情報処理学会論文誌　プログラミング　Vol.7　No.3　12–21 (July 2014)

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

竹下 若菜[1,a)]　千葉 滋[1,b)]

受付日 2013年11月28日, 採録日 2014年3月19日

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こと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．Method Shells は，1 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．

キーワード：モジュール機構，Java，破壊的クラス拡張

The Semantics and Implementation of Method Shells: Avoiding Method Conflicts Caused by Destructive Class Extensions

Wakana Takeshita[1,a)]　Shigeru Chiba[1,b)]

Received: November 28, 2013, Accepted: March 19, 2014

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.

Keywords: module system, Java, destructive class extension

1. はじめに

現在，アプリケーションを 1 からすべてそのプログラマが作るということは珍しくなっている．多くの場合，既存

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] takeshita@csg.ci.i.u-tokyo.ac.jp
[b)] chiba@acm.org

© 2014 Information Processing Society of Japan

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を追加して開発する．このようなコードの再利用は開発の効率を上げ，生産性を上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．

機能を変更をする際, 既存のコードを書き換えずに別ファイルに差分のみを記述することで機能を変更できると便利なことが多い．変更部分をモジュール化でき, 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．このような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）[12] 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 [1]，GluonJ [6]，AHEAD [3]，MultiJava [14]，Jiazzi [13] 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．破壊的クラス拡張はコードの再利用を促進する反面，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，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，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．メソッドの衝突が起きると，プログラマが意図しない動作をプログラムがする可能性がある．

我々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 は有効にするメソッドを実行時に切り替えることで，メソッドの衝突を回避する．メソッドの切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．

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も行い，その効果を実験により示す．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．

以降，2 章では破壊的クラス拡張とその問題点を例を交えて提示し，3 章では我々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 章ではその提案した実装方法の効果を測定した結果を提示する．6 章では関連研究について論じ，7 章では本論文についてのまとめを述べる．

2. 破壊的クラス拡張とその問題点

破壊的クラス拡張はコードの再利用性を高めるが，メソッドの衝突を引き起こす危険性がある．メソッド衝突は破壊的クラス拡張の有名な問題点である [2], [7], [8]．この章では，破壊的クラス拡張とその問題点を例とともに示す．

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，さらにいくつかの機能の追加を行いたいとする．

図 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，破壊的クラス拡張として GluonJ の reviser を用いている．**図 2** は HTML-renderer ライブラリ中の WebPage クラスである．図 1 の reviser は図 2 の 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 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 1, 2 のように書き，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 のコード中の setPopupItem メソッドは図 1 中の setPopupItem メソッドに置き換わり，さらに図 1 中の onClick メソッドは WebPage クラスに追加される．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，つねにアラートメッセージを出すようになる．

```
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 1　Reviser の例
Fig. 1　An example of reviser.

```
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　WebPage クラス
Fig. 2　WebPage class.

© 2014 Information Processing Society of Japan

```
//in the viewer library
revise WebPage{
  void setPopupItem(Popup p, HTML text){
    //show red and bold text
  }
}
class Viewer{
  void check(file f){
    if(isConfidential(f))
      new WebPage().popup("confidential");
  }
}
```

図 3　Viewer ライブラリ
Fig. 3　Viewer library.

```
//in the application library
class App{
  void view(WebPage w){
    new Viewer().check(new File("test.txt"));
  }
  public static void main(String[] args){
    WebPage w = new WebPage()
    new App().view(w); // want to viewing red and bold text
    w.onClick(getMouse()); // want to prevent popup window
  }
}
```

図 4　Browser と viewer の両方を使いたい
Fig. 4　Using both browser and viewer.

