コールスタックに基づいてクラス拡張の有効範囲を制御するための言語機構の提案

福室 嶺　千葉 滋

クラス拡張の有効範囲を制御するための言語機構を提案する．クラス拡張は既存のクラスに対して元のソースコードを変更することなくメソッドの追加や再定義を行う仕組みで，コードの再利用性を高めることができる一方，クラス拡張がプログラマの意図しない領域で有効になることで誤動作を誘発するなど，危険も大きい．クラス拡張を安全に利用するために，本機構ではプログラマのコードに対する理解度に応じて段階的に有効範囲を拡張できるようにした．これにより，プログラマの理解を超えた領域で不意にクラス拡張が有効になってしまうようなことを回避できる．本機構はクラス拡張を実装するあらゆるプログラミング言語に対して適用可能なアイディアであるが，今回は特に Ruby に対して適用し，まずプロトタイプとして Ruby のライブラリという形での実装を行った．

1　はじめに

直接ソースコードを書き換えることなく既存のクラスを拡張することが出来るクラス拡張 (*class extensions*) は，コードの再利用性を高める手段の 1 つとして幅広く利用されている．クラス拡張に類する言語機構は Smalltalk[3] や，Aspect-Oriented Programming[6], Context-Oriented Programming[5]，C#[8]，Ruby[7] 等，様々なプログラミング言語やパラダイムに取り入れられている．Ruby ではオープンクラスという名前で実装されており，特に人気の高い Web フレームワークである Ruby on Rails[4] では Ruby のビルトインクラスさえもオープンクラスを用いて積極的に拡張している．しかしその一方でクラス拡張は意図しない動作を引き起こしやすい．これは拡張されたクラスの振る舞いが，拡張前の振る舞いを前提にしていたコードと互換性があるとは限らないからである．完全な後方互換性を保ってクラス拡張を実装することは常に可能とは限らなく，また，利用範囲の広いクラスへの拡張の場合はクラス拡張がプログラム全体にどのような影響を与えるかを測ることは難しい．このような問題への対策として，クラス拡張の有効範囲を何らかの方法で制限する仕組みがこれまで数多く提案されてきた．例えば，selector namespaces[10] や，Ruby の refinements，classboxes[2]，method shelters[1], method shells[9] 等がある．しかしこれらの手法は，安全性は高いものの制御できる有効範囲が狭すぎてクラス拡張の有用性を損なってしまっていたり，あるいは逆に有効範囲が広すぎて安全性に不安があるといったように，未だ十分なものになっていないと我々は考えている．

そこで我々は，安全性を損なわない範囲でクラス拡張の有効範囲を制御可能な言語機構 method seals を提案する．method seals はプログラマが理解度に応じて段階的にクラス拡張の有効範囲を広げていくことができる．プログラマの理解を超えた範囲で勝手にクラス拡張が有効になることを防げるため，クラス拡張によってプログラムが不意に破壊されるというようなことは起こりにくくなる．

以降，2 章ではクラス拡張とその問題について述べ，何故クラス拡張の有効範囲を制御する必要があるかを論じ，3 章では我々が提案する method seals の概要を例とともに述べる．4 章では我々が Ruby 上に実装した method seals のプロトタイプについて紹介

Ryo Fukumuro, Shigeru Chiba, 東京大学大学院情報理工学系研究科創造情報学専攻, Dept. of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo.

```
1  puts 1/2        # => 0
2
3  class Fixnum
4    def /(other)
5      quo(other)
6    end
7  end
8
9  puts 1/2        # => 1/2
```

