東京大学
情報理工学系研究科 創造情報学専攻
修士論文

適用するメソッドを切り替えることで破壊的クラス拡張で生じるメソッド衝突を避けられるモジュール機構 Method Shells

Method Shells: a module system for avoiding method conflicts on destructive class extensions by switching applied methods

竹下 若菜
Wakana Takeshita

指導教員　千葉 滋 教授

2014 年 1 月

概要

本研究では，有効にするメソッドを切り替えることで破壊的クラス拡張で生じるメソッド衝突を避けられるモジュール機構 Method Shells を提案する．AspectJ, Ruby, GluonJ など多くの言語は既存のコードを書き換えずに，差分のみを別部分に記述することで，既存のコードに変更を加える事ができるメカニズムを持つ．我々はこのメカニズムを破壊的クラス拡張と呼んでいる．破壊的クラス拡張を用いるとコードの再利用性を高めることができるが，メソッドの衝突を引き起こす危険性がある．メソッドの衝突とは，同じメソッドを複数の破壊的クラス拡張が再定義してしまい，メソッド呼び出しの候補が複数存在してしまうことを指す．メソッドの衝突を解消するために，本研究では Method Shells というモジュール機構を提案する．MethodShells は，一度に有効にするメソッドをグループ化し，そのグループを実行時に切り替えることで，有効にするメソッドを状況に応じて切り替えることができる．本研究ではこのモジュール機構のセマンティクスを定義した．また，セマンティクスをそのままに実装してしまうと，モジュールの増加に応じて実行時間が増加する場合があるため，分割コンパイルをあきらめることでメソッド呼び出しの最適化をする実装方法も提案した．本研究ではその提案した実装方法とセマンティクスそのままの実装との比較実験も行っている．

Abstract

We propose Method Shells, which is a module system for avoiding method conflicts caused by destructive class extensions. A number of programming languages have a mechanism for customizing code without rewriting original source code but by only describing differences in a separate file, for example, aspect of AspectJ, Ruby's openclass, reviser of GluonJ. We call this mechanism destructive class extensions. By using destructice class extensions, code reusability is improved. However, different customizations often modify the same method in the heavy use of destructive class extensions. We call this problem method conflicts. To address this problem, Method Shells provide a mechanism for avoiding method conflicts by switching applied methods at runtime. With this system, programmers can group methods which is applied together and switch those groups at runtime. We present the formal semantics of this system and its optimized implementation. The runtime speed increases as the number of modules increases with the naive implementation. We also show the comparison between the runtime speed of the optimized and naive implementations.

目次

第 1 章

はじめに

現在，アプリケーションを 1 から全てそのプログラマが作るというのは珍しくなっている．多くの場合，既存のライブラリやフレームワークを集めてそのコードを再利用し，アプリケーション独自の機能のみを実装することで開発する．今日では、多くのアプリケーションやソフトウェアは既存のライブラリやフレームワークを再利用して作られており、そのライブラリやフレームワークもまた別のライブラリやフレームワークを再利用して作られている。このようなコードの再利用は開発の効率を上げ，生産性をあげることにつながる．

また，既存のコードをただ使いまわすのではなく，既存コード中の一部の機能を変更することで，より多くのコードを再利用できる．少し機能が違う，あるいは少し機能を追加したい，そのような場合にもコードを再利用できるため，生産性の向上が期待できる．

このような機能の変更を行う際，最も単純に思いつく方法はコードを直接書き換えることであるが、既存のコードを書き換えずに差分のみを別ファイルに記述することで機能を変更できた方が便利なことが多い．変更部分のみを別ファイルにモジュール化できるため、変更にミスがあったとしてもその別ファイルを消去するだけで元のコードを取得できる．また、別のプログラマが差分のみを再利用することも可能になる。

差分のみを別ファイルに記述するだけで機能変更を行えるメカニズムはすでに数多く存在している．有名なのは，Aspect-oriented Programming(AOP)[1] である．AOP を用いると，別ファイルに差分のみを記述するだけで，既存のコードに新たなコードを挿入することができ，プログラムの振る舞いをコードを書き換えることなく変更できる．他にも，Ruby[2]，GluonJ[3]，AHEAD[4]，MultiJava[5]，Jiazzi[6] など様々な言語が，既存のコードを書き換えずに既存のコードにメソッドの追加/再定義を行う機能を導入している．我々はこのような機能を破壊的クラス拡張と呼んでいる．破壊的クラス拡張を用いると，差分のみの記述でコードに変更を加える事ができ，さらに変更部分をモジュール化できるため，コードの再利用を促進できる．

しかし，破壊的クラス拡張は利点を持つ反面，メソッドの衝突と呼ばれる問題を引き起こす危険性がある．メソッドの衝突とは，同じメソッドを複数の破壊的クラス拡張が変更してしまい，同名のメソッドの複数の定義がプログラム中に存在してしまうことを指す．破壊的クラス拡張を導入している言語の多くは，メソッドの衝突が起きると，エラーを出す，あるいは有効なメソッドを一意に決定して実行を続ける，などの処置をとる．しかし，それではプログラマ

が意図せぬ動作をプログラムがする可能性がある．特にライブラリやフレームワークを再利用しているときにこの可能性は顕著に現れる．ライブラリやフレームワークを再利用する際，多くのプログラマはそれらのコードをブラックボックスとして扱い，どのような実装がなされているかに目を通さない．再利用しているライブラリやフレームワークが破壊的クラス拡張を用いていた場合，それらが作成中のアプリケーションの破壊的クラス拡張と衝突してしまう，あるいは，再利用している別のライブラリの破壊的クラス拡張と衝突してしまうといった問題が生じる可能性がある．別のライブラリをインポートしただけなのに，意図せぬプログラムの動作を引き起こしてしまうのである．エラーを出す場合にはもちろんのこと，有効なメソッドを言語処理系が勝手に一意に決定してしまう場合には，全てのメソッド衝突に対してプログラマの意図を的確に実現するような処理を行うとは限らないため，プログラマの意図せぬ動作をプログラムがする可能性が出てきてしまう．

本研究では破壊的クラス拡張によるメソッドの衝突を回避できるモジュール機構 Method Shells を提案する．Method Shells は有効にする破壊的クラス拡張を実行時に切り替えることができる．有効な破壊的クラス拡張の切り替えにより，同名のメソッドが複数あった場合にも状況に応じてそれらを使い分けることができ，メソッドの衝突を回避できる．破壊的クラス拡張の切り替えはグループごとに行われる．まとめて有効にしたい破壊的クラス拡張をグループ化し，そのグループをプログラマの事前の指定に応じて実行時に切り替える．なお，同一グループ内ではメソッドの衝突を許さない．

本論文では使用例とともに Method Shells を説明し，そのセマンティクスを記す．また，本研究では Method Shells のメソッド呼び出しを最適化した実装方法の提案およびそのプロトタイプの実装も行った．セマンティクスそのままに Method Shells を実装すると，モジュールの増加に応じてメソッド探索の時間が増加する場合がある．本論文で提案する実装方法を用いれば，メソッド探索の時間の増加を抑えることができる．既存の Java コンパイラを拡張し，Method Shells 独自の機能を組み込むことでこのプロトタイプの実装を実現した．さらに本研究では，実際にセマンティクスに沿った実装方法と提案した実装方法との実行時間の比較を行い，メソッド探索にかかる時間の増加を抑えられることを示した．

以降，2 章では破壊的クラス拡張とその問題点および既存の解決手法を議論し，3 章では本研究の提案するモジュール機構 Method Shells について述べる．4 章では，Method Shells のメソッド呼び出しの最適化を行った実装方法を提案し，5 章では実装について述べる．6 章では，4 章で提案した実装方法の効果を測定した結果を提示し，7 章では本論文についてのまとめを述べる．

第 2 章

破壊的クラス拡張とメソッドの衝突

この章では，既存のコードを書き換えることなく既存のコードにメソッドの追加や再定義をできる破壊的クラス拡張というメカニズムを紹介し，その問題点を例を交えながら述べる。その後，破壊的クラス拡張を導入している既存の言語および問題点の既存の解決手法について述べる。

2.1 破壊的クラス拡張

ライブラリをカスタマイズするときに既存のコードを直接書き換えずに，差分のみを別ファイルに記述することで，既存のコードに変更を加えられると便利なことが多い．変更部分のみをモジュール化することができ，またそのモジュール化により保守管理もしやすくなる．変更にバグが存在している場合にも，変更部分のモジュールを削除するだけで元のコードを取得できる．このような変更を可能にする機能は多数存在し，多くの言語が導入している．AspectJ のアスペクト [7]，Ruby[2] のオープンクラス，GluonJ の reviser[3] などがその機能の例である．これらの機能を用いると，既存のコードを直接書き換えることなく，差分のみを別ファイルに記述するだけで，新しいメソッドの追加や既存のメソッドの再定義などを行うことができる．我々はこのような機能を**破壊的クラス拡張**と呼んでいる．破壊的クラス拡張は前述した AspectJ，Ruby，GluonJ の他にも AHEAD[4]，MultiJava[5, 8]，Jiazzi[6] など多くの言語に導入されている。

今，HTML 文書をレンダリングする HTML-renderer ライブラリがすでに存在し，それを利用してアプリケーション組み込みのウェブブラウザを構築するための browser ライブラリを作りたいと仮定する．組み込みウェブブラウザではポップアップウィンドウを表示するべきでないため，HTML-renderer ライブラリ中の WebPage クラスに対して，ポップアップウィンドウ表示をブロックするというカスタマイズを行いたい．また，さらにこの WebPage クラスに対して browser ライブラリ特有のいくつかの機能の追加を行いたいとする．破壊的クラス拡張を利用することで，このような HTML-renderer ライブラリの WebPage クラスへの機能の変更および追加をし，この browser ライブラリの実装を実現しようと考えてみる。

図 2.1 が browser ライブラリ中の破壊的クラス拡張である．この例では，破壊的クラス拡

```
1  //in the browser library
2  revise WebPage{
3    void setPopupItem(Popup p, HTML text){
4      alert(‘‘disabled’’);
5    }
6    void onClick(Mouse m){
7      URL url = m.getURL();
8      popup(url);
9    }
10 }
```

図 2.1. reviser の例．2 行目から 10 行目の revise 節で指定されたブロックが reviser である．この例では，WebPage クラスに setPopupItem メソッドと onClick メソッドを再定義/追加している．

張として GluonJ の reviser を用いている．図 2.1 の 2 行目から 10 行目の revise というキーワードで指定されたブロックが reviser を表している．図 2.1 の reviser は WebPage クラスの setPopupItem メソッドを再定義し，onClick メソッドの追加を行っている．図 2.2 が図 2.1 の reviser で指定されている WebPage クラスである。WebPage クラスは HTML-renderer ライブラリで定義されている．図 2.1，2.2 のようなソースコードを記述し，コンパイラにこの 2 つのソースコードを与えることで，図 2.2 の WebPage クラスで宣言されたオリジナルの setPopupItem メソッドは図 2.1 中の reviser で宣言された setPopupItem メソッドに置き換わり，さらに図 2.1 中の onClick メソッドはオリジナルの WebPage クラスには存在しないため追加される．setPopupItem メソッドの変更によってポップアップウィンドウを表示させようとすると，常にアラートメッセージを出すようになる．

破壊的クラス拡張を用いると，このように差分のみを別ファイルに記述するだけで元のソースコードに変更を加えることができる．どの破壊的クラス拡張を有効にするかはコンパイラに指定する必要がある．サブクラスでのメソッドの上書きと異なり破壊的クラス拡張では，変更先のクラスで宣言された元のメソッドの定義が，変更後のものに置き換わる．変更先のクラスのインスタンスにおいても変更後のメソッド定義が有効になる．

同じクラスを別の破壊的クラス拡張で再定義することで，別のライブラリを作ることもできる．HTML で書かれたファイルのビューアを提供する viewer ライブラリを HTML-renderer ライブラリを拡張して作るとする．このビューアは部外秘のファイルを与えられると，赤太字の注意メッセージをポップアップで出すようにしたい．そのために viewer ライブラリでは，browser ライブラリと同様，HTML-renderer ライブラリの WebPage クラス中の setPoupItem メソッドに変更を加える．図 2.3 が viewer ライブラリのコードである．図 2.3 では，reviser を用いて WebPage クラスの setPopupItem メソッドに変更を加え，テキストを赤太字で表示するようにしている．viewer ライブラリと HTML-renderer ライブラリのソースコードをコンパイラに与えることで，setPopupItem メソッドは viewer ライブラリの破壊的クラス拡張で定

```
1  //in the HTML-renderer library
2  class WebPage{
3    void setPopupItem(Popup p, HTML text){
4      //set text
5    }
6    void popup(HTML text){
7      Popup p = new Popup();
8      setPopupText(p, text);
9    }
10 }
```

図 2.2. WebPage クラス．HTML-renderer ライブラリで宣言されており，setPopupItem メソッドと popup メソッドを含む．popup メソッド中では，setPopupItem メソッドを呼び出している．

```
1  //in the viewer library
2  revise WebPage{
3    void setPopupItem(Popup p, HTML text){
4      //show red and bold text
5    }
6  }
7  class Viewer{
8    void check(file f){
9      if(isConfidential(f))
10       new WebPage.popup(‘‘confidential’’);
11   }
12 }
```

図 2.3. viewer ライブラリ．WebPage クラスの reviser と Viewer クラスの宣言を含んでいる．WebPage クラスの setPopupItem メソッドを再定義している．

義したものに置き換わり，ポップアップウィンドウはテキストを赤太字で表示するようになる．このように，破壊的クラス拡張を用いると，差分のみを別ファイルに定義するだけで望み通りの変更を既存のプログラムに加えることができる．

2.2 メソッドの衝突

破壊的クラス拡張はコードの再利用性を高める反面，メソッドの衝突を引き起こす危険性がある．メソッドの衝突とは，複数の破壊的クラス拡張が同じメソッドを再定義してしまい，プログラム中に同じメソッドの定義が複数存在することを指す．メソッドの衝突が起きると，プログラマの意図せぬプログラムの動作を引き起こす可能性がある．

前節で用いた browser ライブラリと viewer ライブラリを同時に使うことを考えてみる．

```
1  //in the application library
2  class App{
3    void view(WebPage w){
4      new Viewer().check(new File(``test.txt''));
5    }
6    public static void main(String[] args){
7      WebPage w = new WebPage()
8      new App().view(w); // want to viewing red and bold text
9      w.onClick(); // want to prevent popup window
10   }
11 }
```

図 2.4. browser と viewer の両方を使いたい．しかし，setPopupItem メソッドが browser ライブラリと viewer ライブラリの両方で再定義されているため，メソッドが衝突している．そのため，プログラマの意図通りにプログラムが動かない可能性がある．

browser ライブラリと viewer ライブラリを独立に扱っている場合にはいいが，一緒に使うとメソッドの衝突が起きる危険がある．コンパイラに与えるソースコードとして HTML-renderer ライブラリと browser ライブラリ，viewer ライブラリを選択してしまうと，WebPage クラス中の setPopupItem メソッドの再定義が複数存在してしまい，メソッドが衝突してしまう．メソッドが衝突してしまうと，本来ならば browser ライブラリの setPopupItem メソッドを有効にしたいところで，viewer ライブラリの setPopupItem メソッドが有効になり呼び出されてしまう，あるいは逆に viewer ライブラリの setPopupItem メソッドを有効にしたい欲しいところで，browser ライブラリのメソッドが有効になり呼び出されてしまう，といった可能性がある．例えば，browser ライブラリ中から HTML-renderer ライブラリの popup メソッドを呼んだ場合，その中での setPopupItem メソッドの呼び出しでは，browser ライブラリで再定義した setPopupItem メソッドを有効にしたいと考えるのが自然である．逆も同様に，viewer ライブラリ中から HTML-renderer ライブラリの popup メソッドを呼んだ場合には，viewer ライブラリで再定義した setPopupItem メソッドを有効にしたい．しかし，メソッド衝突が起きている状態では，どのメソッドが有効になるかを言語処理系が正しく判断できるとは限らず，プログラマの思った通りの動作をしないおそれがある．

2.3　関連研究および既存の解決手法

本節では，破壊的クラス拡張の機能を導入している言語，および破壊的クラス拡張によって生じるメソッドの衝突を避ける事ができる既存の解決手法について紹介する．破壊的クラス拡張は多数の言語に導入されており，メソッドの衝突も数多く問題として取り上げられている．

これ以降，まず，破壊的クラス拡張の機能を導入している言語，そしてすでにメソッド衝突が問題とされている言語として，アスペクト指向プログラミング言語を紹介する．また，それに続いて，破壊的クラス拡張によって生じるメソッド衝突を避ける事ができる既存の解決手法

として，次の 2 種類の解決方法を用いた既存研究を紹介する．

- 破壊的クラス拡張のスコープを制限する方法
  - Virtual class
  - NewSpeak
  - Classboxes
  - Ruby の refinements
  - Worlds
- 有効な破壊的クラス拡張を実行時に切り替える方法
  - JPred
  - Us
  - Context-oriented Programming

1 つ目の「破壊的クラス拡張のスコープを制限する方法」を用いると，メソッドの衝突を避け有効なメソッドを一意に定めることができるため，ある種のメソッド衝突を解決できる．しかし，状況に応じて有効なメソッドを切り替えることができないため，プログラマの意図を実現できない場合がある．例えば，図 2.4 のような意図は実現することができない．

