave 時間の計測 (秒)

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

```
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。例えば 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 5.3] 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 時間も短いことがわかった。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。例えば Josh では、Expr などを完全修飾名で記述しなければならない。

第6章　まとめ

本論文では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。本論文が提案した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、従来は不可能であったjoinpointの抽出方法の指示が可能になった。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に過ぎないといえる。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。

Joshではユーザがpointcut指定子を新たに定義することが可能であるが、利便性のため使用頻度の高い基本的な指定子はあらかじめJoshでも与えている。現段階ではAspectJのcall, get, set, initializatin, within, withincode, target, thisと同じ機能を持つpointcut指定子を提供している。AspectJのargsは、pointcutとしてjoinpointを選別するのとコンテクスト引き渡しの2つの役割を持つ。joinpoinの選別に関しての機能は特に必要ないといえる。例えばcall指定子と併用するときにはcall指定子の方で引数に関する条件を記述すればよいのからである。コンテクスト引き渡しに関しては、Joshでは明示的な宣言すること無く$で始まる特別な変数が使えるので必要がない。execution指定子は、メソッドボディの先頭や最後にアドバイスを挿入するものである。これはJoshのjoinpointモデルでは、インタータイプ宣言に分類される。なぜならこれはメソッド呼び出しなどの演算を対象にしたものではなく、メソッド自体という構造を対象としているからである。同様にstaticinitialization指定子もインタータイプ宣言に分類できる。どちらの指定子もまだ実装していないが、インタータイプ宣言で扱えるように実装するのは容易である。handler, cflow, cflowbelow, if指定子らも標準ではまだ提供していない。

joinpointを操作可能にすることにも弊害はある。もしAspectJが同様な方法をとったとしても、BCEL[5]やJMangler[13]のような低レベルのバイトコード編集ツールを使う必要がでるくらいに、難解なものになるだろう。そ

の点 Josh は、ソースコードレベルの抽象度を保ちつつ、高度な joinpoint 操作を提供する。

また本論文では Josh と AspectJ の性能比較の実験をおこない、結果を表で示した。Josh は AspectJ よりも高速に weave できるが、実行時間のオーバヘッドがあることがわかった。汎用性をあげたことによる欠点があらわれたともいえるが、性能向上のための工夫の余地はまだ残っている。

参考文献

[1] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach". In *European Conference on Object-Oriented Programming (ECOOP) 1992*, pages 372–395. Springer-Verlag, 1992.

[2] Johan Brichau, Kim Mens, and Kris De Volder. "Building Composable Aspect-Specific Languages with Logic Metaprogramming". In *Proceedings of 2nd International Conference on Generative Programming and Component Engineering*, pages 110–127. Springer-Verlag, 2002.

[3] Shigeru Chiba. "Load-Time Structural Reflection in Java". In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2000)*, pages 313–336. Springer-Verlag, 2000.

[4] Shigeru Chiba and Muga Nishizawa. "An easy-to-use toolkit for efficient Java bytecode translators". In *Proceedings of 2nd International Conference on Generative Programming and Component Engineering(GPCE2003)*, pages 364–376. Springer-Verlag New York, Inc., 2003.

[5] Markus Dahm. "Byte Code Engineering with the BCEL API". Technical Report B-17-98, Freie Universit at Berlin, Institut f ur Informatik, April 2001.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison-Wesley, 1995.

[7] Kris Gybels and Johan Brichau. "Arranging language features for more robust pattern-based crosscuts". In *Proceedings of Aspect-Oriented Software Development (AOSD2003)*, pages 60–69. ACM Press, 2003.

[8] Stefan Hanenberg and Rainer Unland. "Parametric Introductions". In *Proceedings of Aspect-Oriented Software Development (AOSD2003)*, pages 80–89. ACM Press, 2003.

[9] Java Grande at EPCC, http://www.epcc.ed.ac.u;/javagrande/index_1.html. *JavaG Benchmarking.*

[10] Gregor Kiczales. "The Fun Has Just Begun.". Keynote talk at 2nd International Conference on Aspect-Oriented Software Development(AOSD 2003).

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jffrey Palm, and William G. Griswold. "An Overview of AspectJ". In *European Conference on Object-Oriented Programming(ECOOP 2001)*, LNCS 2072, pages 327–353. Springer, 2001.

[12] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. "Aspect-Oriented Programming". In *European Conference on Object-Oriented Programming(ECOOP 1997)*, LNCS 1241, pages 220–242. Springer, 1997.