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．

次に 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（図 **3**）．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．viewer ライブラリと WebPage クラスのソースコードをコンパイラに与えることで，望みどおりの変更ができる．

この 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，一緒に使うとうまく働かない危険性がある（図 **4**）．コンパイラに与えるソースコードとして browser ライブラリと viewer ライブラリの両方を選択してしまうと，WebPage クラス中の setPopupItem メソッドの定義および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 メソッドが呼び出されてほしいところで，viewer ライブラリの setPopupItem メソッドが意図せず呼び出されてしまう可能性がある．たとえば，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とおりの動作をしないおそれがある．

3. 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，我々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．

我々は 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 の文法を用いており，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 を複数記述し，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 グループをコントロールする．

3.1　モジュール method shell

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 **5** 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

© 2014 Information Processing Society of Japan

```
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 5　Renderer methodshell
Fig. 5　Renderer methodshell.

を 1 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，実行することでプログラムは実行される．renderer methodshell を選択すると，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．

3.2　methodshell の拡張と shell グループ

include 宣言を用いると，methodshell を拡張できる．図 6 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 6 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 が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 は推移的であるため，include を繰り返すことで，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メソッドを増やしたりできる．また，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 7 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 からなる methodshell の集合を指す．

```
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 6　Browser methodshell
Fig. 6　Browser methodshell.

```
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 7　Viewer methodshell
Fig. 7　Viewer methodshell.

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．たとえば，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．

3.3　shell グループの切替え

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．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，そのメソッドを呼び出している間だけ，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．切

© 2014 Information Processing Society of Japan

```
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 8　App-for-viewer methodshell
Fig. 8　App-for-viewer methodshell.

り替わると，新たに有効になった shell グループ中のクラス定義と破壊的クラス拡張が有効になり，すべてのオブジェクト中のメソッド定義がそのクラス定義や破壊的クラス拡張中のメソッド定義に切り替わる．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，意図したとおりの動作が可能となる．

図 8 の app-for-viewer methodshell は viewer shell グループを link している．カレントグループが app-for-viewer methodshell 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とおりのカレントグループの切替えが可能となる．

3.4　利用例

2 章の図 4 のコードで実現したかったことは methodshell を用いると図 5～**図 9** のように実現できる．**図 10** は図 5～図 9 までの 4 つの methodshell の include，link 関係を図示している．