図 1　オープンクラスによるビルトインクラスの書換え

し，5 章ではその実装に対して行った性能計測について報告する．6 章では関連研究について述べ，7 章では本論文のまとめと今後の課題について述べる．

2　クラス拡張の有効範囲を制御する仕組みの必要性

クラス拡張はコードの再利用性を高めるが，既存のコードを不意に破壊してしまう危険性がある．クラス拡張とは既存のクラスを外部から拡張する仕組みであり，メソッドの追加や再定義を直接コードを書き換えることなく行える (実装によっては再定義を禁止するものもある)．このような言語機構は C# や Ruby をはじめとする様々な言語で提供されている．例えば Ruby ではオープンクラスという名前で提供されている．オープンクラスはメソッドの追加や再定義を行うための専用の構文は持たず， 通常のクラス定義と同様の構文を用い，既に同名のクラスがあった場合はそのクラスに対しての拡張となる．オープンクラスはグローバルに有効になり，一度有効にすると無効にすることは出来ない．図 1 はオープンクラスによって Ruby のビルトインクラスである `Fixnum` クラスのメソッドを再定義し，整数型の除算において有理数を返すように変更を加える例である．このような影響範囲の大きな変更は，自身の書いたプログラム内だけでなく，サードパーティライブラリなど外部のコードに影響を及ぼし，正しく動作しなくなる恐れがある．

不意の挙動を防ぎ，クラス拡張を安全に利用するためにはクラス拡張の有効範囲を適切に制御する仕組みを欠かすことは出来ない．図 1 のような危険な例でも，そのクラス拡張が適用されることを前提にして書かれたコード上でのみクラス拡張が有効になるよ

```
1  module ExtFixnum
2    refine Fixnum do
3      def /(other)
4        quo(other)
5      end
6    end
7  end
8
9  class MyMath
10   using ExtFixnum
11   def half(other)
12     other/2
13   end
14 end
15
16 puts 1/2                        # => 0
17 puts MyMath.new.half(1)         # => 1/2
```

図 2　refinements によるビルトインクラスの書換え

うに制御することさえ出来れば問題無い．

Ruby の refinements のように有効範囲をレキシカルスコープに基づいて決定する手法は安全性は高いが，拡張対象となるメソッドが間接的に呼び出される場合ではクラス拡張を有効にすることは出来ない．図 2 は図 1 を refinements を用いて書きなおしたものである．refinements ではクラス拡張は定義しただけでは有効にならず，`using` メソッドを用いて明示的に有効化しなくてはならない．そしてその有効範囲は `using` の呼び出しがレキシカルスコープ下で参照可能な範囲に限定される．つまり，この例では，`MyMath` クラス内に限定されており，`half` メソッドの定義が実際に影響を受けている．16 行目の除算など `MyMath` の外には一切の影響が無い．一方，図 3 では `HTMLReader` の内部で呼び出しているファイルを読み込むための `Reader` クラスを書換え，テストのために一時的に定数を返すようなスタブを作成しようとしている．しかし，`Reader` クラスを直接呼び出した場合は動作するが，`HTMLReader` 経由で呼び出すと `ReaderStub` は有効にならない．これは先に述べたように，`ReaderStub` の有効範囲は `using` 呼び出しがレキシカルスコープ下で参照可能な範囲に限られることによる．すなわち，3 行目の `Reader` の呼び出しからみて 16 行目の `using` 呼び出しが参照不可能な位置にあることに原因がある．間接的に呼び出される場合に

一方，Context-Oriented Programming(文脈指向プログラミング，COP) [5] のようにクラス拡張の有効

```
1  class HTMLReader
2    def read_html(path)
3      r = Reader.new.read
4      parse_html r.read path
5    end
6  end
7
8  module ReaderStub
9    refine Reader do
10     def read(path)
11       return "<div>foo</div>"
12     end
13   end
14 end
15
16 using ReaderStub
17 Reader.new.read "dummy"
18 # => "<div>foo</div>"
19 HTMLReader.new.read_html "dummy"
20 # => No such file or directory - dummy
```

図 3　refinements によるスタブの作成 (期待通りに動作しない)

範囲を動的に決定する手法では拡張対象のメソッドが間接的に呼ばれた場合でも有効にすることができるが，意図しないところでも有効化されてしまう危険がある．“意図しないところ” とは，プログラマがコードを読んでおらず，挙動をよく理解していない部分とそこから間接的に呼ばれるコードのことである．図 3 の例を拡張し，HTMLReader が内部で CSS の解釈も行うようにし，それを動的スコープで有効化するようにしたものを図 4 に示す．with を用いてクラス拡張を有効化し，そのブロック内では間接的に呼び出されるものも含めて全ての関数呼び出しにおいてクラス拡張が有効になり，HTMLReader 内部で呼び出される Reader#read は ReaderStub によって書換えられたものになる．そして同様に CSSReader 内部で呼ばれている Reader#read に対してもクラス拡張は有効になる．仮に，プログラマが HTMLReader の挙動は理解しているが CSSReader の動作については理解していないものとすると，CSSReader 内部の Reader#read において拡張が有効になることはプログラマの意図から外れることになり，致命的なバグを生み出しかねない．実際，図 3 で示した ReaderStub は HTMLReader で使われることを前提に実装されており，CSSReader が解釈できない文字列を返す．