2 つ目の「有効な破壊的クラス拡張を実行時に切り替える方法」は，本研究の提案手法が取っているメソッド衝突回避の方法ととても良く似ている．本研究の提案手法と「有効な破壊的クラス拡張を実行時に切り替える方法」を用いている既存研究でできることはほぼ同じである．違いを述べるならば，本研究の方法の方がより宣言的である．「有効な破壊的クラス拡張を実行時に切り替える方法」を用いている既存研究では，有効な破壊的クラス拡張を切り替えるたびに一々どの破壊的クラス拡張に切り替えるかを指定しなければならない．そのため，プログラマが現在どのような破壊的クラス拡張がプログラム内にあり，どの破壊的クラス拡張が有効にされているのかといったプログラム全体に関する情報を把握する必要がある．一方，本研究では切り替え先の破壊的クラス拡張を一度宣言しておけば，一々指定しなくとも自動で切り替えが行われる．このような差異が生じているのは，本研究と既存研究との目的が異なっているためである．本研究ではモジュールを組み合わせることを目的としているのに対し，既存研究では実行時に処理を切り替えることを目的としている．

また，本節の最後では本研究の前身研究である Method Shelters を紹介する．

2.3.1 アスペクト指向言語

Aspect-oriented programming(AOP)[1] はアスペクトと呼ばれるモジュールの中に差分を記述することで，既存のコードを書き換えずに処理を変更できるプログラミング技法である．アスペクト中に記述された差分は，コンパイル時または実行時に適用され，既存のコードに新しい処理を追加あるいは既存のコードを再定義する．既存のコードを書き換えることなく，メソッドの追加や再定義ができるため，破壊的クラス拡張の一種であると考えられる．

AOP を導入している言語はすでにいくつか存在し，その 1 つが AspectJ[7] である．AspectJ

は AOP をサポートするように Java 言語を拡張したプログラミング言語である．AspectJ では，アスペクトをコンパイル時に適用する．このアスペクトをプログラムに適用する処理を，AOP では「**織り込み**」と呼ぶ．アスペクトに記述された差分はコンパイル時に既存のコードに織り込まれ，その織り込みにより処理の追加あるいは再定義が達成される．AspectJ のアスペクトは次の 2 つの要素で構成される．

- **アドバイス**：メソッドに相当するもので織り込まれる処理のこと
- **ポイントカット**：アドバイスをプログラムのどこに織り込むかを記述したもの

アドバイスとポイントカットからなるアスペクトをコンパイル時にプログラムに織り込むことで，既存のプログラムを直接書き換えることなくメソッドの追加および再定義ができる．また，変更部分のみをアスペクト中にまとめることで，変更をモジュール化することもできる．

AOP には，2.2 章で説明した破壊的クラス拡張のメソッド衝突と似たような問題が存在する．この問題は，AOP 界隈では *aspect interference* という名で問題とされてきた [9, 10]．*aspect interference* は，複数のアスペクトが一度に有効にされたときにプログラマの意図せぬ相互作用が起きることを指す．意図せぬ相互作用は様々な形で存在する．複数のアスペクトが同じポイントカットにアドバイスを織り込もうとしたり，あるアスペクトが既存のメソッドにアドバイスを織り込んだがためにそのメソッドを呼び出している他のアスペクトの挙動が変わってしまったり，などが例としてあげられる．

Douence らは aspect interference を解決するようなコンポーネントを提案している [11, 12]．[11] では，Douence らは AOP をサポートするための形式化されたフレームワークを定義している．また，アスペクト同士が依存していない状態とはどのような状態かを形式化し，その状態かどうかを静的に解析する方法を提案している．依存しているかいないかを解析することにより，衝突が起きているアスペクトを検出することができる．さらに，そうして検出したアスペクト間の衝突を解決できるようなコンポーネントも提案している．コンポーネントは次の 2 種が提案されている．

- 衝突している 2 つのアスペクトを，衝突の起きない 1 つのアスペクトへと変換する
- アスペクトのスコープを制限することで，プログラム全体にアスペクトの影響が及ぶことを避ける

[12] では，[11] で定義したフレームワークに対して 3 つの拡張を行い，それぞれについて議論を行っている．拡張により，[11] のフレームワークと比べてより高い表現力や正確な衝突の解析といった利点を得ている．しかし，いずれの拡張でもアスペクトの衝突の有無を静的に解析する能力は失わずに保持されている．

[11] および [12] のコンポーネントを用いると，アスペクトの衝突を検出することができ，その衝突を避けるようにプログラムを変換することができるため，より安全にアスペクトを用いることができる．

Takeyama らの研究で提案されている *resolver*[13] もまた aspect interference を解決できるようなシステムである．Takeyama らは，aspect interference で起きる問題のうち，特にアス

ペクト中のアドバイスで起きる衝突 (*advice interference*) の解決に主眼をおいている．advice interference は，アスペクトが織り込まれるときに同じジョインポイントに複数のアドバイスが織り込まれてしまい，プログラマの意図せぬ動作をプログラムがしてしまうことを指す．resolver はアドバイスの一種であり，指定したアドバイス間で衝突が起きているときのみジョインポイントで呼び出される．衝突しているアドバイスは resolver で定義された処理で上書きされる．そのため，resolver 中に衝突時の処理を適切に記述しておけば，特定のアドバイス間の衝突を回避できる．Takeyama らは [13] 中で revolver を導入している言語 *Airia* も提案している．

Open Modules[14] も aspect interference を避けるのに使うことができる．Open Modules はアドバイスやポイントカットをモジュール化でき，かつ，他のモジュールからの再定義を避ける事ができるモジュールシステムである．Open Modules では，モジュール中に記述されたアドバイスやポイントカットのうち，外にエクスポートするものを選択することができる．エクスポートされたポイントカットやアドバイスには，他のモジュールからアドバイスを加えることができる．しかし，エクスポートされていないメソッドを他のモジュールから呼び出したり，アドバイスを加えたりすることはできない．加えて，エクスポートされていないメソッド中でのメソッド呼び出しは，他のモジュールのアドバイスの影響を受けない．例えば，あるモジュールがメソッド m_eと m_h を持っており，そのうちメソッド m_e がエクスポートされており，メソッド m_h がエクスポートされていなかったとしよう．このとき，メソッド m_h はそのメソッドボディでメソッド m_e を呼び出していたとする．他のモジュールがメソッド m_e に新しいアドバイス a を加えたとしても，メソッド m_h からメソッド m_e を呼び出したとき a の影響は受けない．このように，エクスポートされていないメソッドの実装は他のモジュールからは隠されている．Open Modules を使うと上記のように，アドバイスのスコープを制限することができ，かつ予期せぬ再定義による意図しないプログラムの動きを避ける事ができる．

Ongkingco らは Open Modules を AspectJ に導入した [15]．Ongkingco らは，open modules を拡張することで AspectJ の全てのプリミティブなポイントカットに対応できるようにし，実装を行っている．また既存の AspectJ のプロジェクトとどのように適合するかを示し，Open Modules の使用例をあげている．

2.3.2 Virtual class

Virtual class[18, 19] はベースとなるクラスの中で宣言することができるクラスである．これ以降，ベースとなるクラスのことをエンクロージングクラスと呼ぶ．Virtual class はエンクロージングクラスのサブクラスで拡張することができ，フィールドやメソッドを追加/再定義できる．拡張する際には，通常のサブクラスで定義をオーバライドするのと同じように，追加/再定義したいフィールドとメソッドのみをサブクラス中に記述すればよい．つまり，差分のみの記述だけでクラス定義を変更することができる．そのため，virtual class と破壊的クラス拡張はとてもよく似ている．virtual class を宣言しているエンクロージングクラスとそのサブクラスの中でのみ，その virtual class とその拡張は有効になるため，複数のエンクロージングク

ラスとそのサブクラスを使うことで複数の拡張を使い分けることができる．しかし，どのメソッドが有効になるかはクラスやインスタンスによって一意に定まり，切り替えられない．そのため，同じクラスに対する複数の破壊的クラス拡張を状況に応じて切り替えながら使うことはできない．

2.3.3　NewSpeak

NewSpeak[20, 21] は virtual class の考え方を導入したプログラミング言語である．NewSpeak は，次の 2 つのアイデアを元に作られている．

- 全ての名前は遅延束縛される
- グローバルな namespace は存在しない

NewSpeak では，1 つ目のアイデアに基づき，メソッド呼び出しやインスタンス生成が全て遅延束縛される．また，グローバルな namespace は存在しないため，nested class を宣言する必要がある．nested class は一種のモジュールであり，中にクラスを含むことができる．言い換えれば，NewSpeak では全てのクラスは nested class をエンクロージングクラスとした virtual class である．同名のメソッドをサブクラス中に定義することで既存のメソッドを再定義することができ，メソッドの追加も行える．Virtual class と同様に複数のクラスとそのサブクラスを使うことで複数の拡張を使い分けることができるが，一度インスタンス化してしまうとどのメソッドが有効になるかは一意に定まり，切り替えることができない．

2.3.4　Classboxes

Classboxes[22] は破壊的クラス拡張を導入した言語であり，破壊的クラス拡張のスコープを制限することでメソッドの衝突を回避できるモジュール機構である．Classboxes のモジュールを *classbox* という．classbox はクラス宣言と破壊的クラス拡張を含む．classbox の中で宣言されたクラス宣言と破壊的クラス拡張は，その classbox の中でのみ有効となる．また，他の classbox から classbox 中にクラスをインポートすることができる．インポートしたクラスはインポートしている classbox の中で有効にすることができる．また，インポートしたクラスをさらに再定義することもできる．

図 2.5 と図 2.6 が Classbox/J[23] のコード例である．Classbox/J は classbox を Java-like な静的型付き言語に実装した言語である．図 2.5 では，rendererCB という名前の classbox を宣言している．rendererCB は WebPage クラスを含んでおり，WebPage クラス中では setPopupItem メソッドと popup メソッドが定義されている．クラス定義や破壊的クラス拡張は宣言された classbox の中でのみ有効なので，WebPage クラス宣言は rendererCB の中でのみ有効となる．図 2.6 では，browserCB という名前の classbox を宣言している．browserCB は rendererCB の WebPage クラスをインポートしている．インポートしたクラスはインポートしている classbox の中で有効にすることができ，さらに再定義をすることもできる．実際，

browserCB はインポートした rendererCB の WebPage クラスを refine 節を用いて再定義している．classbox では，破壊的クラス拡張は refine というキーワードを使って宣言される．図 2.6 の 3~11 行目が refine 節による破壊的クラス拡張である．refine 節で再定義するクラス名を指定し，ブロックの中に新しいメソッド定義を記述することでインポートしたクラスにメソッドの追加および再定義をしている．browserCB 中ではインポートしている WebPage クラスと，自身が含んでいるクラス宣言および破壊的クラス拡張が有効になる．

このように classbox では，破壊的クラス拡張のスコープを classbox 内，そしてインポートしている classbox 内に制限することができる．しかし，有効にする破壊的クラス拡張を状況に応じて切り替えることができないため，プログラマの意図するプログラムを実現できない場合がある．

例えば，図 2.7 のような viewerCB という名前の classbox があったとする．viewerCB は rendererCB の WebPage クラスをインポートして，さらに再定義している．このとき，あるプログラマが browserCB と viewerCB 両方の機能を使うようなアプリケーションを作ろうとしたとする．このプログラマは図 2.8 のような appCB という名前の classbox を作り，browserCB の WebPage クラスと viewerCB の Viewer クラスをインポートした．このとき，アプリケーション中で Viewer クラスの check メソッドを呼ぶと，ポップアップはブロックされてしまう．viewerCB で実装したときは，check メソッドを呼んだときには，赤太字のポップアップを出すように作られていたはずである．appCB 中で有効なクラス宣言および破壊的クラス拡張は，appCB 中で宣言されたクラス宣言および破壊的クラス拡張と，インポートしている browserCB の WebPage クラスと，viewerCB の Viewer クラスである．このように appCB 中にインポートされている WebPage クラスが browserCB で定義されたものであるがために，Viewer クラスは定義したときと挙動が変わり，ポップアップをブロックするようになってしまった．もし，プログラマが Viewer クラスを viewerCB で定義したときの定義のまま使いたいと考えた場合，これでは実現することができない．

このように，破壊的クラス拡張のスコープを制限するだけではプログラマの意図を実現できない場合がある．classbox では，同名のメソッドがプログラム中に複数存在することを許さないため，切り替えて使用することができないからである．有効にする破壊的クラス拡張を切り替えられるようにする機能が必要である．

2.3.5 Ruby の refinement

プログラミング言語 Ruby[2] は open class[24] という名前の破壊的クラス拡張の機能を備えている．open class を用いると，差分のみを記述するだけで簡単にメソッドを再定義することができる．Ruby の open class では破壊的クラス拡張のために，特別なキーワードを用意していない．通常のクラス宣言およびメソッド定義と同じように書くだけで，メソッドの追加および再定義ができる (図 2.9)．そのため，破壊的クラス拡張を特別な学習なしに簡単に扱うことができる．しかし，open class のスコープはグローバルであるため，再定義の影響がプログラム全体に及んでしまい，意図せぬ再定義をしてしまう場合がある．

```
1  package rendererCB;
2  class WebPage{
3    void setPopupItem(Popup p, HTML text){
4      //set text
5    }
6    void popup(HTML text){
7      Popup p = new Popup();
8      setPopupText(p, text);
9    }
10 }
```

図 2.5. rendererCB という名前の classbox．WebPage クラスが定義されている．WebPage クラスは setPopupItem メソッドと popup メソッドを含んでおり，popup メソッド内では setPopupItem メソッドが呼び出されている．

```
1  package browserCB;
2  import rendererCB.WebPage;
3  refine WebPage{
4    void setPopupItem(Popup p, HTML text){
5      alert(‘‘disabled’’);
6    }
7    void onClick(Mouse m){
8      URL url = m.getURL();
9      popup(url);
10   }
11 }
```

図 2.6. browserCB という名前の classbox．rendererCB の WebPage クラスをインポートしている．classbox の破壊的クラス拡張は refine というキーワードで宣言される．この図では，setPopupItem メソッドを再定義しており，ポップアップウィンドウをブロックしている．

意図せぬ再定義の例を 2 つあげる．1 つ目の例は，ライブラリをインポートした場合である．あるライブラリ l を現在実装中のプログラム p でインポートしたとする．ライブラリ l で既存のクラス C のメソッド m が再定義されているが，プログラマはライブラリ l をブラックボックスとして扱っており，ライブラリ l の実装を知らないため，再定義されていることが分からない．このとき，プログラム p 内でこのメソッド m を呼び出すと，ライブラリ l 内で再定義されたメソッド m が呼び出されてしまう．プログラマはただライブラリ l をインポートしただけなのに，プログラムの挙動が変わってしまう．このように，ライブラリ内で既存のメソッドを再定義していると知らずに，ライブラリをインポートしてしまい，プログラムに影響をおよぼすことがある．2 つ目の例は，ライブラリのコードを自身で再定義した場合である．プログラマ A はライブラリ l_1 と l_2 を使ってあるアプリケーションを開発している．A がライ

```
1  package viewerCB;
2  import rendererCB.WebPage;
3  refine WebPage{
4    void setPopupItem(Popup p, HTML text){
5      // show red and bold text
6    }
7  }
8  class Viewer{
9    void check(file f){
10     if(isConfidential(f))
11       new WebPage.popup(‘‘confidential’’);
12   }
13 }
```

図 2.7. viewerCB という名前の classbox．rendererCB の WebPage クラスをインポートしており，キーワード refine を用いてインポートした WebPage クラスの setPopupItem メソッドを再定義している．

```
1  package appCB;
2  import browserCB.WebPage;
3  import viewerCB.Viewer;
4  class App{
5    void view(WebPage w){
6      new Viewer().check(new File(‘‘test.txt’’));
7    }
8    public static void main(String[] args){
9      WebPage w = new WebPage();
10     new App().view(w); // preventing popup window!!
11     w.onClick();
12   }
13 }
```

図 2.8. appCB という名前の classbox．browserCB の WebPage クラスと viewerCB の Viewer クラスをインポートしている．classbox 中でインポートしているクラスが有効になるため，Viewer クラス中で WebPage クラスのメソッドを呼び出したときには，browserCB で定義したメソッドが呼び出される．

ブラリ l_1 のメソッドを作成中のアプリケーションに合うような形に書き換えたとする．このとき，ライブラリ l_2 があるいは l_2 が使っているライブラリが l_1 を使っていた場合，l_1 中のメソッドが再定義されているため，l_2 の挙動まで変わってしまうことがある．

Ruby の refinement は open class のスコープを制限することでメソッドの衝突を避けられるようにするシステムである．refinement では，open class をモジュールの中に書くことで，そのモジュールの中でのみ open class を有効にすることができる．そうすることで，open class

```
1  class WebPage
2    def setPopupItem(popup, text)
3      # set text
4    end
5    def popup(text)
6      p = Popup.new()
7      setPopupText(p, text);
8    end
9  end
10
11 # ここから破壊的クラス拡張
12 class WebPage
13   def setPopupItem(popup, text)
14     alert(‘‘disabled’’)
15   end
16   def onClick(mouse)
17     url = mouse.getURL()
18     popup(url)
19   end
20 end
```