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．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 グループなので，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いる間，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．

```
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 9　App methodshell
Fig. 9　App methodshell.

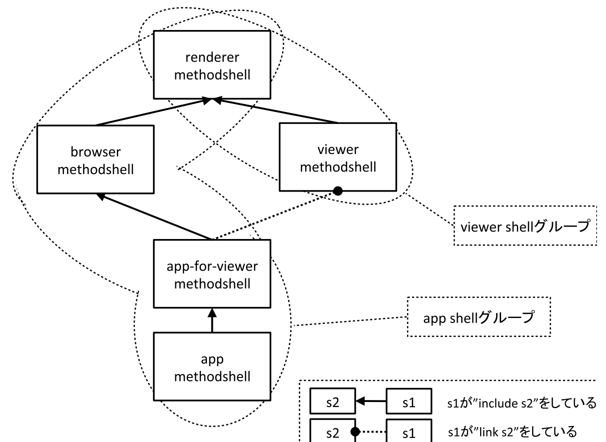

図 10　図 5～図 9 の methodshell の include，link 関係
Fig. 10　The include and link relations in Fig. 5 to 9.

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 8 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．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

© 2014 Information Processing Society of Japan

のテキストを表示するようになる．view 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 グループに戻される．

3.5 セマンティクス

この節では Method Shells の形式的な意味論を示す．

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．Method Shells の形式化は Featherweight Java（FJ）[11] と GluonFJ の拡張によって表現される．文法は次のようになっている．

$\mathsf{SL}::=\mathsf{methodshell}\ \mathsf{S};\overline{\mathsf{IL}}\ \overline{\mathsf{LL}}\ (\mathsf{CL}\ \|\|\ \mathsf{RV})*\mathsf{MF}$	method-shell declaration
$\mathsf{IL}::=\mathsf{include}\ \mathsf{S};$	include declaration
$\mathsf{LL}::=\mathsf{link}\ \mathsf{S};$	link declaration
$\mathsf{CL}::=\mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{C}\{\overline{\mathsf{C}}\ \overline{\mathsf{f}};\mathsf{K}\ \overline{\mathsf{M}}\}$	class declaration
$\mathsf{RV}::=\mathsf{revise}\ \mathsf{C}\{\overline{\mathsf{M}}\}$	reviser declaration
$\mathsf{M}::=\mathsf{C}\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\{\mathsf{return}\ \mathsf{e};\}$	method declaration
$\mathsf{MF}::=\mathsf{void}\ \mathsf{main}()\{\mathsf{return}\ \mathsf{e};\}$	method declaration
$\mathsf{e}::=\mathsf{x}\ \|\ \mathsf{e.f}\ \|\ \mathsf{e.m}(\overline{\mathsf{e}})\ \|\ \mathsf{new}\ \mathsf{C}(\overline{\mathsf{e}})\ \|\ \mathsf{e}\ \mathsf{in}\ \mathsf{S};\mathsf{S_C}$	expressions
$\mathsf{v}::=\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})$	values

メタ変数について説明する．S，T は methodshell の名前である．B，C，D 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；K はコンストラクタ宣言である．v，w は値である．K と body の文法はここでは示さないが，FJ の文法に従う．列はオーバラインによって表す．たとえば，$\overline{\mathsf{x}}$ は "$\mathsf{x_1, x_2, \cdots, x_n}$" を表し，$\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ は "$\mathsf{C_1\ f_1, C_2\ f_2, \cdots, C_n\ f_n}$" を表す．

SL は methodshell の宣言であり，methodshell 名と include 宣言と link 宣言，クラス宣言，reviser 宣言，main 関数を含む．CL はクラス宣言である，クラス名とスーパクラス，フィールド，コンストラクタ，メソッドを含む．RV は reviser 宣言であり，reviser 名とメソッドを含む．reviser はフィールドを持たない．e は式であり，$\mathsf{e\ in\ S; S_C}$ という新しい形を導入している．この形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 $\mathsf{S_C}$ 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．プログラムは CRT と methodshell 名のペアである．プログラム実行はその methodshell 中の main 関数から始まる．

メソッド呼び出しのセマンティクス

簡約は $\mathsf{S; S_C \vdash e \rightarrow e'}$ 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ときに，methodshell S が選択されたとすると，式 e は methodhell S の中で簡約され，カレントグループは S になる．ほとんどの簡約規則は FJ と GluonFJ のものに従うため，methodshell 特有の簡約規則のみ下に示す．