[13] G. Kniesel, P. Costanza, and M. Austermann. "JMangler – A Framework for Load-Time Transformation of Java Class Files". In *Proceedings of IEEE Workshop on Source Code Analysis and Manipulation*, 2001.

[14] Hidehiko Masuhara and Kazunori Kawauchi. "Dataflow Pointcut in Aspect-Oriented Programming". In *Proceedings of The First Asian Symposium on Programming Languages and Systems(APLAS03)*, LNCS 2895, pages 105–121, 2003.

[15] Mira Mezini and Karl Lieberherr. "Adaptive plug-and-play components for evolutionary software development". In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications(OOPSLA 1998)*, pages 97–116. ACM Press, 1998.

[16] Harold Ossher and Peri Tarr. "Hyper/J: multi-dimensional separation of concerns for Java". In *Proceedings of the 22nd International Conference on Software Engineering*, pages 734–737. ACM Press, 2000.

[17] Sun Microsystems, Inc. *Java^{TM} Core Reflection API and Specification.*, 1997.

[18] XML.APACHE.ORG, http://xml.apache.org/xerces2-j/index.html. *Xerces-2.6.0.*

[19] 河内一了 and 増原英彦. ”アスペクト指向言語におけるデータフローポイントカット”. In **日本ソフトウェア科学会学会誌:コンピュータソフトウェア**, 2003.

[20] 千葉 滋. ”自己反映言語 Open C++とその分散処理への適用の実際”. In **コンピュータソフトウェア**, Vol.11, No.3, pages 33–48, 1994.

[21] 千葉 滋 and 立堀 道昭. ”Java バイトコード変換による構造リフレクションの実現”. In **情報処理学会 論文誌**, 42 巻 11 号, pages 2752–2760, 2001.

[22] 渡部 卓雄. ”チュートリアル リフレクション”. In **コンピュータソフトウェア**, Vol.11, No.3, pages 5–14, 1994.

付 録A　Joshの使用方法

Josh のコマンドラインでの実行方法を示す。Josh では weave 対象となる通常の Java クラスと.josh アスペクトを、静的またはロード時のどちらのタイミングでも weave 可能である。静的な weave の利点は、weave 処理を全て終えた状態でアプリケーションの実行を始められるところである。一方ロード時の weave の利点は、必要とされたクラスにのみ weave をおこなうので無駄が無いところである。しかしながら欠点として、weave エラーの発見がロード時になってしまう。

A.1　静的 weave

実行概要

josh.StaticWeaver [オプション] [josh files] [class files]

引数は混在してよい

説明 Josh 言語の josh ファイルとクラスファイルを静的に weave して出力する。引数の中で拡張子が.josh で終わっているものは自動的にアスペクトだと判別される。引数のクラスファイル名には拡張子は不要である。また引数にディレクトリがあると、そのディレクトリ直下にある全てのクラスファイルが weave の対象となる。サブディレクトリは対象としない。

オプション

-dir=*DIRECTORY*

weave 済みクラスの出力先を指定する。初期設定ではクラスファイルと同じ場所であり、自動的に上書きする。

-compiler=*COMPILER*

アスペクトは Java のソースコードに変換された後に、バイトコードにコンパイルされるが、その際どのコンパイラを使うかを指定する。例えば -compiler=jikes のようにして使う。ただし、ここで指定したコンパイラをプログラム内から外部プロセスとして呼び出すだけなので、パスやその他の環境を調整しておく必要がある。

初期設定では、com.sun.tools.javac.Main クラスの compile メソッドを呼ぶ。

-aspectlist=*LIST*
アスペクトファイルをリストファイルで指定する。

-notranslate
アスペクトのコンパイルをしないで weave だけするように指定する。通常はアスペクトファイルの変更の有無に関わらず、毎回コード変換とコンパイルをしてから weave をおこなう。アスペクトを変更していない場合にこの手間を省くことができる。

A.2　ロード時 weave

実行概要

josh.LoadtimeWeaver [AppMain クラス] [オプション] [実行時引数]

オプションと実行時引数は混在してもよい

説明

ロードされるクラスファイルに順々に weave しながら、アプリケーションを実行する。AppMain クラスの main メソッドが実行される。オプションが無い場合は、直接 AppMain クラスを実行するのと同じである。

オプション

-aspect=*ASPECT*
アスペクトファイルを拡張子 (.josh) 付きで指定する。

-aspectlist=*LIST*
アスペクトファイルをリストファイルで指定する。LIST に一行ごとにアスペクトファイルを記述しておくと、それら全てが weave に使われる。

-notranslate
アスペクトのコンパイル処理を省く。