図 2.9. Ruby でのクラス宣言と open class．1 行目から 9 行目がクラス宣言を表し，12 行目から 20 行目が open class を表す．このように，特別なキーワードなしに普通のクラスやメソッドを宣言すると同じように open class を用いることができる．

の影響がプログラム全体に及ぶことを避ける事ができ，メソッドの衝突を回避できる．しかし，refinement では classbox 同様，有効にする open class を実行時に切り替える機能は持たない．

2.3.6　Worlds

Worlds[25] は副作用のスコープをコントロールすることができる言語コンストラクタである．Warth らは，[25] 中で worlds の仕様を説明し，worlds をサポートするように拡張した JavaScript と Smalltalk でこのアイデアの有用性の調査や形式化などを行っている．

言語コンストラクタ worlds では，全ての計算とそれで生じる副作用は一つの world の中に閉じ込められる．ある world での計算やその副作用は他の world に影響を与えない．メソッドも各 world 毎に定義することができ，他の world には影響を与えないため，同名のメソッドであるが異なるメソッドボディを持つメソッドを各 world に作ることができる．

Worlds を用いると，破壊的クラス拡張と似たようなことができる．それを説明するために，まず world の「発芽」と「commit」という機能を説明する．Worlds では，ある world から他の world を「発芽」させることができる．発芽とは，ある world をコピーして新しい world を

作り出すことを指す．発芽した world は，発芽元の world と親子関係を持つ．発芽した world を child world と呼び，発芽元の world を parent world と呼ぶ．child world は parent world に「commit」することができる．commit すると，child world での変更が parent world に伝搬される．つまり，parent world と child world に同名のメソッドが存在する場合には，parent world の定義は child world の定義で上書きされる．child world に parent world にないメソッドが存在する場合には，parent world に追加される．これは，破壊的クラス拡張でできることととてもよく似ている．

world を用いると，メソッドの追加/再定義ができ，かつどのメソッドのスコープも制限されているため，同名のメソッド定義を衝突を起こすことなく使い分けることができる．しかし，world にはいくつかの制限がある．まずメソッドの追加/再定義は world 間に親子関係がなければできない．そして，同じ world 内で同名のメソッドを複数切り替えながら用いることはできない．

2.3.7 JPred

破壊的クラス拡張に似たような機能として，predicate dispatch[16] がある．JPred[17] は，predicate dispatch を導入した Java ベースな言語である．predicate dispatch を用いると，同名のメソッドを複数定義することができる．どのメソッドが呼び出されるかは，プログラムのコンテキストによって変わる．プログラムのコンテキストは，コンテキストを表すメッセージやメソッドの引数によって指定される．JPred は Java の元々の型チェックやコンパイル方針は変えずに，predicate dispatch を導入するのに成功している．

predicate dispatch を用いれば，コンテキストを指定することによって複数の同名のメソッドを切り替えながら使うことができる．しかし，同名のメソッドのうち，どのメソッドを使うのかを毎度メッセージや引数によって指定しなければならない．

2.3.8 Us

Us[26] は Self[27] をベースにしたプログラミング言語で，subjective-oriented programming をサポートしている．subjective-oriented programming では，メソッド呼び出しごとに “perspective” と呼ばれるメソッド探索空間でメソッド探索が行われる．perspective はその名の通り，プログラムに対する「視点」のようなものであり，プログラムをどの *layer* を通して見るかを表したものである．layer はメソッドや変数といったプログラムのコードを含み，layer l を通して見るとはプログラムに layer l のコードを足すことを意味する．もし，layer l 中で定義されているメソッド m がプログラム中にあるならば，プログラム中のメソッド m は layer l 中のメソッド m で再定義されるし，ないならば追加される．layer は複数重ねることもでき，重ね方によって様々な perspective を設定することができる．また，perspective はメソッド呼び出しごとに指定するため，メソッド呼び出しごとに別の perspective を設定することができる．メソッド呼び出しごとに別の perspective を設定することによって，プログラム中で動的に有

効にするメソッドを切り替えることができる．

Us を用いると，このように破壊的クラス拡張のような機能を扱え，さらに有効にする破壊的クラス拡張を切り替えることもできるが，メソッド呼び出しごとに一々 perspective を指定する必要がある．また，一度 layer を重ねてしまうと再び重ね方を指定するまで layer の重なりを解くことはできない．そのため，layer 毎にモジュール化させるのには不向きである．

2.3.9　Context-oriented Programming

Context-oriented Programming(COP)[28] は，実行時にプログラムのコンテキストに応じて有効にするメソッドを切り替えられるプログラミングパラダイムである．COP を導入している言語は，Lisp ベースの ContextL[29]，Java ベースの ContextJ[30]，Smalltalk ベースな ContextS[31]，Python ベースな ContextPy[32] と PyContext[33]，など数多く存在している．COP では，layer と呼ばれるモジュールの中にメソッド定義を書く．異なる layer 中に同じメソッドの定義を含むことができ，どの layer を有効にするかをプログラム中で切り替えられる．切り替えには with 節と without 節を用いる．with 節で layer を有効にすることができ，without 節で無効にすることができる．

図 2.10 と図 2.11 が ContextJ[30] のコード例である．ContextJ は Java ベースな言語で Context-oriented Programming をサポートしている．ContextJ では，クラスの中に layer を書くことができる．図 2.10 のコードでは，WebPage クラスと Viewer クラスの 2 つのクラスが宣言されており，それぞれのクラスの中で rendrer, browser, viewer という名前の 3 つの layer が宣言されている．3 つの layer の中にはそれぞれの用途に応じたメソッドが定義されている．特徴的なのは，3 つの layer がそれぞれ setPopupItem メソッドを持っていることだろう．このように COP では異なる layer に同名のメソッドを定義することができる．図 2.11 のコードでは，App クラスが宣言されており，その中で app layer が宣言されている．App クラスの main メソッドの中では，with 節と without 節が使われている．with 節では，指定した layer がブロックの中で有効になる．without 節では，指定した layer がブロックの中で無効になる．有効な layer を with 節と without 節を用いて切り替えることで，2.2 章で紹介したメソッド衝突の問題を解決している．

しかし，COP では図 2.11 の main メソッドのように，有効な破壊的クラス拡張を切り替えたいときには常にブロックで指定しなければならない．そのため，プログラマがプログラム中にどのような layer があるか，現在どの layer が有効か，あるいは無効か，といった情報を常に把握している必要がある．これは，他のプログラマが開発したコードの layer 構造だけでなく，ライブラリやフレームワークといったブラックボックスとして扱いたいプログラムの layer 構造も把握する必要があることを意味する．ブラックボックスとして扱いたいプログラムもブラックボックスとすることができないため，コードの再利用には不向きである．

また ContextJ では，layer はクラスの中に記述されるため，差分のみを別ファイルに記述することは不可能である．これは ContextJ 特有の問題であるため，COP 全体の問題ではない．実際，別の COP をサポートする言語では異なる layer の構文や構造が用いられている．

```
1  class WebPage{
2    layer(renderer){
3      void setPopupItem(Popup p, HTML text){
4        //set text
5      }
6      void popup(HTML text){
7        Popup p = new Popup();
8        setPopupText(p, text);
9      }
10   }
11   layer(browser){
12     void setPopupItem(Popup p, HTML text){
13       alert(‘‘disabled’’);
14     }
15     void onClick(Mouse m){
16       URL url = m.getURL();
17       popup(url);
18     }
19   }
20   layer(viewer){
21     void setPopupItem(Popup p, HTML text){
22       //show red and bold text
23     }
24   }
25 }
26
27 class Viewer{
28   layer(viewer){
29     void check(file f){
30       if(isConfidential(f))
31         new WebPage.popup(‘‘confidential’’);
32     }
33   }
34 }
```

図 2.10. COP のコード例．WebPage クラスと Viewer クラスが宣言されており，その中で renderer layer と browser layer，viewer layer が宣言されている．

ContextJ においては，これが破壊的クラス拡張のコードの再利用という利点を得難くする要因の 1 つになっている．

```
1  class App{
2    layer(app){
3      void view(WebPage w){
4        new Viewer().check(new File(‘‘test.txt’’));
5      }
6    }
7    public static void main(String[] args){
8      WebPage w = new WebPage();
9      with(renderer){
10       with(browser){
11         with(viewer){
12           with(app){
13             without(browser){
14               new App().view(w);
15             }
16             without(viewer){
17               w.onClick();
18             }
19           }
20         }
21       }
22     }
23   }
24 }
```

図 2.11. COP のコード例．App クラスが宣言されており，その中で app layer が宣言されている．main メソッドの中では，with 節と without 節が用いられており，layer の有効/無効が切り替えられている．

2.3.10　Method Shelters

Method Shelters[34] は本研究の前身研究であり，破壊的クラス拡張を導入しているモジュールシステムである．[34] では，Method Shelters を Ruby ベースなモジュールシステムとして実装している．Method Shelters を用いると，破壊的クラス拡張のスコープを制限することができる．Method Shelters は，**method shelter** というモジュールを持っており，method shelter は **hidden chamber** と **exposed chamber** という二つの部分に分かれている．それぞれの chamber には open class を含めた Ruby のコードを記述することができる．method shelter は他の method shelter をインポートすることができ，インポートした method shelter の exposed chamber に記述されたコード中のメソッドのみ，インポートしている method shelter から呼び出したり，open class でさらに再定義したりできる．インポートした method shelter の hidden chamber に記述されたメソッドは，呼び出すことも再定義することもできない．exposed

chamber と hidden chamber を使い分けることで，破壊的クラス拡張の影響がプログラム全体に及ぶことを防ぐことができ，メソッドの衝突を回避できる．また，hidden chamber に記述されたメソッドはそれ以降再定義されないため，メソッドを意図せぬ再定義から守ることにもつながる．

Method Shelters はメソッドの衝突の回避にかなり有効であるが，モジュールの内部をさらに 2 つのモジュールに分けているため，構成が複雑である．また，複雑であるがゆえに形式化も難しかった．

本研究では，chamber を廃止し，2 種類のインポートを用意することでモジュールシステムの簡素化し，形式化を行った．また，Method Shelters は Ruby ベースな実装であったのに対し，本研究が提案する Method Shells は Java ベースの実装である．

第 3 章

Method Shells: メソッド衝突回避可能なモジュール機構

破壊的クラス拡張によるメソッド衝突を回避するために，本研究では新しいモジュール機構 *Method Shells* を提案する．Method Shells は有効にするメソッドを実行時に切り替えることで，メソッド衝突を回避することを可能にした．切り替えはグループごとに行われる．まとめて有効にしたいメソッドを shell グループと呼ばれるグループにグループ化し，そのグループをプログラマの事前の指定に応じて実行時に切り替える．なお，同一グループ内ではメソッド衝突を許さない．

本研究では Method Shells のプロトタイプを実装するために Java の拡張を行った．この拡張では，破壊的クラス拡張として GluonJ の reviser の文法を用いている．また，新しいモジュールとして *methodshell* を導入している．methodshell は Java の package と似たようなもので，クラス宣言と reviser を複数含み，モジュール化できる．Method Shells ではメソッドを 2 段階にグループ化する．1 段階目のグループ化は methodshell によるグループ化で，クラス宣言と reviser をまとめることができる．2 段階目のグループ化は shell グループによるグループ化で，methodshell をグループ化できる．また，methodshell には include 宣言と link 宣言という 2 つの宣言を記述することができる．この 2 つの宣言により，shell グループのグループ化や切り替えをコントロールする．

3.1 モジュール methodshell

Method Shells のモジュールの最小単位は methodshell である．methodshell は Java の package と似たようなもので，複数の reviser とクラス宣言を含むことができる．同じ methodshell 中で定義されたクラス宣言と reviser は実行時にまとめて有効になる．図 3.1 は renderer と名付けられた methodshell の定義である．renderer methodshell は WebPage クラスのクラス宣言のみを含むため，Java の package とよく似た形をしている．

methodshell 中には main メソッドという特別なメソッドを一つだけ記述できる．プログラムの実行はこの main メソッドから始まる．main メソッドを持つ methodshell を選択し，実行

```
1  methodshell renderer;
2  class WebPage{
3    void setPopupItem(Popup p, HTML text){
4      //set text;
5    }
6    void popup(HTML text){
7      Popup p = new Popup();
8      setPopupItem(p, text);
9    }
10 }
11 public static void main(String[] args){
12   WebPage w = new WebPage();
13   w.popup(‘‘test’’);
14 }
```

図 3.1. renderer methodshell．renderer methodshell 中では，WebPage クラスと main メソッドが宣言されている．

することでプログラムは実行される．renderer methodshell を選択して実行すると，renderer methodshell 中のクラス宣言と reviser が有効になり，renderer methodshell の main メソッドが実行される．renderer methodshell の main メソッド中では，WebPage クラスの popup メソッドが呼び出されている．popup メソッドは renderer methodshell 中の WebPage クラスの宣言の中で定義されているため，その popup メソッドが呼び出され，“test” というテキストを表示するポップアップが表示される．

3.2 methodshell 拡張と shell グループ

include 宣言を用いると，methodshell を拡張できる．include 宣言をしている methodshell が選択されると，その methodshell が include している methodshell も芋づる式に選択される．図 3.2 の browser methodshell は 2 行目で renderer methodshell を include している．これにより，browser methodshell を選択すると，renderer methodshell も芋づる式に選択され，browser methodshell と renderer methodshell で定義されたクラス宣言と reviser はまとめて有効になる．この include 宣言により芋づる式に有効になる一連の methodshell の集まりを **shell グループ**と呼ぶ．

include した methodshell 中のクラス宣言と reviser の中のメソッドは reviser で再定義できる．図 3.2 で browser methodshell 中の setPopupItem メソッドは renderer methodshell 中の setPopupItem メソッドを再定義している．browser methodshell を選択し，その main メソッドを起動させると，renderer methodshell も芋づる式に選択され，renderer methodshell と browser methodshell のクラス宣言と reviser がまとめて有効になる．main メソッド中では，WebPage クラスの popup メソッドが呼び出される．そして，その中の setPopupItem メソッ

```
1  methodshell browser;
2  include renderer;
3  revise WebPage{
4    void setPopupItem(Popup p, HTML text){
5      alert(‘‘disabled’’);
6    }
7    void onClick(Mouse m){
8      URL url = m.getURL();
9      popup(url);
10   }
11 }
12 public static void main(String[] args){
13   WebPage w = new WebPage();
14   w.popup(‘‘Available?’’); //not shown
15 }
```

図 3.2. browser methodshell．renderer methodshell を include し，その WebPage クラスを再定義している．

ドの呼び出しでは，renderer methodshell ではなく browser methodshell 中の setPopupItem メソッドが呼び出される．なぜなら，browser methodshell の setPopupItem メソッドは renderer methodshell の setPopItem メソッドを再定義しているからである．そのため，ポップアップウィンドウは表示されずアラートが表示される．

include は推移的である．そのため，include を繰り返すことで，同じメソッドを何度も再定義したり，shell グループを拡張してまとめて有効にする reviser を増やしたりできる．例えば，methodshell A が methodshell B を include しており，methodshell B が methodshell C を include していたとする．このとき，methodshell A は methodshell B 内のメソッドだけでなく，methodshell C 内のメソッドも再定義することができる．また，methodshell A が選択されると，methodshell B と C も芋づる式に選択され，3 つの methodshell 内のクラス定義と reviser がまとめて有効になる．

include は循環を許さない．そのため，

- 2 つの異なる methodshell が互いを include していること
- 異なる methodshell 内の破壊的クラス拡張が同じメソッドを相互に再定義すること

のような状態は存在しない．

同一の methodshell を再利用して異なる shell グループを作ることもできる．図 3.3 では，renderer methodshell を include している viewer methodshell を定義している．実行する際に browser ではなく，viewer methodshell を選択することで，viewer methodshell をルートとする shell グループが有効になり，renderer methodshell も同時に有効になる．「methodshell S をルートとする shell グループ」とは，S が推移的に include している methodshell と S からな

```
1  methodshell viewer;
2  include renderer;
3  revise WebPage{
4    void setPopupItem(Popup p, HTML text){
5      //show red and bold text
6    }
7  }
8  class Viewer{
9    void check(file f){
10     if(isConfidential(f))
11       new WebPage.popup(‘‘confidential’’);
12   }
13 }
```

図 3.3. viewer methodshell

る methodshell の集合を指す．そのため，「viewer methodshell をルートとする shell グループ」とは，viewer methodshell と renderer methodshell からなる集合を指す．

これ以降，各 shell グループをルートとなる methodshell の名前を付けた形で便宜上識別することにする．例えば，browser methodshell をルートとし browser methodshell と renderer methodshell からなる shell グループを「browser shell グループ」と呼び，viewer methodshell をルートとし viewer methodshell と renderer methodshell からなる shell グループを「viewer shell グループ」と呼ぶ．1 つの methodshell から構成される shell グループも存在する．「renderer shell グループ」は renderer methodshell のみから構成される shell グループである．

3.3 shell グループの切り替え

複数の shell グループを切り替えながら使いたいときに用いるのが link 宣言である．methodshell 中に link 宣言を記述すると，その methodshell から移動したい移動先の shell グループを指定することができる．link 宣言で指定した shell グループ中で定義または再定義されているメソッドを呼び出すと，有効な shell グループがその shell グループに自動的に切り替わる．これ以降，有効な shell グループのことを，**カレントグループ**と呼ぶ．include 宣言が推移的であったのに対し，link 宣言は宣言している methodshell の中でしか効果を発揮しない．つまり，link している shell グループへのカレントグループの切り替えは，link 宣言をしている methodshell の中からのみ行える．そのため，適切にモジュールを分けることで，意図した通りの動作が可能となる．