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

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

最初の規則では，$\mathsf{e_0}$ は $\mathsf{S; S_C}$ が与えられているところでも in によって $\mathsf{T; T_C}$ 中で簡約されることを示している．2 つ目の規則は，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．2 つ目の規則は，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．

mbody の式のセマンティクスを**図 11** に示す．$mbody(\mathsf{m, C, S, S_C})$ は 4 つの引数からメソッド探索を行い，$\overline{\mathsf{x}}.\mathsf{e\ in\ T; T_C}$ 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．$linked\text{-}shells(\mathsf{S})$ は methodshell S が link している methodshell の集合を返す．$include\text{-}shells(\mathsf{S})$ は methodshell S が include している methodshell の集合を返す．$mbodyshell(\mathsf{m, C, S})$ 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．$mbodyglobal(\mathsf{m, C, S, S_C})$ 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すべての methodshell から include されている特殊な methodshell で，特定の methodshell 内に含まれていないクラス宣言や reviser はすべてグローバルなグループに所属していると考える．図 11 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 のスーパクラスを探索するために *mbody* を再帰的に呼び出す．

$mbody(\mathsf{m, C, S, S_C})$ は次のような順序で探索を行う．まず，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

© 2014 Information Processing Society of Japan

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

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

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

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

$$\frac{\begin{array}{c}\mathtt{CRT}(\mathtt{S},\mathtt{C}) = (\mathtt{revise}\ \mathtt{C}\{\overline{\mathtt{M}}\}) \mid\mid (\mathtt{class}\ \mathtt{C} \cdots \{\cdots \overline{\mathtt{M}}\}) \\ \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbodyshell}(\mathtt{m},\mathtt{C},\mathtt{S}) = \bar{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{S}}$$

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

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

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

図 11 メソッド探索のセマンティクス

Fig. 11 The semantics of method lookup.

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，C のスーパクラス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．

3.6 制限

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．

また，methodshell 中には include 宣言と link 宣言を複数書くことができるが，同じメソッドが include された複数の methodshell 内で定義されている，あるいは link された複数の methodshell 内で定義されているとエラーになる．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とき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，プログラマの意図しないメソッド衝突を避けることにつながる．また，これらの制限により，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．

4. 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．我々は 3.5 節で 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 の増加にともない，実行時間が増加してしまう場合がある．たとえば，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 すべての探索を行う必要がある．

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 $\mathsf{m}_{\mathsf{S_i}}$ を $\mathsf{m}_{\mathsf{S_i}\cdot\mathsf{T_1}}, \mathsf{m}_{\mathsf{S_i}\cdot\mathsf{T_2}}, \mathsf{m}_{\mathsf{S_i}\cdot\mathsf{T_3}}, \cdots$ のように複製し，$\mathsf{T_j}$ がカレントグループであるときに $\mathsf{m}_{\mathsf{S_i}\cdot\mathsf{T_j}}$ を用いるようにする．メソッドボディの方も $\mathsf{S_i}$ と $\mathsf{T_j}$ に合わせた形に書き換える．型環境 Γ の下で methodshell が S でカレントグループが $\mathsf{S_C}$ のときの変換規則を $\mathsf{S};\mathsf{S_C};\Gamma \vdash \mathsf{e} \Rightarrow \mathsf{e}'$ 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次のような変換を行う．

$$\frac{\mathtt{S};\mathtt{S_C};\Gamma\vdash\mathtt{e_0}\Longrightarrow\mathtt{e_0}' \quad \mathtt{S};\mathtt{S_C};\Gamma\vdash\bar{\mathtt{e}}\Longrightarrow\bar{\mathtt{e}}' \quad \mathtt{S};\mathtt{S_C};\Gamma\vdash\mathtt{e_0}:\mathtt{C} \quad \mathit{mname}(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\mathtt{m}'}{\mathtt{S};\mathtt{S_C};\Gamma\vdash\mathtt{e_0}.\mathtt{m}(\bar{\mathtt{e}})\Longrightarrow\mathtt{e_0}'.\mathtt{m}'(\bar{\mathtt{e}}')}$$

$\mathsf{S};\mathsf{S_C};\Gamma\vdash \mathsf{e_0} : \mathsf{C}$ は「式 $\mathsf{e_0}$ 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，下のように定義される：

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\bar{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{null};\mathtt{null}}{\mathit{mname}(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\mathtt{m}}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\bar{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{T};\mathtt{T_C}}{\mathit{mname}(\mathtt{m},\mathtt{C},\mathtt{S},\mathtt{S_C})=\mathtt{m}_{\mathtt{T}\cdot\mathtt{T_C}}}$$

グローバルなグループからメソッドが選ばれるとき，呼

© 2014 Information Processing Society of Japan

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 節のセマンティクスどおりに実装すると，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．

5. 実験

本章では，3.5 節で示したセマンティクスどおりの実装と 4 章の提案手法による実装との実行時間の比較実験を行った．実験環境は，CentOS release 6.2，Intel® Xeon® CPU E5-2687W 3.10 GHz，メモリは 64 GB，OpenJDK 1.7.0 である．提案手法の実装は，JastAddJ [9] を用いて実装した．JastAddJ は JastAdd というツールを用いて作られた Java コンパイラであり，拡張が容易である．我々は JastAddJ を拡張し，3.5 節で定義した文法，4 章で定義した実装方法を導入した Method Shells のコンパイラを実装し，実験に用いた．

セマンティクスどおりの実装と提案手法による実装

セマンティクスどおりの実装と提案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．セマンティクスどおりの実装は，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どおりの 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 12 中の a）を 2 種，include の幅を増やしていくもの（図 12 中の b）を 2 種，link の深さを増やしていくもの（図 12 中の c），link の幅を増やしていくもの（図 12 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 12 中の e）の 8 種である．それぞれ 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し，図中の n を 10 から 300 まで 10 ずつ増やして実験を行った．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．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．c も同様に，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．e についてはやや複雑で root 内から (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 2 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．

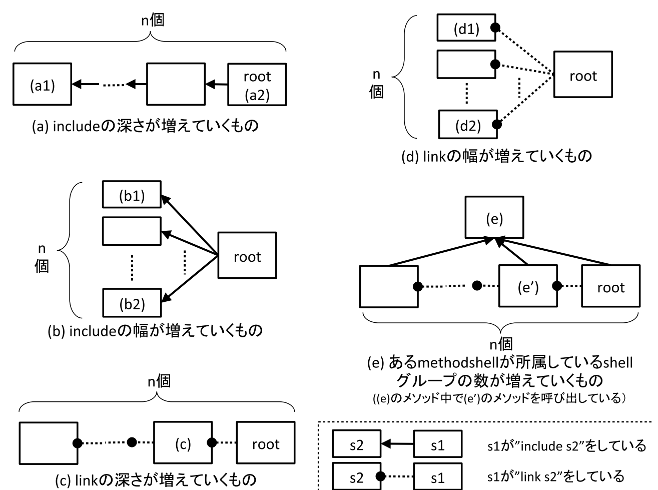

図 12　実験内容：各四角が methodshell を表している．実線の矢印が include 関係，点線の矢印が link 関係を表している

Fig. 12　Experiments: a rectangle denotes a methodshell. An arrow denotes include and a dotted arrow denotes link.

実験結果は図 13，図 14 の 8 つのグラフのようになっている．これはメソッド呼び出し 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グラフ化したものである．(a1)，(b1)，(b2)，(d1)，(d2) では ReflectionAPI を用いた実装では線形に実行時間が増えているのに対し，提案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 数増加にともなう実行時間の増加を実験の範囲では抑えられている．これは，ReflectionAPI を用いたときにはすべての methodshell を順に探索しなければならない場合があるが，提案手法ではメソッド呼び出しが一意に定まるためである．