図 3.4 の app-for-viewer methodshell は viewer shell グループを link し，browser shell グループを include している．カレントグループが app-for-viewer shell グループであるとき，app-for-viewer，browser，renderer methodshell 中のクラス宣言と reviser が有効になる．このとき，app-for-viewer methodshell 中で check メソッドを呼び出したとする．check メ

```
1 methodshell app-for-viewer;
2 include browser;
3 link viewer;
4 class App{
5   void view(WebPage w){
6     new Viewer().check(new File(‘‘test.txt’’));
7          // switch the shell group
8   }
9 }
```

図 3.4. app-for-viewer methodshell

```
1 methodshell app;
2 include app_for_viewer;
3 revise App{
4   ...
5 }
6 public static void main(String[] args){
7   WebPage w = new WebPage();
8   w.onClick(); //prevent popup window
9   view(w); // show popup window
10 }
```

図 3.5. app methodshell

ソッドは viewer shell グループ中にあるため，check メソッドが呼ばれた瞬間にカレントグループは app-for-viewer shell グループから viewer shell グループに切り替わり，viewer methodshell と renderer methodshell のクラス宣言と reviser が有効になる．この間，app-for-viewer shell グループのクラス宣言と reviser は無効になる．そのため，check メソッドの実行中に setPopupItem メソッドが呼ばれると，viewer methodshell 中の setPopupItem メソッドが呼び出される．check メソッドの実行が終了すると，カレントグループは再度 app-for-viewer shell グループとなる．viewer shell グループを使いたい部分を適切にモジュールに切り出し，viewer shell グループと link させることで意図した通りのカレントグループの切り替えが可能となる．

3.4　利用例

2 章の図 2.4 のコードで実現したかったことは methodshell を用いると図 3.1~ 図 3.5 のように実現できる．図 3.6 は図 3.1~ 図 3.5 までの 4 つの methodshell の include，link 関係を図示している．それぞれの四角が methodshell を表しており，実線の矢印が include 関係を，点線の矢印が link 関係を表している．さらに，点線の曲線で囲まれている部分が 1 つの shell グ

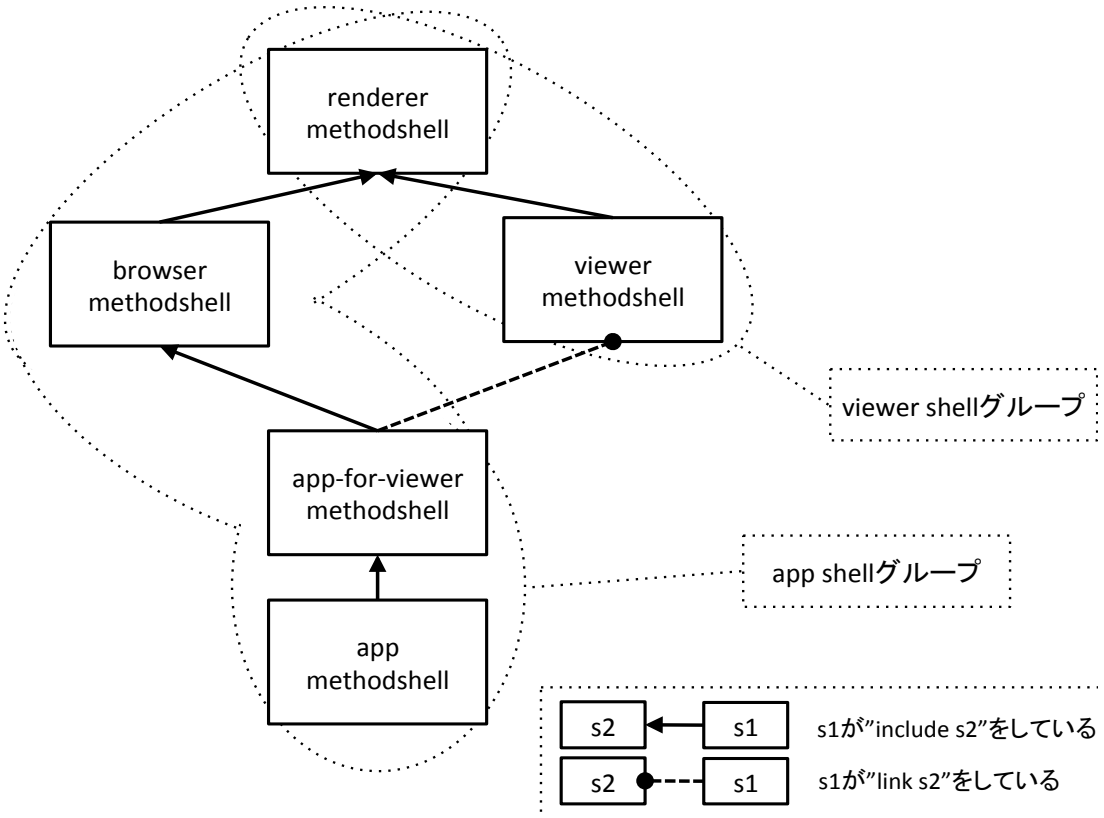

図 3.6. 図 3.1~ 図 3.5 の methodshell の include，link 関係

ループを表している．図中には図 3.1~ 図 3.5 で定義した 5 つの methodshell が図示されている．また，今後の説明で用いる app shell グループと viewer shell グループという 2 つの shell グループのみ図示している．

このプログラムを app methodshell を選択して実行すると，app methodshell の main メソッドが実行される．そして，カレントグループは app methodshell をルートとし，app, app-for-viewer, browser, renderer methodshell からなる shell グループ，つまり app shell グループになる．app methodshell を選択するだけで，app-for-viewer, browser, renderer methodshell は芋づる式に選択され，4 つの methodshell 中のクラス宣言と reviser が有効になる．

app methodshell の main メソッド中の onClick メソッド呼び出しでは，browser methodshell 中の onClick メソッドが呼び出される．カレントグループは app shell グループのままであり，onClick メソッドが呼ばれている間，app shell グループ中で定義されているクラス宣言と reviser が有効になる．そのため，onClick メソッドが実行されてる間，ポップアップウィンドウを表示しようとすると，アラートメッセージが表示され，ポップアップウィンドウはブロックされる．

onClick メソッドの実行が終了すると，次に view メソッドが呼び出される．view メソッドは app-for-viewer methodshell 内で定義されている．app-for-viewer methodshell は viewer

shell グループを link しているため，app-for-viewer methodshell 内から viewer shell グループ内のメソッドを呼び出すと，カレントグループが app shell グループから viewer shell グループに切り替わる．図 3.4 の view メソッド中では，check メソッドが呼ばれており，check メソッドは viewer shell グループ内で定義されている．そのため，check メソッドが呼ばれた瞬間に，カレントグループは viewer shell グループに切り替わり，check メソッドの実行が終了するまで，viewer shell グループ内で定義されているクラス宣言と reviser が有効になる．その間，app shell グループは無効になり，app shell グループ中のクラス宣言と reviser は無視される．viewer shell グループがカレントグループである間，ポップアップウィンドウは赤太字のテキストを表示する．check メソッドの実行が終了すると，カレントグループは viewer shell グループから app shell グループに戻され，view メソッドの続きが実行される．

3.5　セマンティクス

この章では Method Shells の形式的な意味論を示す．Method Shells の形式化は Featherweight Java(FJ)[35] と GluonFJ の拡張によって表現される．

3.5.1　文法

形式的な意味論を示す前に，まず形式化した Method Shells の文法を示す．文法は図 3.7 のようになっている．

メタ変数について説明する．S,T,U は methodshell の名前である．B,C,D,E はクラス名である．f はフィールド名である．m はメソッド名である．x は変数名である．SL は methodshell 宣言である．IL は include 宣言である．LL は link 宣言である．CL はクラス宣言である．K はコンストラクタ宣言である．RV は reviser 宣言である．M はメソッド宣言である．MF は main 関数の宣言である．e は式である．v,w は値である．列はオーバラインによって表す．例えば，「$\overline{x}$」は「$x_1, x_2, \cdots, x_n$」を表し，「$\overline{C}\ \overline{f};$」は「$C_1\ f_1;, C_2\ f_2;, \cdots, C_n\ f_n;$」を表し，「this.$\overline{f}$ = $\overline{f}$;」は「this.f_1 = f_1;, this.f_2 = f_2; $\cdots$, this.f_n = f_n を表す．

methodshell 宣言 methodshell S; $\overline{\text{IL}}\ \overline{\text{LL}}$ (CL || RV) $*$ MF は，S という名前の methodshell を宣言する．methodshell は include 宣言 $\overline{\text{IL}}$ と link 宣言 $\overline{\text{LL}}$，クラス宣言 CL，reviser 宣言 RV，main 関数 MF を含む．include 宣言 include S; は methodshell S を include することを表す．link 宣言 link S; は methodshell S を link することを表す．クラス宣言 class C extends D{$\overline{C}\ \overline{f}$; K $\overline{M}$} は，スーパクラスが D でクラス名 C のクラスを宣言する．クラスは，型が $\overline{C}$ のフィールド $\overline{f}$，1 つのコンストラクタ K，メソッド $\overline{M}$ を含む．クラス C でクラス D にあるメソッドと同じ名前のメソッドを宣言した場合には，クラス C のメソッドがクラス D のメソッドをオーバライドする．コンストラクタ宣言 C($\overline{C}\ \overline{f}$){super($\overline{f}$); this.$\overline{f}$ = $\overline{f}$; } はどのようにクラス C のフィールドを初期化するかを表す．reviser 宣言 revise C{$\overline{M}$} はクラス C の reviser を表す．reviser はメソッド宣言 $\overline{M}$ を含む．メソッド宣言 C m($\overline{C}\ \overline{x}$){return e; } は型 $\overline{C}$ の引数 $\overline{x}$ で返り値の型が D でメソッド名 m のメソッドを宣言する．main 関数宣言 void main(){return e; } は main 関数の

Syntax

SL	::=		*method-shell declaration* :
		methodshell S; $\overline{\text{IL}}$ $\overline{\text{LL}}$ (CL || RV) $*$ MF	
IL	::=		*include declaration* :
		include S;	
LL	::=		*link declaration* :
		link S;	
CL	::=		*class declaration* :
		class C extends C{$\overline{\text{C}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$}	
K	::=		*constructor declarations* :
		C($\overline{\text{C}}$ $\overline{\text{f}}$){super($\overline{\text{f}}$); this.$\overline{\text{f}}$ = $\overline{\text{f}}$; }	
RV	::=		*reviser declaration* :
		revise C{$\overline{\text{M}}$}	
M	::=		*method declaration* :
		C m($\overline{\text{C}}$ $\overline{\text{x}}$){return e; }	
MF	::=		*main method declaration* :
		void main(){return e; }	
e	::=		*terms* :
		x	*variable*
		e.f	*find access*
		e.m($\overline{\text{e}}$)	*method invocation*
		new C($\overline{\text{e}}$)	*object creation*
		(C)e	*cast*
		e in S; S_C	*inshell-group*
v	::=		*values* :
		new C($\overline{\text{v}}$)	

図 3.7. 文法

宣言を表す．Method Shells で書かれたプログラムはこの main 関数を実行することで始まる．式 e は，x, e.f, e.m($\overline{\text{e}}$), new C($\overline{\text{e}}$), (C)t, e in S; S_C の 5 つの形を持つ．前 4 つは FJ で紹介されている．e in S; S_C は我々が導入した新しい式である．この式はセマンティクスを示すために用意したもので，プログラム中には現れない．これは式 e が S という methodshell にあり，カレントグループが S_C であることを示している．

CRT をクラス・reviser テーブルを表すメタ変数とする．クラス・reviser テーブルは methodshell S とクラス名 C のペアをクラス宣言 CL または reviser 宣言 RV にマッピングする．プ

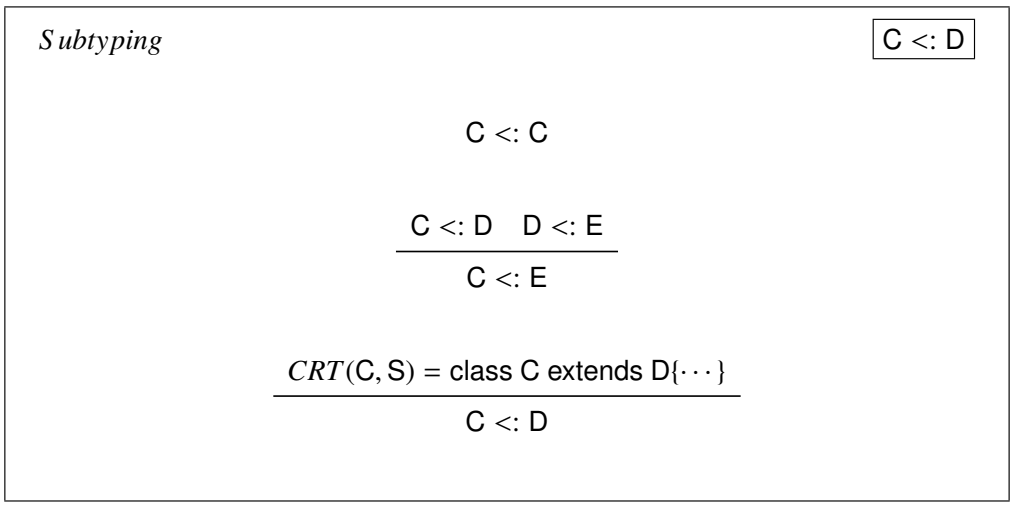

図 3.8. 部分型の規則

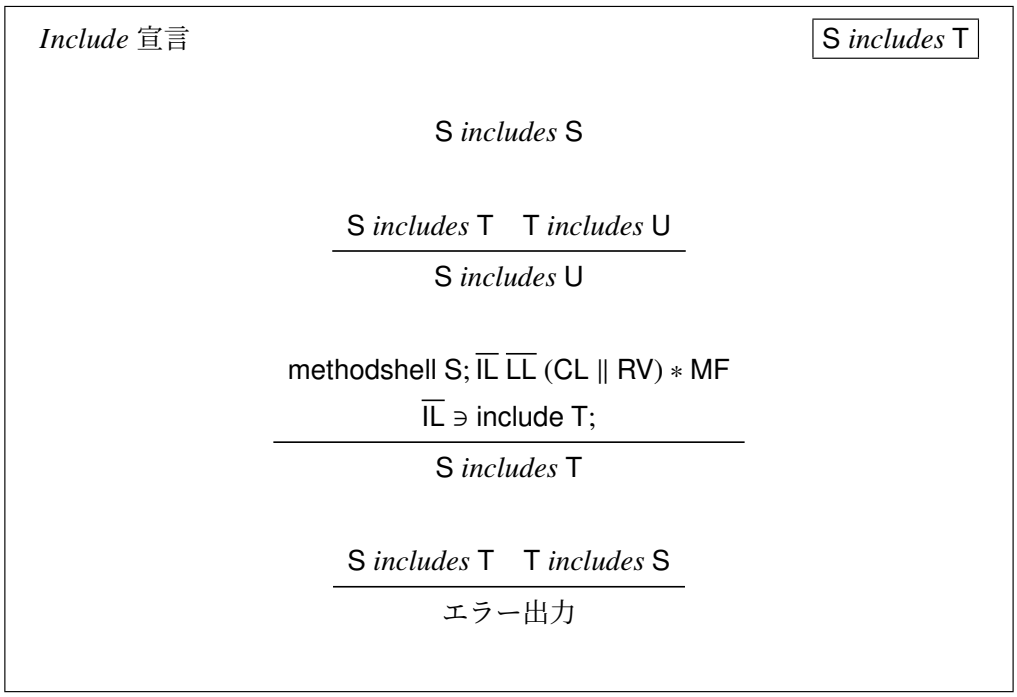

図 3.9. include 宣言の規則

ログラムは *CRT* と methodshell 名 S のペア (*CRT*, S) である．プログラム実行は最初に与えられる methodshell 中の main 関数から始まる．

部分型規則を図 3.8 に示す．C <: D のように書き，クラス C がクラス D の部分型であることを表す．部分型関係はクラステーブルを見ることによって，定めることができる．部分型は *CRT* 中のクラス宣言から得られるサブクラス関係の反射推移閉包となる．図 3.8 の 1 つ目の規則が反射率を表し，2 つ目の規則が推移律を表す．また，<: で表される関係が循環することはない．

include 関係の規則を図 3.9 に示す．S *includes* T のように書き，methodshell S が include T を宣言していることを表す．methodshell の include 関係は methodshell 宣言から得られる．1 つ目の規則が反射率を表し，2 つ目の規則が推移律を表す．また，興味深いのは最後の規則であり，include が循環する場合にはエラーを出力することを表す．

3.5.2 メソッド呼び出しのセマンティクス

簡約は $\mathsf{S}; \mathsf{S_C} \vdash \mathsf{e} \rightarrow \mathsf{e}'$ の形で表し，「カレントグループが $\mathsf{S_C}$ であるとき，methodshell S 中で，式 e は 1 ステップで式 e′ に評価される」と読む．プログラム実行がスタートする時に，methodshell S が選択されたとすると，カレントグループは S になり，式 e は methodhell S の中で簡約される．