提案手法を用いた場合，コンパイル時間はいずれも 1 ミリ秒前後である．コンテキスト数が増えると，複製するメソッドが増えるため，(c) および (e) の実験ではコンパイラによって生成されるバイトコードのバイト数が線形に増加している．

GluonJ，オリジナルの Java との比較

メソッドの再定義を行う方法は 2 つある．1 つは破壊的クラス拡張を用いて再定義する方法であり，もう 1 つは直接ソースコードを書き換える方法である．また，破壊的クラス拡張を導入している他の言語のうち，GluonJ を含むいくつかの言語は include の機能を有している．

我々は，4 章の Method Shells の実装と，GluonJ を用

© 2014 Information Processing Society of Japan

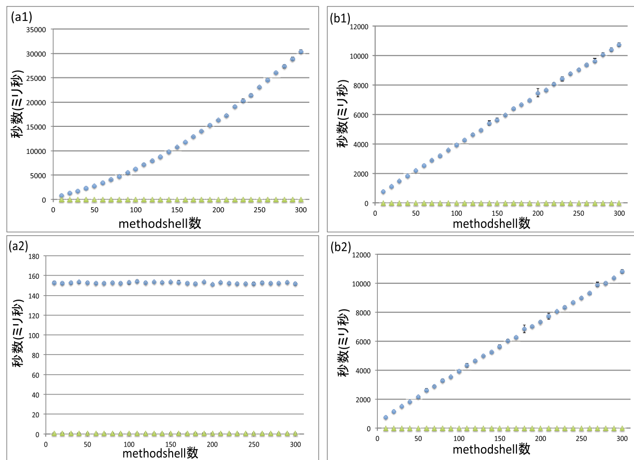

図 13　(a1)，(a2)，(b1)，(b2) の実験結果．青い丸が reflectionAPI を，緑の三角が提案手法による実装を表している

Fig. 13　The results of (a1), (a2), (b1), (b2). Blue dots denote the implementation using reflection API. Green triangles denote the proposed implementation.

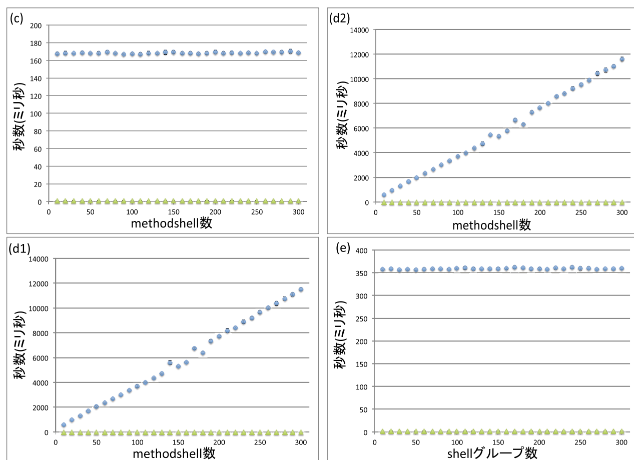

図 14　(c)，(d1)，(d2)，(e) の実験結果．青い丸が reflectionAPI を，緑の三角が提案手法による実装を表している

Fig. 14　The results of (c), (d1), (d2), (e). Blue dots denote the implementation using reflection API. Green triangles denote the proposed implementation.

いた実装およびオリジナルの Java との比較実験を行った．前者は 4 章の 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ない実行速度であることを示すためである．後者は 4 章の 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ない実行速度であることを示すためである．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 で比較実験を行った．実験環境は，前述したものと同じである．GluonJ については GluonJ 2.3 を用いた．

実験結果は**図 15** のようになった．これはメソッド呼び出し 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グラフ化したものである．4 章の提案手法，GluonJ，オリジナルの Java，ともに実行時間に大きな差が見られないことを示せた．また，実験の範囲では include による実行速度の増加がほとんどないことが確認できた．

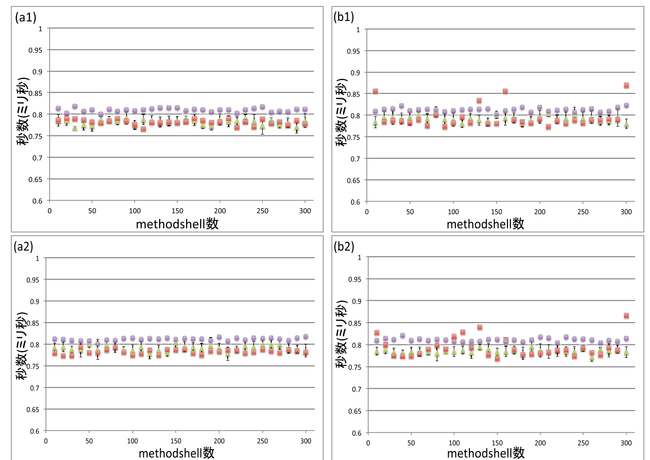

図 15　(a1)，(a2)，(b1)，(b2) の実験結果．緑の三角が提案手法による実装，赤い四角が GluonJ，紫の丸がオリジナルの Java を表している

Fig. 15　The results of (a1), (a2), (b1), (b2). Green triangles denote the proposed implementation. Red rectangles denote the implementation in GluonJ. Violet dots denote the one in Java.

6.　関連研究

破壊的クラス拡張は，前述した AspectJ，Ruby，GluonJ のほかに，MultiJava [14]，Jiazzi [13]，そして後述する Newspeak，Classbox，Context-oriented programming など様々な言語に取り入れられている．

NewSpeak [5] では，すべてのクラス名がバーチャルになっており，サブクラスで上書きすることができる．サブクラスで上書きを繰り返す部分は，include で破壊的クラス拡張の上書きを繰り返す機能とよく似ている．しかし，NewSpeak は link のような機能は持たない．

Ruby [1] の Refinement と Classbox [4] はともに，破壊的クラス拡張に対してスコープの制限を行うことができるメカニズムである．それぞれ独自のモジュールを持ち，その中に破壊的クラス拡張を書き，そのモジュールの中でのみ破壊的クラス拡張が有効であるようにしている．これらのメカニズムでは，破壊的クラス拡張のスコープを制限することはできるが，実行時にスコープを切り替える機能は持たない．

Context-oriented programming（COP）[10] は，どの破壊的クラス拡張を有効になるかを実行時に切り替えることができるメカニズムである．COP では，破壊的クラス拡張は layer と呼ばれるモジュールの中に書かれる．その layer を有効にするかしないかを選択することで，有効な破壊的クラス拡張を実行時に切り替えることができる．COP では，破壊的クラス拡張を切り替えたいときには，つねに明

© 2014 Information Processing Society of Japan

示的に切替え先を指定する必要がある．そのため，メソッド呼び出しによって暗黙的に破壊的クラス拡張が切り替わる本モジュール機構とは異なっている．

本論文で示したモジュール機構はすでに我々が論文 [15] で発表している．本論文はこの論文を拡張し，セマンティクスの改良や実験データの追加を行ったものである．より実用性の高いセマンティクスへと改良した．

7. まとめ

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ことができるモジュール機構 Method Shells を提案した．破壊的クラス拡張とは，既存のソースコードを書き換えることなく，差分のみの記述だけで，メソッドの追加・再定義を行うことができるメカニズムである．破壊的クラス拡張はコードの再利用を促進する反面，同じメソッドを複数回再定義してしまいメソッドの衝突を招く恐れがある．本モジュール機構は有効にするメソッドを実行時に切り替えることによって，メソッドの衝突を避けることを可能にした．本研究では，本モジュール機構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．さらに，提案した実装方法が実際に実行時間の増加を抑えることができることを実験により示した．