簡約規則は図 3.10 のようになる．ほとんどの簡約規則は FJ と GluonFJ のものに従うが，(R-In) と (R-Invk) のみ methodshell 特有である．FJ では，call-by-value の戦略が取られているが，Method Shells でもその戦略を取ることとする．$[\overline{\mathsf{d}}/\overline{\mathsf{x}}, \mathsf{e}/\mathsf{y}]\mathsf{e}_0$ は式 e_0 中の x_0 を d_0 で，x_1 を d_1 で，$\cdots$，x_n を d_n で置き換え，y を e で置き換えた結果を表す．

(R-In) は，$\mathsf{T}; \mathsf{T_C}$ が与えられているときに e_0 が e'_0 に簡約できるならば，e_0 は $\mathsf{S}; \mathsf{S_C}$ が与えられているところでも in によって $\mathsf{T}; \mathsf{T_C}$ 中で簡約されることを示している．(R-Invk) は，メソッド探索をするための関数 *mbody* についてである．*mbody* は FJ での *mbody* とは異なり，4 つの引数をとる．*mbody* はメソッド名 m，対象オブジェクトのクラス名 C，その式が存在している methodshell S，カレントグループ $\mathsf{S_C}$ を引数としてとり，探索を行う．(R-Invk) は，新しいカレントグループが $\mathsf{T_C}$ となるような methodshell T 中で対象メソッドが見つかったならば，メソッド呼び出しは簡約され，カレントグループが $\mathsf{S_C}$ から $\mathsf{T_C}$ に変わることを表す．

fields 関数の形式化を図 3.11 に示す．*fields*(C) はクラス C のフィールドを返す．フィールドは $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ と書き，それぞれのフィールドの型と名前を表すペアの列で表現される．

mbody の式の形式化を図 3.12 に示す．$\mathit{mbody}(\mathsf{m}, \mathsf{C}, \mathsf{S}, \mathsf{S_C})$ は 4 つの引数からメソッド探索を行い，$\overline{\mathsf{x}}.\mathsf{e}\ \mathsf{in}\ \mathsf{T}; \mathsf{T_C}$ という形を返す．ここで $\overline{\mathsf{x}}$ は見つかったメソッドの引数を表し，e はメソッドボディを表し，T はメソッドが見つかった methodshell を表し，$\mathsf{T_C}$ は見つかったメソッドボディが実行されるカレントグループを表す．

mbody 中で使われているいくつかの関数について説明する．*linked-shells*(S) は methodshell S が link している methodshell の集合を返す．*include-shells*(S) は methodshell S が直接 include している methodshell の集合を返す．$\mathit{mbodyshell}(\mathsf{m}, \mathsf{C}, \mathsf{S})$ は methodshell S 内を探索するための関数であり，methodshell S 中のクラス C でメソッド m が見つかったならば，それを返す．見つからないならば，S が include している methodshell について，再帰的に *mbodyshell* を呼び出す．$\mathit{mbodyglobal}(\mathsf{m}, \mathsf{C}, \mathsf{S}, \mathsf{S_C})$ はグローバルなグループを探索する．グローバルなグループとは全ての methodshell から include されている特殊な methodshell である．特定の methodshell 内に含まれていないクラス宣言や reviser は全てグローバルなグループに所属していると考える．図 3.12 中の Global はグローバルなグループを表すメタ変数である．もし探索対象のメソッドがグローバルなグループで発見されないならば，C のスーパクラス D を探索

Reduction rules　　$\boxed{S; S_C \vdash e \longrightarrow e'}$

$$\frac{T; T_C \vdash e_0 \longrightarrow e_0'}{S; S_C \vdash e_0 \text{ in } T; T_C \longrightarrow e_0' \text{ in } T; T_C} \quad \text{(R-In)}$$

$$\frac{\mathit{fields}(C) = \overline{C}\,\overline{f}}{(\text{new } C(\overline{e})).f_i \longrightarrow e_i} \quad \text{(R-Field)}$$

$$\frac{\mathit{mbody}(m, C, S, S_C) = \overline{x}.e_0 \text{ in } T; T_C}{S; S_C \vdash \text{new } C(\overline{v}).m(\overline{w}) \longrightarrow ([\overline{w}/\overline{x}, \text{new } C(\overline{v})/\text{this}]e_0) \text{ in } T; T_C} \quad \text{(R-Invk)}$$

$$\frac{C <: D}{(D)(\text{new } C(\overline{v})) \longrightarrow \text{new } C(\overline{v})} \quad \text{(R-Cast)}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.f \longrightarrow e_0'.f} \quad \text{(RC-Field)}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.m(\overline{e}) \longrightarrow e_0'.m(\overline{e})} \quad \text{(RC-Invk-Recv)}$$

$$\frac{e_i \longrightarrow e_i'}{e_0.m(..., e_i, ...) \longrightarrow e_0.m(..., e_i', ...)} \quad \text{(RC-Invk-Arg)}$$

$$\frac{e_i \longrightarrow e_i'}{\text{new } C(..., e_i, ...) \longrightarrow \text{new } C(..., e_i', ...)} \quad \text{(RC-New-Arg)}$$

$$\frac{e_0 \longrightarrow e_0'}{(C)e_0 \longrightarrow (C)e_0'} \quad \text{(RC-Cast)}$$

図 3.10. 簡約規則

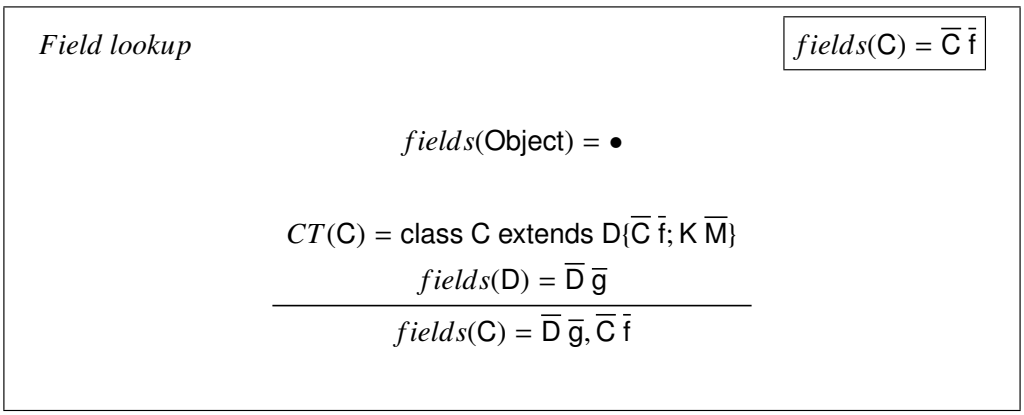

図 3.11. フィールド探索の規則

するために *mbody* を再帰的に呼び出す.

$mbody(\mathsf{m}, \mathsf{C}, \mathsf{S}, \mathsf{S_C})$ は次のような順序で探索を行う．まず，methodshell S が link している methodshell をルートとする shell グループの中を探索する．もしメソッドボディ m が発見されたならば，そのメソッドボディが見つかった shell グループが次のカレントグループになる．link している shell グループ内で発見されないならば，次にカレントグループ内を探索する．もしカレントグループでも発見されないならば，グローバルなグループを探索する．最後に，どの methodshell でも発見されないならば，クラス C のスーパクラス D を同じように探索する．link している shell グループでメソッドが発見された場合にのみ，カレントグループは変更される．

3.5.3 制限

Method Shells にはいくつかの制限がある.

まず，reviser 中では，フィールドの追加をすることができない．そのため，フィールドを追加しなければならないような拡張に，Method Shells を使うことはできない.

次に，include 宣言は循環を許さない．循環を許してしまうと，メソッド探索が停止しない恐れがある．循環を許さないからこそ，図 3.12 中で定義されている *mbodyshell* メソッドが無限ループに陥るのを防ぐことができる.

次に，methodshell 中には link 宣言を複数書くことができるが，探索しているメソッドが link された複数のコンテキスト内で定義されているとメソッド呼び出し時にエラーが起きる．図 3.12 中で形式化した関数 *mbody* に定義されている通り，link している shell グループの中からただ 1 つだけ探索しているメソッドが見つかった場合のみ，メソッド探索は成功する．これはプログラマの自由な破壊的クラス拡張を制限する．また，複数のライブラリを用いた時にエラーを引き起こしやすくなるため，プログラミングが複雑になる．しかし，これらの制限によりモジュールをインポートするのみで生じるメソッド衝突を検出することができるため，プログラマの意図せぬメソッド衝突を避けることにつながる．また，この制限により，セマンティ

クスを簡素化でき，次章で述べるような静的なコンパイルが可能になる．

同様に include している複数の methodshell 内で探索中のメソッドが宣言されているとエラーになる．ただし，include の場合は link と違い，同じ名前のメソッドが複数 include されていたとしても，再定義され，関数 *mbody* によるメソッド探索結果がただ 1 つに定まる場合にはエラーにはならない．link と同様にこの制限により，いくつかの不便を被る反面，意図せぬメソッド衝突の回避やセマンティクスの簡素化といった利点を得ることができる．

次に，link した methodshell 内のメソッドを再定義することはできない．link 宣言している methodshell 内で link している shell グループ内のメソッドを定義しても，常に link している shell グループ内のメソッドが優先される．この制限により，メソッドを再定義から守ることができるため，プログラムの挙動が変わることを防ぐことができる．ライブラリやフレームワークのようなブラックボックスとして扱いたいプログラムを意図せずに再定義することはなるべく避けたい．link 宣言を適切に使用することにより，ブラックボックスとして扱いたいプログラム中のメソッド呼び出しを気付かずに再定義し，挙動が変わってしまうことを避けることができる．

Method body lookup $\boxed{mbody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t})}$

$$\frac{\overline{\mathtt{T_c}} = linked\text{-}shells(\mathtt{S}) \quad \exists \mathtt{T_{ci}} \in \overline{\mathtt{T_c}} \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{T_{ci}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T} \quad \forall \mathtt{T_{cj}} \in \overline{\mathtt{T_c}}(\mathtt{i} \neq \mathtt{j}) \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{T_{cj}})\ \mathtt{undefined}}{mbody(\mathtt{m}, \mathtt{C}, \mathtt{S}, \mathtt{S_c}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T}; \mathtt{T_{ci}}}$$

$$\frac{\overline{\mathtt{T_c}} = linked\text{-}shells(\mathtt{S}) \quad \forall \mathtt{T_{ci}} \in \overline{\mathtt{T_c}} \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{T_{ci}})\ \mathtt{undefined} \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{S_c}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T}}{mbody(\mathtt{m}, \mathtt{C}, \mathtt{S}, \mathtt{S_c}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T}; \mathtt{S_c}}$$

$$\frac{\overline{\mathtt{T_c}} = linked\text{-}shells(\mathtt{S}) \quad \forall \mathtt{T_{ci}} \in \overline{\mathtt{T_c}} \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{T_{ci}})\ \mathtt{undefined} \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{S_c})\ \mathtt{undefined}}{mbody(\mathtt{m}, \mathtt{C}, \mathtt{S}, \mathtt{S_c}) = mbodyglobal(\mathtt{m}, \mathtt{C}, \mathtt{S}, \mathtt{S_c})}$$

$$\frac{}{mbody(\mathtt{m}, \mathtt{C}, \mathtt{null}, \mathtt{null}) = mbodyglobal(\mathtt{m}, \mathtt{C}, \mathtt{null}, \mathtt{null})}$$

$$\frac{\mathtt{CRT}(\mathtt{S}, \mathtt{C}) = (\mathtt{revise}\ \mathtt{C}\{\overline{\mathtt{M}}\}) \parallel (\mathtt{class}\ \mathtt{C} \cdots \{\cdots \overline{\mathtt{M}}\}) \quad \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{M}}}{mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{S}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{S}}$$

$$\frac{\mathtt{CRT}(\mathtt{S}, \mathtt{C}) = (\mathtt{revise}\ \mathtt{C}\{\overline{\mathtt{M}}\}) \parallel (\mathtt{class}\ \mathtt{C} \cdots \{\cdots \overline{\mathtt{M}}\}) \quad \mathtt{m}\ \mathtt{is\ not\ defined\ in}\ \overline{\mathtt{M}} \quad \overline{\mathtt{S}} = include\text{-}shells(\mathtt{S}) \quad \exists \mathtt{S_i} \in \overline{\mathtt{S}} \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{S_i}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T} \quad \forall \mathtt{S_j} \in \overline{\mathtt{S}}(\mathtt{i} \neq \mathtt{j}) \quad mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{S_j})\ \mathtt{undefined}}{mbodyshell(\mathtt{m}, \mathtt{C}, \mathtt{S}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T}}$$

$$\frac{\mathtt{CRT}(\mathtt{Global}, \mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \quad \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{M}}}{mbodyglobal(\mathtt{m}, \mathtt{C}, \mathtt{S}, \mathtt{S_c}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{null}; \mathtt{null}}$$

$$\frac{\mathtt{CRT}(\mathtt{Global}, \mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \quad \mathtt{m}\ \mathtt{is\ not\ defined\ in}\ \overline{\mathtt{M}}}{mbodyglobal(\mathtt{m}, \mathtt{C}, \mathtt{S}, \mathtt{S_c}) = mbody(\mathtt{m}, \mathtt{D}, \mathtt{S}, \mathtt{S_c})}$$

図 3.12. メソッド探索の規則

第 4 章

メソッド呼び出しの最適化

本章では，Method Shells のメソッド呼び出しの最適化を行った実装方法を提案する．本論文では 3.5 章にて Method Shells のセマンティクスを示した．これを Java 上に実装することを考えたときに，セマンティクスをそのままに実装すると，methodshell の増加に伴い，実行時間が増加してしまう場合がある．例えば，ある methodshell S が n 個の methodshell を link しているとき，*mbody* の最初のステップで n 個の methodshell 全ての探索を行う必要がある．そのため，n の値が増加するに従って，メソッド探索の時間が増加することが考えられる．

実行時間の最小化のために，コンパイル時に複数のグループに所属するメソッドをグループの数だけコピーするような変換を行うことにした．つまり，メソッド m_{S_i} を $m_{S_i \cdot T_1}, m_{S_i \cdot T_2}, m_{S_i \cdot T_3}, \cdots$ のように複製し，T_j がカレントグループであるときに $m_{S_i \cdot T_j}$ を用いるようにする．メソッドボディの方も S_i と T_j に合わせた形に書き換える．こうようにメソッドをコピーすることで，1 つのメソッド名に対し，1 つのメソッドボディが対応するようになるため，実行時のメソッド探索コストが抑えられる．

型環境 Γ の下で methodshell が S でカレントグループが S_C のときのコンパイル時の変換規則を $S; S_C; \Gamma \vdash e \Rightarrow e'$ と書くことにする．ほとんどの式で大きな変換を行う必要はないが，メソッド呼び出しのみ図 4.1 のような変換を行う．$S; S_C; \Gamma \vdash e_0 : C$ は「式 e_0 は型環境 Γ の下で型 C が与えられる」と読む．

mname は *mbody* のようなメソッド探索の関数で，図 4.2 のように定義される．*mname* はメソッド名 m，対象オブジェクトのクラス名 C，その式が存在している methodshell S，カレントグループ S_C の 4 つの引数をとる．*mname* は *mbody* による探索結果に該当するメソッド名を返す．

グローバルなグループからメソッドが選ばれるとき，呼び出すメソッド名は変換されない．

プログラムが methodshell S の main メソッドから実行を始めるならば，その main メソッドボディは methodshell S でカレントグループ S に合わせた形に変換される．

この実装方法を用いると実行時間の増加を抑えられるが，代わりに分割コンパイルができなくなってしまう．なぜなら，実行したい shell グループの include/link 関係をコンパイル時に静的に決定できる必要があるためである．一方，3.5 章のセマンティクス通りに実装すると，実行時に動的にメソッドを探索するので分割コンパイルも容易である．

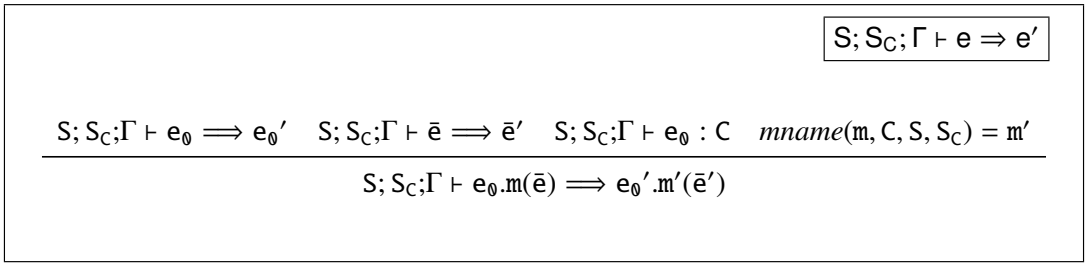

図 4.1. コンパイル時のメソッド呼び出しの変換規則

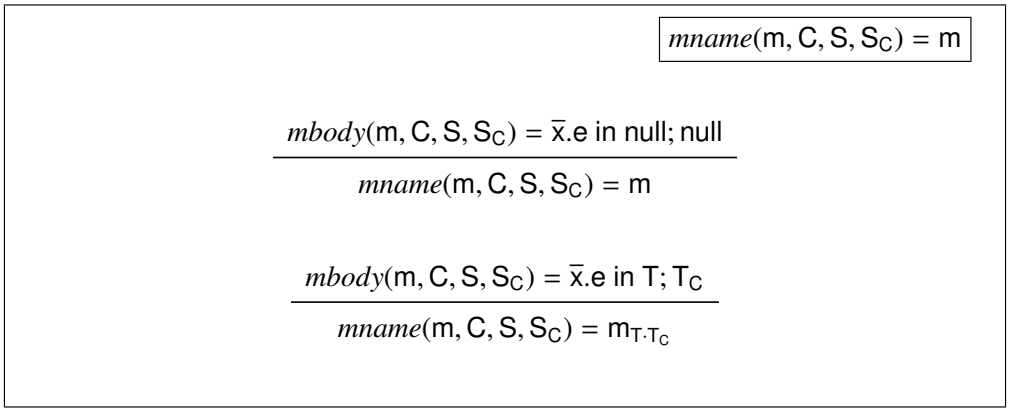

図 4.2. 関数 *mname* の定義

第 5 章

実装

本研究では，4 章の最適化手法を用いて Method Shells のコンパイラのプロトタイプを実装した．Method Shells のコンパイラのプロトタイプは，JastAddJ[36, 37] という Java コンパイラを拡張することで実現した．この章では，その実装の方法および実装に用いた JastAddJ というコンパイラについて述べる．

5.1 JastAddJ

JastAddJ は JastAdd[38] という Java ベースなコンパイラを作成するシステムを用いて作られた拡張可能な Java コンパイラである．JastAdd を用いると，差分のみを別ファイルに記述し，それを追加することでコンパイラを拡張することができる．JastAddJ 自身も JastAdd を利用して作られ，拡張されている．JastAddJ のベースは Java1.4 であり JastAdd を利用して作られているが，さらに Java1.5 と Java7 の処理を別ファイルに記述し追加することで拡張もされている．

JastAddJ は主に以下の 4 つの部分からなる．

- *abstract grammar*：抽象構文木 (AST) の構造を定義する
- *behavior specifications*：AST の振る舞いを定義する
- *context-free grammar*：プログラムをどのように AST に構文解析するかを定義する
- *main program*：入力として渡されたプログラムを読み，構文解析器を実行することで AST を取得し，AST の振る舞いによって出力を生成する

この 4 つの部分からどのようにコンパイラが作られるかを図 5.1 に示す．JastAddJ を作るために定義する必要があるのは，上の 4 つと字句定義のみである．

abstract grammar と behavior specification は JastAdd 言語を用いて記述される．JastAdd はこの 2 つをまとめて処理し，オブジェクト指向の Java 言語で書かれたクラスを出力する．このクラスが AST の各ノードを表す．

context-free grammar は Java 言語の構文解析器を生成するような既存のパーサジェネレータによって処理される．実際の JastAddJ では，パーサジェネレータとして Java ベースな構文

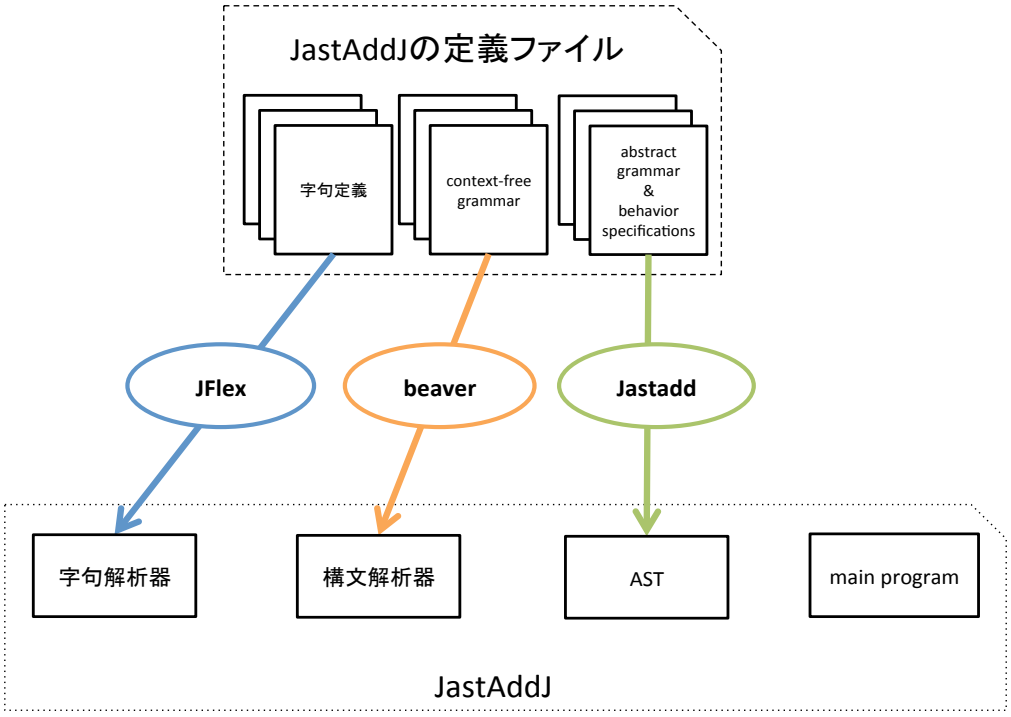

図 5.1. JastAddJ の生成概略図．字句解析定義から JFlex を用いて字句解析器が生成される．abstract grammar から beaver を用いて構文解析器が生成される．behavior specifications と context-free grammar から AST が生成される．

解析器を構築する beaver を用いている．また，字句解析器生成には Java ベースな字句解析器を構築する JFlex を用いている．beaver を用いて context-free grammar から構文解析器を生成し，JFlex を用いて字句定義から字句解析器を生成する．

main program は生成された字句解析器，構文解析器，AST をつなぐ役割を果たす．main program が字句解析器や構文解析器などを実行し，それぞれの結果を次の実行へと受け渡す．JastAddJ では，main program は字句解析器 → 構文解析器 → エラーチェック → クラスファイル出力，の順に実行するようコードが書かれている．

JastAddJ のオリジナルの abstract grammar, behavior specifications, context-free grammar, main program と，追加したい処理を定義した abstract grammar, behavior specifications, context-free grammar, main program を JastAdd で処理すれば，JastAddJ を望むようなコンパイラへと拡張できる．また，追加したい abstract grammar, behavior specifications, context-free grammar, main program は別ファイルに記述することができ，差分のみをモジュール化することができる．つまり，JastAddJ は破壊的クラス拡張の応用の一つである．

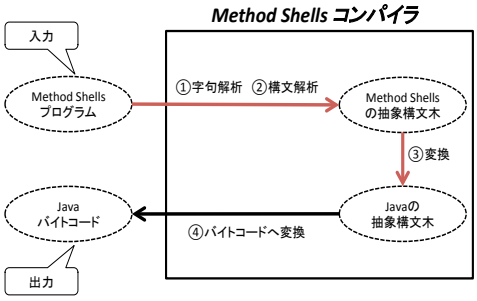

図 5.2. Method Shells のコンパイラの
プロトタイプの概略図

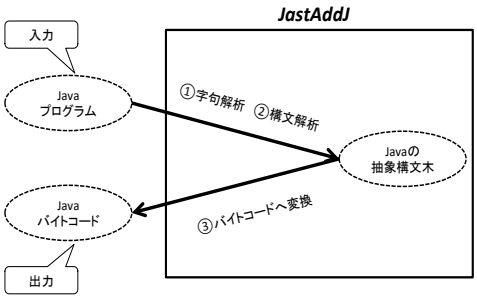

図 5.3. JastAddJ の概略図

5.2　実装手法

Method Shells のコンパイラのプロトタイプは，JastAddJ[36, 37] を拡張することで実現した．図 5.2 が Method Shells コンパイラの概略図であり，図 5.3 が JastAddJ の概略図である．本来，JastAddJ は図 5.3 のように Java プログラムを入力として受け取って，字句解析し，構文解析し，Java の AST を作り出し，それを Java バイトコードへと変換し，出力する．本研究ではこれを拡張し，Method Shells のプログラムを入力とし，Java バイトコードを出力するようにした．

新たに追加した部分は図 5.2 の赤い矢印の部分である．JastAddJ に 3.5 章で定義した文法を追加し，Method Shells の抽象構文木を 4 章の最適化手法に従って Java の抽象構文木に変換するような処理を追加した．具体的には，① 字句解析 ② 構文解析の処理を 3.5 章の定義に従って context-free grammar で定義し，③ 変換の処理を 4 章の最適化手法に従って abstract grammar と behavior specifications で定義した．それらを JastAddJ のファイルと合わせて，JFlex を用いて字句解析器を，beaver を用いて構文解析器を，JastAdd を用いて AST を生成することで，コンパイラを作成した (図 5.4)．④ の Java の AST をバイトコードへ変換する部分については，元々の JastAddJ の機能を再利用している．

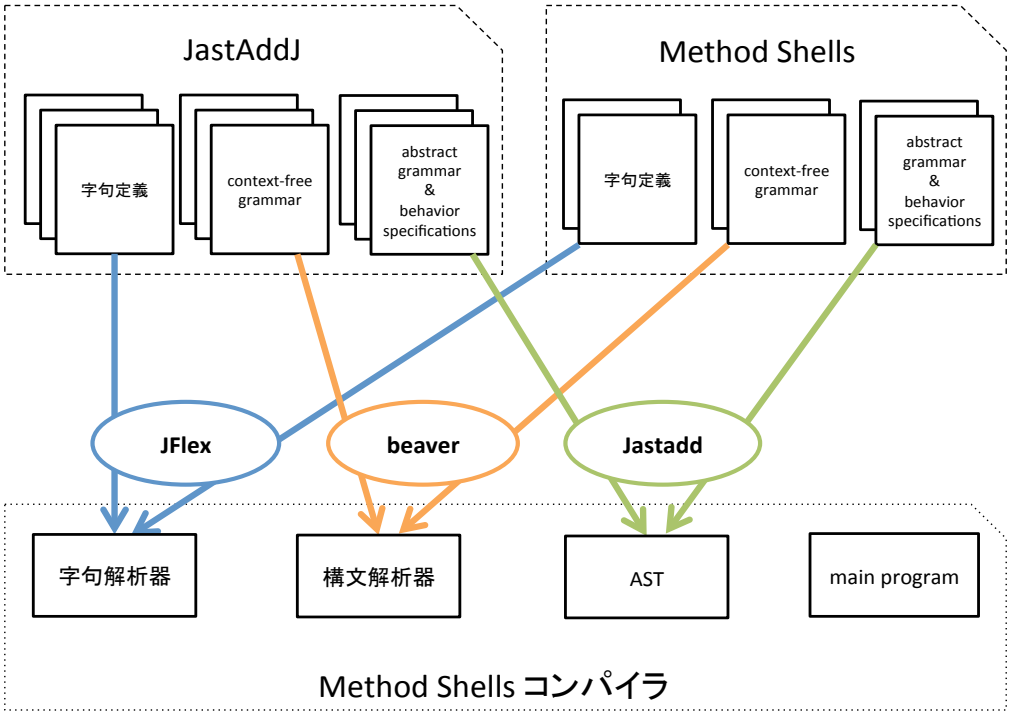

図 5.4. Method Shells コンパイラのプロトタイプの生成概略図．Method Shells と JastAddJ の字句解析定義から字句解析器が生成され，abstract grammar から構文解析器が生成され，behavior specifications と context-free grammar から AST が生成される．

第 6 章

実験

本章では，4 章の最適化手法の実装の効果を示すために行った実験について述べる．実験環境は，CentOS release 6.2，Intel(R) Xeon(R) CPU E5-2687W 3.10GHz，メモリは 64GB，OpenJDK 1.7.0 である．4 章の最適化手法による実装の実験には，5 章で述べた Method Shells のコンパイラのプロトタイプを用いた．

本研究では大きく分けて次の 2 つの実験を行った．

- セマンティクスに沿った実装と最適化手法による実装のメソッド探索の時間の比較
  - セマンティクスに沿った実装では，methodshell の数が増えた時にメソッド探索の時間が増加する場合がある．最適化手法による実装では，メソッド探索の時間の増加を抑えられることを確認する．
- GluonJ およびオリジナルの Java とのメソッド探索の時間の比較
  - GluonJ は破壊的クラス拡張を導入している言語であり，include 宣言に似た機能を有している．GluonJ と最適化手法を比較することにより，既存の破壊的クラス拡張を導入している言語と比べても，methodshell の include 宣言の影響が少ないことを確認する．
  - 破壊的クラス拡張を使わずに直接ソースコードを書き換えた場合とのメソッド探索の実行時間の比較をするために，オリジナルの Java と比較する．破壊的クラス拡張の影響が少ないことを確認する．

6.1 セマンティクスに沿った実装と最適化手法による実装

セマンティクスに沿った実装と最適化手法による実装を比較するために，マイクロベンチマークによる実験を行った．セマンティクスに沿った実装は，実行時の処理部分だけを Java の ReflectionAPI を利用して再現した．reviser を実行前に手動で通常の Java プログラムに変換し，Reflection API を用いた実行時システムが実行時にセマンティクス通りの mbody メソッドによるメソッド探索を実行するようにした．

実験は 8 種類あり，include の深さを増やしていくもの (図 6.1 中の a) を 2 種，include の幅

を増やしていくもの (図 6.1 中の b) を 2 種，link の深さを増やしていくもの (図 6.1 中の c)，link の幅を増やしていくもの (図 6.1 中の d) を 2 種，ある methodshell が所属している shell グループを増やしていくもの (図 6.1 中の e) の 8 種である．図 6.1 中の各四角は methodshell を表しており，実線の矢印が include 関係を，点線の矢印が link 関係を表している．それぞれの実験では，root と書かれている methodshell を選択してその main メソッドを実行するとする．また，図中の n を 10 から 300 まで 10 ずつ増やして，それぞれでメソッド呼び出しの実行時間の計測を行った．

各実験について詳細を説明する．

- a の実験については, root の中のメソッドを呼び出す (a2), root から最も遠い methodshell 内のメソッドを呼び出す (a1) の 2 種の実行時間を計測した．図 6.2 が (a1) と (a2) の詳細である．四角が methodshell を表しており，四角の中に書かれているコードがその methodshell のコードを表す．また実線の矢印が include 関係を表しており，点線の矢印が link 関係を表している．矢印の向きと include および link 関係に関しては，図 6.1 を参照してほしい．n 個の methodshell を用意し，ShellN で Shell(N-1) を include する．各 methodshell は Example クラスの foo メソッドを再定義し，さらに root と書かれている methodshell から最も遠い methodshell は bar メソッドも再定義する．root と書かれている methodshell から bar メソッドを呼び出すのを実験 (a1) とし，foo メソッドを呼び出すのを実験 (a2) とした．
- b の実験については，一番最初に include した methodshell 内のメソッドを呼び出す (b1)，一番最後に include した methodshell で内のメソッドを呼び出す (b2) の 2 種の実行時間を計測した．図 6.3 が (b1) と (b2) の詳細である．root と書かれている methodshell が n 個の methodshll を include するる．そのうち，一番最初に include している methodshell では，Example クラスの foo メソッドを再定義する．一番最後に include している methodshell では，Example クラスの bar メソッドを再定義する．root と書かれている methodshell から foo メソッドを呼び出すのを実験 (b1)，bar メソッドを呼び出すのを実験 (b2) とした．
- c の実験の詳細を図 6.4 に示す．n 個の methodshell を用意し，ShellN で Shell(N-1) を link する．各 methodshell は Example クラスの foo メソッドを再定義する．root と書かれている methodshell で foo メソッドを呼び出すのを実験 (c) とする．
- d の実験については，一番最初に link した methodshell 内のメソッドを呼び出す (d1)，一番最後に link した methodshell 内 (d2) のメソッドを呼び出す，の 2 種の実行時間を計測した．図 6.5 が (d1) と (d2) の詳細である．root と書かれている methodshell が n 個の methodshell を link する．そのうち，一番最初に link している methodshell では，Example クラスの foo メソッドを再定義する．一番最後に link している methodshell では，Example クラスの bar メソッドを再定義する．root と書かれている methodshell から foo メソッドを呼び出すのを実験 (d1)，bar メソッドを呼び出すのを実験 (d2) とする．