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．

参考文献

[1] Ruby programming language, available from ⟨http://www.ruby-lang.org/⟩.

[2] Aksit, M., Rensink, A. and Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points, *Proc. 8th ACM Int'l. Conf. on Aspect-Oriented Software Development, AOSD '09*, pp.39–50, ACM (2009).

[3] Batory, D., Sarvela, J.N. and Rauschmayer, A.: Scaling Step-Wise Refinement, *IEEE Trans. Softw. Eng.*, Vol.30, No.6, pp.355–371 (2004).

[4] Bergel, A.: Classbox/J: Controlling the scope of change in Java, *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'05*, pp.177–189, ACM (2005).

[5] Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W. and Miranda, E.: Modules as objects in newspeak, *Proc. 24th European Conf. on Object-Oriented Programming, ECOOP'10*, pp.405–428, Springer-Verlag (2010).

[6] Chiba, S., Igarashi, A. and Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch, *Proc. ACM Conf. on Object-Oriented Programming Systems Languages and Applications, OOPSLA '10*, pp.539–554, ACM (2010).

[7] Douence, R., Fradet, P. and Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions, *Proc. 1st ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering, GPCE '02*, pp.173–188, Springer-Verlag (2002).

[8] Douence, R., Fradet, P. and Südholt, M.: Composition, reuse and interaction analysis of stateful aspects, *Proc. 3rd Int'l. Conf. on Aspect-Oriented Software Development, AOSD '04*, pp.141–150, ACM (2004).

[9] Ekman, T. and Hedin, G.: The jastadd extensible java compiler, *Proc. ACM Conf. on Object-Oriented Programming Systems and Applications, OOPSLA '07*, pp.1–18, ACM (2007).

[10] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-Oriented Programming, *Journal of Object Technology*, Vol.7, No.3, pp.125–151 (2008).

[11] Igarashi, A., Pierce, B.C. and Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ, *ACM Trans. Prog. Lang. Syst.*, Vol.23, No.3, pp.396–450 (2001).

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pp.220–242, Springer (1997).

[13] Mcdirmid, S., Flatt, M. and Hsieh, W.C.: Jiazzi: New-Age Components for Old-Fashioned Java, *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '11*, pp.211–222, ACM (2001).

[14] Millstein, T., Reay, M. and Chambers, C.: Relaxed MultiJava: Balancing extensibility and modular typechecking, *Proc. ACM Conf. on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA '03*, pp.224–240, ACM (2003).

[15] Takeshita, W. and Chiba, S.: Method Shells: Avoiding Conflicts on Destructive Class Extensions by Implicit Context Switches, *Proc. 13th Int'l Conf. on Software Composition*, LNCS 8088, pp.49–64, Springer (2013).

竹下 若菜

2012 年東京工業大学理学部情報科学科卒業．2014 年東京大学大学院情報理工学研究科創造情報学専攻修士課程修了．プログラミング言語の研究に従事．

千葉 滋　（正会員）

1991 年東京大学理学部情報科学科卒業．1996 年東京大学大学院理学系研究科情報科学専攻博士課程退学．博士（理学）．東京大学助手，筑波大学講師，東京工業大学講師，准教授，教授を経て，2012 年より東京大学大学院情報理工学系研究科教授．プログラミング言語，システムソフトウェアの研究に従事．

© 2014 Information Processing Society of Japan