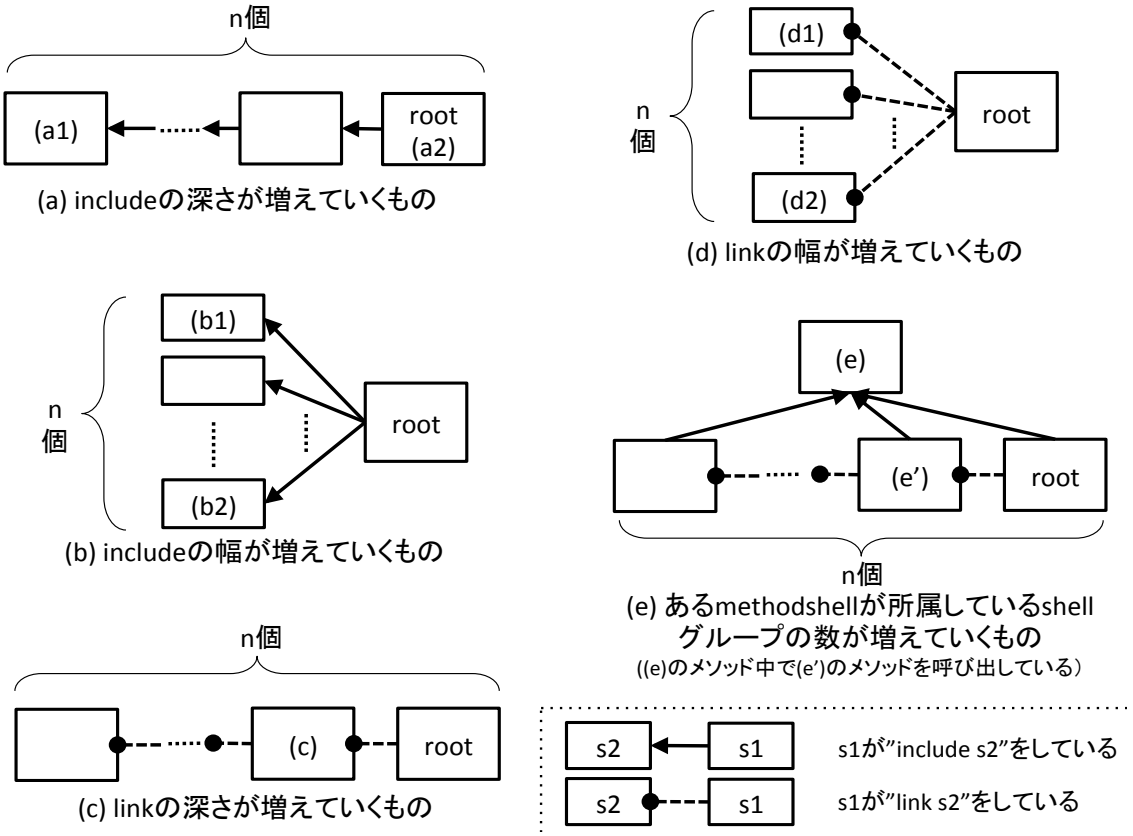

図 6.1. 実験内容：各四角が methodshell を表している．四角の中に書かれているコードがその methodshell のコードを表す．実線の矢印が include 関係，点線の矢印が link 関係を表している．

- e の実験はやや複雑で図 6.1 の root と書かれている methodshell から (e) と書かれた methodshell 内のメソッドを呼び出し，そのメソッド内から (e') と書かれた methodshell 内のメソッドを呼び出す，という二回のメソッド呼び出しにかかる時間を計測した．これはある methodshell S が多くの shell グループに所属しているとき，S からメソッドを呼び出すのにかかる時間を計測するためである．e の詳細を図 6.6 に示す．n 個の methodshell があり，ShellN は Shell(N-1) を link する．さらに，各 ShellN は Shell という名前の methodshell をそれぞれ include する．つまり，Shell という名前の methodshell は Shell0 グループから ShellN グループまでの n 個のグループに所属することになる．各 Shelli グループは，Shelli methodshell と Shell methodshell の 2 つの methodshell からなる．Shell methodshell 以外の methodshell は，それぞれ Example クラスの foo メソッドを再定義する．Shell methodshell は Example クラスの foo メソッドと bar メソッドを再定義する．このとき，bar メソッド中では foo メソッドを呼び出している．実験 (e) では，root と書かれている methodshell から bar メソッドを呼び出したときの実行時間を測定する．

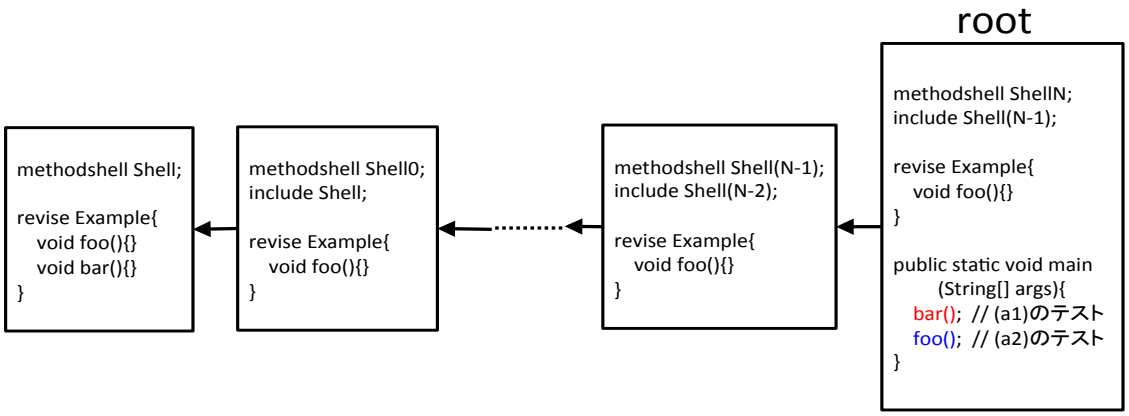

図 6.2. (a1)，(a2) の実験のコード．(a1) の実験では，root と書かれたから最も遠い methodshell で定義されている bar メソッドを呼び出す．(a2) の実験では，root 中で定義されている foo メソッドを呼び出す．

実験結果は図 6.7~6.14 の 8 つのグラフのようになっている．これは，各実験のメソッド呼び出し 10000 回にかかる時間を 10 回計測し，その平均値および標準偏差を計算しグラフ化したものである．水色のひし形が ReflectionAPI による実装を用いた実行時間で，緑の三角が 4 章の最適化手法による実装を用いた実行時間である．縦軸が実行時間を表しており，横軸が methodshell 数 (図 6.1 中の n の値) を表しているが，(e) の実験のみ，横軸はコンテキスト数 (図 6.1 中の (e)n の値) を表している．

(a1)，(b1)，(b2)，(d1)，(d2) では ReflectionAPI を用いた実装では，methodshell 数が増えるに従って線形に実行時間が増えているのに対し，最適化手法による実装では実行時間の増加は見られない．ゆえに，methodshell 数増加に伴う実行時間の増加を実験の範囲では抑えられていることが確認できた．

これは，ReflectionAPI を用いたときには全ての methodshell を順に探索しなければならない場合があるが，最適化手法ではメソッド呼び出しが一意に定まるためである．(a1) の場合には，セマンティクスでは include している methodshell をたどりながら，それぞれの methodshell を順に探索する．(a1) の実験では，一番 root から離れた methodshell にあるメソッドを呼び出すため，methodshell 数が増えればそれだけ実行時間が増加する．(b1)(b2) のように 1 つの methodshell が複数の methodshell を include している場合には，セマンティクスは include し

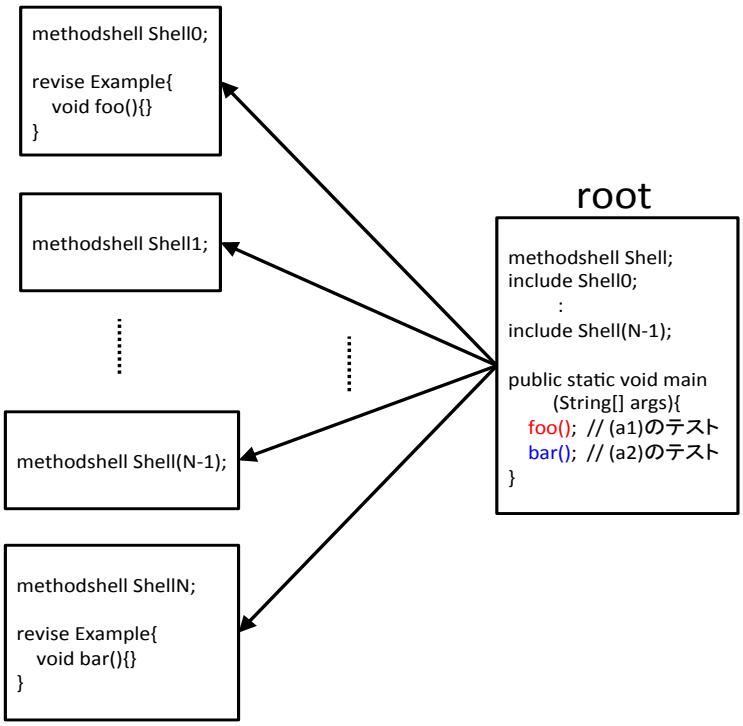

図 6.3. (b1)，(b2) の実験のコード．(b1) の実験では，root と書かれた methodshell が一番最初に include している methodshell で定義されている foo メソッドを呼び出す．(b2) の実験では，一番最後に include している methodshell で定義されている bar メソッドを呼び出す．

ている methodshell 全てを探索する必要があるため，methodshell 数が増えれば実行時間が増加する．(d1)(d2) も同様に，1 つの methodshell が複数の methodshell を link している場合には，セマンティクスでは link している methodshell 全てを探索する必要があるため，methodshell 数が増えれば実行時間が増加する．4 章の実装方法を用いれば，いずれの場合にもコンパイル時に呼び出すメソッドが一意に決定されるため，methodshell 数の増加に伴う実行時間の増加は抑えられる．

6.2　GluonJ，オリジナルの Java との比較

この章で行う実験で，Method Shell のメソッド探索について，

- include 宣言による影響が少ないこと
- 破壊的クラス拡張による影響が少ないこと

の 2 点を確認した．

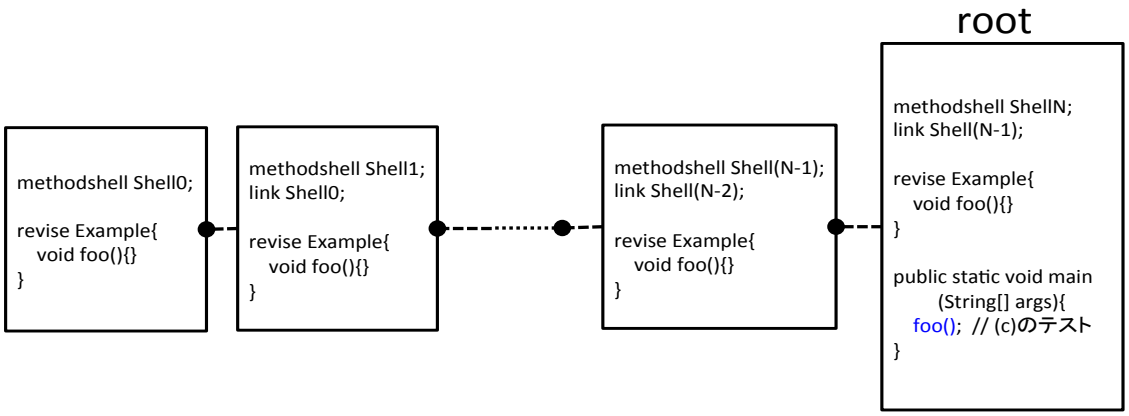

図 6.4. (c) の実験の実験のコード

本研究では，4 章で提案した最適化手法による Method Shells の実装と，GluonJ を用いた実装およびオリジナルの Java との比較実験を行った．前者の GluonJ による実験は，4 章の最適化手法による Method Shells の include の実装が破壊的クラス拡張を導入している他の言語と変わらぬ実行速度であることを示すために行われる．GluonJ は破壊的クラス拡張の一種である reviser を導入している言語で，include 宣言に似た機能を有している．後者のオリジナルの Java を用いた実験では，4 章の最適化手法による Method Shells の実装が破壊的クラス拡張を用いずに元のソースコードを直接書き換えた場合と変わらぬ実行速度であることを示すために行われる．

前述した 8 つの実験のうち，include に関する 4 つの実験 ((a1)，(a2)，(b1)，(b2)) に関して GluonJ とオリジナルの Java で最適化手法との比較実験を行った．実験環境は，前述したものと同じで，CentOS release 6.2，Intel(R) Xeon(R) CPU E5-2687W 3.10GHz，メモリは 64GB，OpenJDK 1.7.0 である．GluonJ については GluonJ 2.3[39] を用いた．

実験結果は図 6.15~6.18 のようになった．これは，各実験のメソッド呼び出し 10000 回にかかる時間を 10 回計測し，その平均値および標準偏差を計算しグラフ化したものである．緑の三角が 4 章の最適化手法による実装を用いた実行時間，赤い四角が GluonJ による実装の実行時間，紫の丸がオリジナルの Java の実行時間を表している．グラフの縦軸が実行時間を表し，横軸が methodshell の数 (図 6.1 中の n の値) を表している．

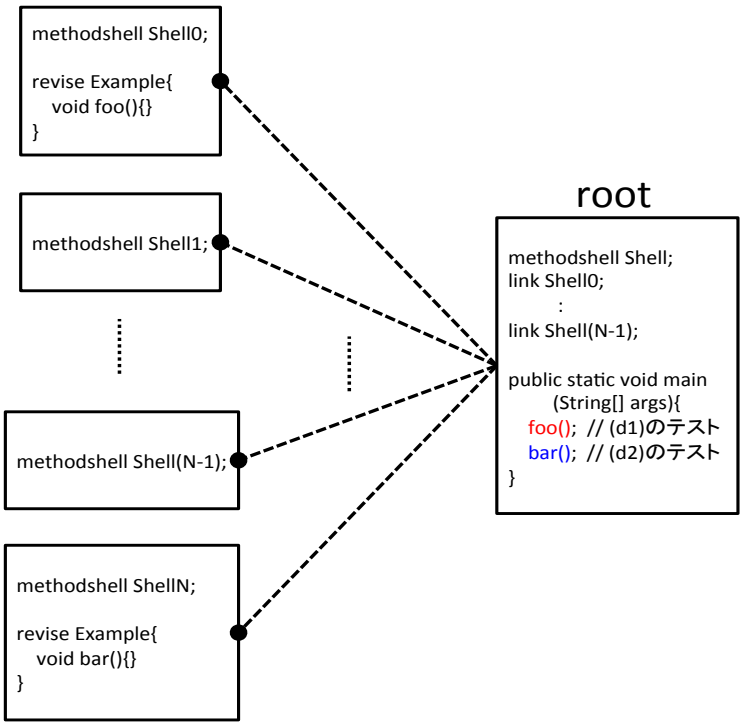

図 6.5. (d1)，(d2) の実験のコード．(d1) の実験では，root と書かれた methodshell が一番最初に link している methodshell で定義されている foo メソッドを呼び出す．(d2) の実験では，一番最後に link している methodshell で定義されている bar メソッドを呼び出す．

4 章の最適化手法，GluonJ，オリジナルの Java，共に実行時間に大きな差は見られない．これにより，実験の範囲では include による実行時間の増加や破壊的クラス拡張による実行時間の増加がほとんどないことが確認できた．

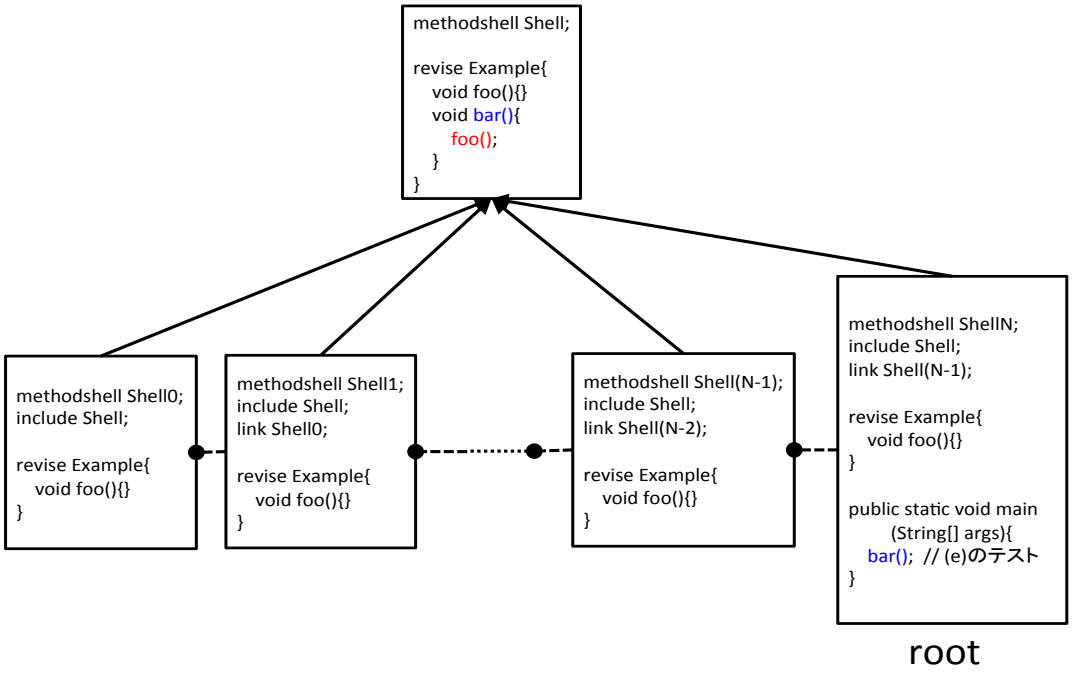

図 6.6. (e) の実験のコード．Shell methodshell は n 個の shell グループに所属している．

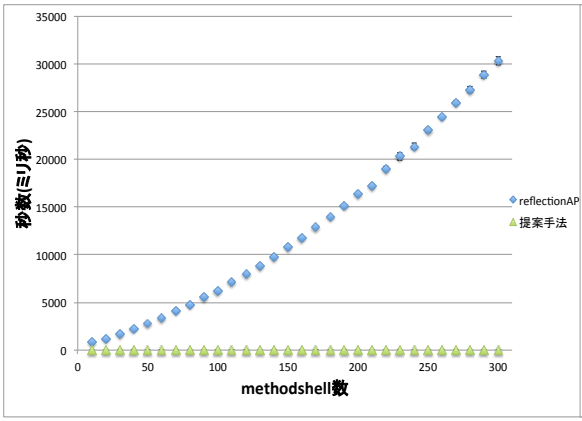

図 6.7. (a1) の実験結果

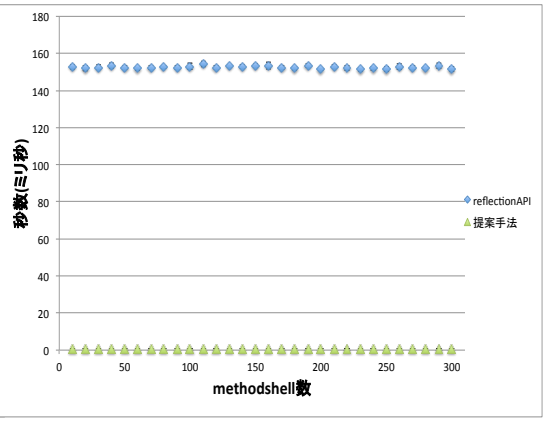

図 6.8. (a2) の実験結果

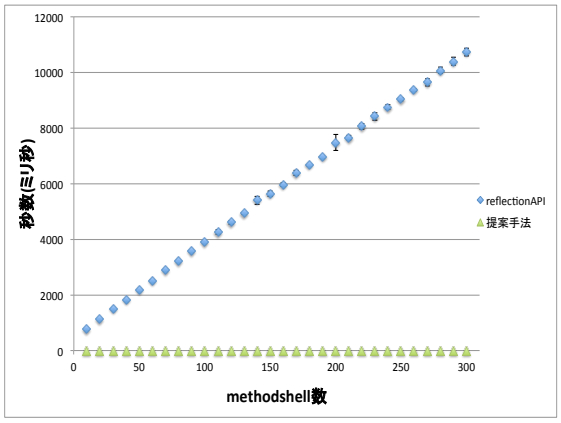

図 6.9. (b1) の実験結果

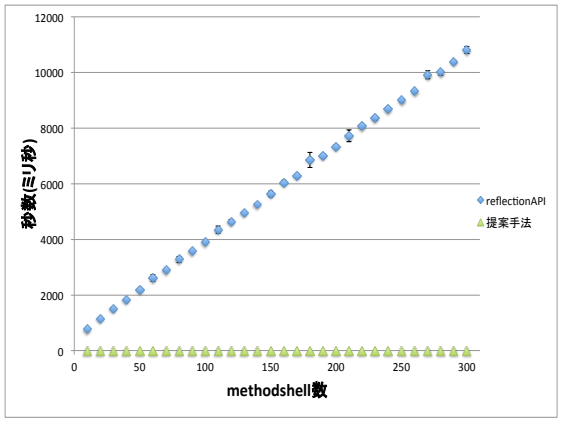

図 6.10. (b2) の実験結果

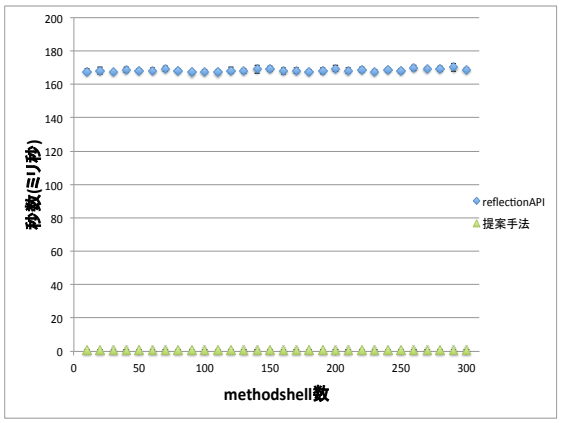

図 6.11. (c) の実験結果

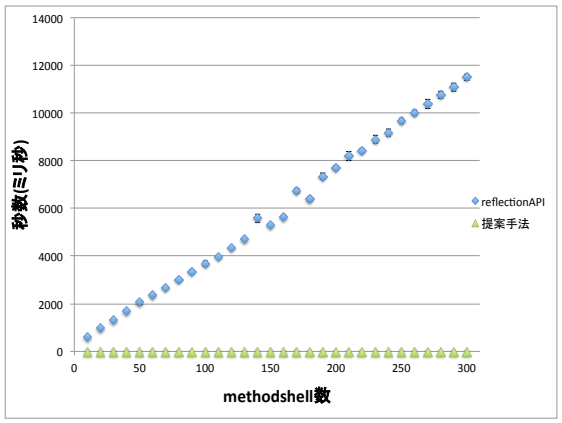

図 6.12. (d1) の実験結果

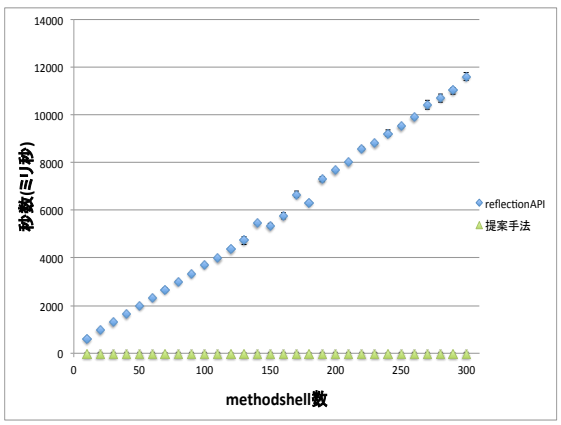

図 6.13. (d2) の実験結果

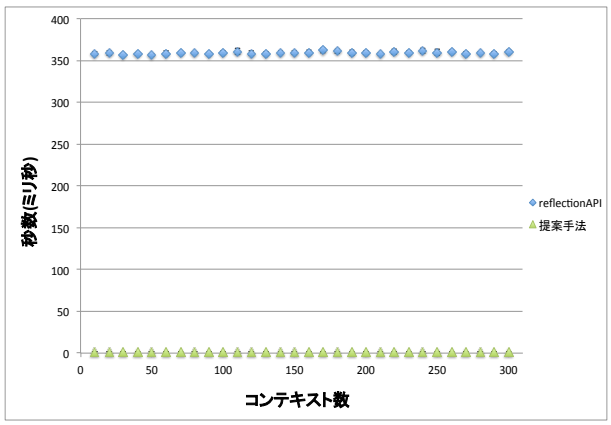

図 6.14. (e) の実験結果

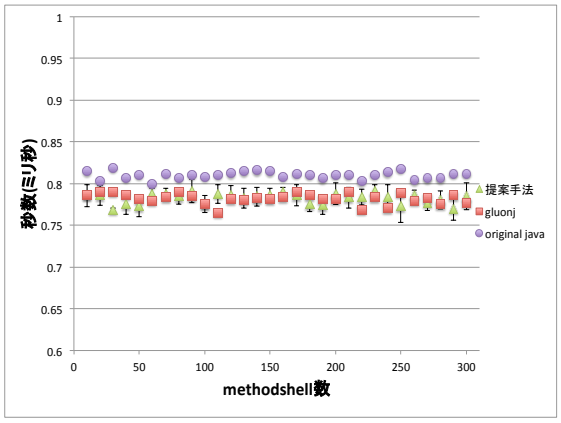

図 6.15. (a1) の実験結果

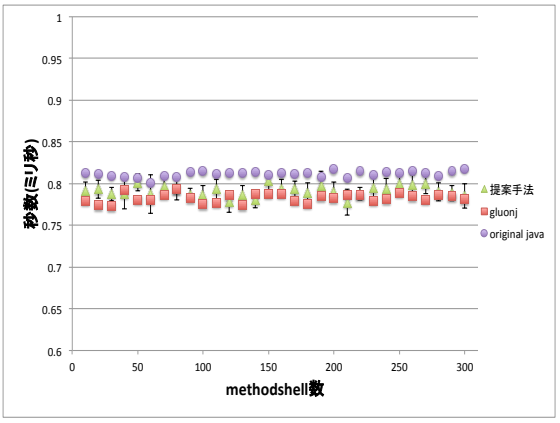

図 6.16. (a2) の実験結果

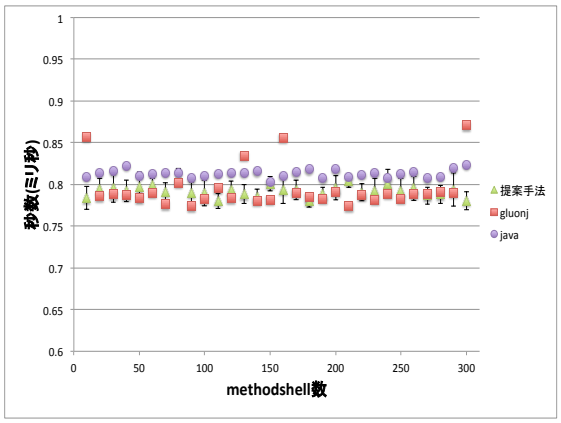

図 6.17. (b1) の実験結果

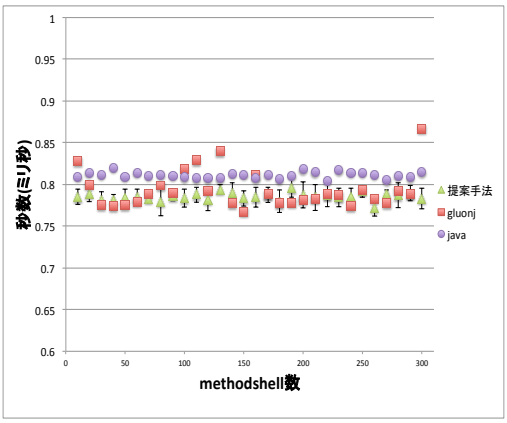

図 6.18. (b2) の実験結果

第 7 章

まとめ

既存のコードを書き換えることなく，別ファイルに差分のみを記述するだけで，メソッドの追加や再定義を行うメカニズムは数多く存在し，様々な言語に実装されている．我々はこのようなメカニズムを破壊的クラス拡張と呼んでいる．破壊的クラス拡張を用いると，そのままであれば再利用できなかったコードを再利用できるため，コードの再利用を促進し開発効率をあげることができる．また，変更部分のみを別ファイルにモジュール化できるため，変更にミスがあった場合にも別ファイルを削除するだけで変更前のコードを取得することができる．

破壊的クラス拡張には利点が多い反面，メソッドの衝突という問題がある．メソッドの衝突とは同名のメソッドの再定義が複数存在してしまうことを指す．破壊的クラス拡張を多用すると，複数の破壊的クラス拡張が同じメソッドを再定義してしまい，プログラマが望む通りにプログラムが動作しない可能性ある．

本研究では，有効にするメソッドをグループ化し，そのグループを実行時に切り替えることによって，破壊的クラス拡張によるメソッドの衝突を避ける事ができるモジュール機構 Method Shells を提案した．Method Shells は二段階のグループ化を用意している．一段階目がモジュール methodshell によるグループ化で，クラス宣言と破壊的クラス拡張をまとめることができる．二段階目が shell グループによるグループ化で，一度に有効化したい複数の methodshell をグループ化することができる．Method Shells はこの shell グループを実行時に切り替えることによって，有効なメソッドを切り替えることを可能にした．Method Shells では methodshell 中に link 宣言と include 宣言を記述することができる．この 2 つの宣言によって，グループ化や shell グループの切り替えをコントロールすることができる．

本研究では，Method Shells のセマンティクスを定義し，メソッド呼び出しの最適化を行った実装方法を提案した．セマンティクスそのままに実装してしまうと，実行時にメソッド探索を行うため，methodshell の数が増えるにつれてメソッド探索の候補が増え，実行時間が遅くなってしまう．本研究が提案している実装方法では，分割コンパイルを諦める代わりに，コンパイル時にメソッドをコピーしメソッド名を変えることで，メソッド呼び出しをコンパイル時に一意に定めている．これによって，methodshell 増加に伴う実行時間の増加を抑えることができると考えた．また，実際に JastAddJ という名前の Java コンパイラを拡張することで，Method Shells のコンパイラのプロトタイプを作成した．

本研究では，さらにこの実装方法の効果を測定するためにいくつかの実験を行った．まずセマンティクスに準じた実装を Java の ReflectionAPI を用いて実装し，比較を行った．結果，セマンティクスに準じた実装では線形に実行時間が増えるところを，最適化手法では抑えられることを確認した．また，GluonJ とオリジナルの Java との実行時間の比較も行った．GluonJ は破壊的クラス拡張を導入している言語であり，include 宣言に似た機能を導入している．. GluonJ と比較することにより，同等の実行速度を実験の範囲では出せることを確認し，include 宣言の影響が小さいことを確かめた．オリジナルの Java との比較により，破壊的クラス拡張を用いずにコードを書き換えた場合とも同等の実行速度を実験の範囲では出せることを確認し，破壊的クラス拡張の影響が小さいことを確かめた．

今後の課題は，本モジュール機構の有用性を示すことである．既存のフレームワークやライブラリなどの実用的なプログラムを，本モジュール機構で実装することが可能かを検証する必要がある．

発表文献と研究活動

(1) 竹下若菜，赤井駿平，千葉滋．破壊的クラス拡張のスコープを制限するモジュール機構の意味論およびその実装方法．日本ソフトウェア科学会第 29 回大会, 2012.08.22-24.

(2) 竹下若菜，千葉滋．Method Shells: Avoiding Conflicts on Destructive Class Extensions by Implicit Context Switches．In proceedings of the 13th international conference on Software Composition (SC'13), LNCS 8088, Springer, pp. 49-64, 2013.

(3) 竹下若菜，千葉滋．破壊的クラス拡張で生じるメソッド衝突を回避可能なモジュール機構 Method Shells とその実装方法，情報処理学会論文誌 プログラミング，vol.7

参考文献

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pages 220–242. Springer, 1997.

[2] Ruby programming language. http://www.ruby-lang.org/.

[3] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 539–554, New York, NY, USA, 2010. ACM.

[4] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(6):2004, 2004.

[5] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.

[6] Sean Mcdirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 211–222. ACM Press, 2001.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[8] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 224–240, New York, NY, USA, 2003. ACM.

[9] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 39–50, New York, NY, USA, 2009. ACM.

[10] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 51–62, New York, NY, USA, 2009. ACM.

[11] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 173–188, London, UK, UK, 2002. Springer-Verlag.

[12] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 141–150, New York, NY, USA, 2004. ACM.

[13] Fuminobu Takeyama and Shigeru Chiba. An advice for advice composition in aspectj. In *Proceedings of the 9th international conference on Software Composition*, SC'10, pages 122–137, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 144–168, Berlin, Heidelberg, 2005. Springer-Verlag.

[15] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to aspectj. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, AOSD '06, pages 39–50, New York, NY, USA, 2006. ACM.

[16] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Objct-Oriented Programming*, ECCOP '98, pages 186–211, London, UK, UK, 1998. Springer-Verlag.

[17] Todd Millstein. Practical predicate dispatch. In *Proc. of ACM OOPSLA*, pages 345–364. ACM, 2004.

[18] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the beta programming language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 285–298, New York, NY, USA, 1983. ACM.

[19] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM.

[20] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] The newspeak programming language specification. http://bracha.org/newspeak-spec.pdf.

[22] Alexandre Bergel, Stphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. In *Computer Languages, Systems and Structures*, 2005.

[23] Alexandre Bergel. Classbox/j: Controlling the scope of change in java. In *In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189. ACM Press, 2005.

[24] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99—Object-Oriented Programming*, pages 279–303. Springer, 1999.

[25] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. Worlds: Controlling the scope of side effects. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 179–203, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] R. B. Smith, David Ungar, Randall B. Smith, and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2:161–178, 1996.

[27] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[28] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[29] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[30] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent java applications with contextj. In *International Workshop on Context-Oriented Programming*, COP '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM.

[31] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. Generative and transformational techniques in software engineering ii. chapter An Introduction to Context-Oriented Programming with ContextS, pages 396–407. Springer-Verlag, Berlin, Heidelberg, 2008.

[32] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2169–2175, New York, NY, USA, 2010. ACM.

[33] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, NY, USA, 2007. ACM.

[34] Shumpei Akai and Shigeru Chiba. Method shelters: avoiding conflicts among class extensions caused by local rebinding. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 131–142, New York, NY, USA, 2012. ACM.

[35] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal

core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[36] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[37] jastadd. http://jastadd.org/web/.

[38] Görel Hedin and Eva Magnusson. Jastadd: An aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, April 2003.

[39] Gluonj. http://www.csg.ci.i.u-tokyo.ac.jp/projects/gluonj/.

謝辞

本研究を進めるにあたって，研究方針や論文構成など様々な助言およびご指導をいただいた指導教員の千葉滋教授に心から感謝致します．論文や発表方法などを熱心にご指導いただいたおかげで，国内だけでなく海外の学会にも参加することができました．多くの経験を積むことができ，私の大きな糧になりました．

また，本モジュール機構の仕様や研究の進め方，論文の書き方など多岐にわたって，様々な助言を赤井駿平氏にいただきました．深く感謝致します．

最後に共に研究し，数々の助言を頂いた千葉研究室の皆様方に心より感謝致します．