3　意図しない間接呼び出しでのクラス拡張の無効化

```
1  class CSSReader
2    def read_css(filepath)
3      parse_css Reader.new.read filepath
4    end
5  end
6
7  class HTMLReader
8    def read_html(filepath)
9      parsed = parse_html Reader.new.read(filepath)
10     css_reader = CSSReader.new
11     styles = parsed.get_css_paths.map do |p|
12       css_reader.read_css p
13     end
14     parsed.set_style styles
15   end
16 end
17
18 with ReaderStub do
19   # Activate the extension in this block
20   HTMLReader.new.read_html "dummy"
21 end
```

図 4　CSS も解釈できるように拡張された HTML パーサ

プログラマがコードを読んで理解した上でクラス拡張を適用したい部分 (パッケージ)，すなわち “意図したところ” を明示できるようにしたクラス拡張 *method seals* を提案する．Method seals では，“意図したところ” であれば間接的に呼び出されたメソッドであってもそのクラス拡張は有効であり，逆に “意図したところ” と明示しない限りそのパッケージとその先ではクラス拡張は無効になる．このようにパッケージ p とその先で呼び出されるパッケージでクラス拡張 e を無効にすることを “クラス拡張 e についてパッケージ p を *seal* する” と呼ぶ．また，クラス拡張 e についてパッケージ p に付与された seal 属性を剥がすことを “クラス拡張 e についてパッケージ p を *unseal* する” と呼ぶ．デフォルトでは全てのパッケージは seal されており，プログラマはクラス拡張の有効範囲を広げるために自身の理解度と必要性に応じて各パッケージを unseal していくことになる．

Method seals で実際にクラス拡張の有効化及び unseal を行うためには unseal 付き using 宣言を用いる．拡張の有効無効を切り替えるアルゴリズムは次のようになる．

(a) using 宣言でクラス拡張 e を有効化．宣言を含むパッケージ p で有効になる．

```
1 using ReaderStub, [HTMLReader]
2 Reader.new.read "dummy" # => "<div>foo</div>"
3 HTMLReader.new.read_html "dummy"
```

図 5　Method seals によるクラス拡張の有効化

(b) パッケージ p から別のパッケージ q のメソッドを呼び出すと，

(b1) q が using 宣言の unseal パッケージリストに含まれていれば q 内部でもクラス拡張 e が有効になる．以降 (b) を繰り返す．

(b2) q が unseal パッケージリストに含まれていない場合は q 内部とその先に呼び出されるパッケージではクラス拡張 e が無効になる．

unseal 付き using 宣言は具体的には例えば図 5 のようになる．これはクラス拡張 ReaderStub を有効にし，さらにそれについて HTMLReader を unseal している．using によってまずは自身を含むパッケージでクラス拡張が有効になる (a)．したがって，2 行目の Reader#read 呼び出しはクラス拡張が適用されたものになる．次に HTMLReader#read_html を呼び出すことを考える (b)．今回は HTMLReader は unseal パッケージリストに含まれているため，HTMLReader でもクラス拡張 ReaderStub が有効になる (b1)．したがって，図 4 の 9 行目の Reader#read 呼び出しはクラス拡張が適用されたものとなる．一方，12 行目の CSSReader#read_css 呼び出しを考えると，また別パッケージのメソッド呼び出しになるため，(b) の処理が繰り返される．今回は unseal パッケージリストに CSSReader は含まれていないため，クラス拡張は無効になり，CSSReader 内部の Reader#read 呼び出しは拡張前のものが呼び出されることになる (b2)．仮に CSSReader が更に別のパッケージのメソッドを呼び出していたとしても，これ以降 (b) の処理は行われない．

4　Ruby 上のプロトタイプ実装

我々は 3 章で示した method seals の基本的なアイディアを Ruby 上にプロトタイプとして実装した．ただし，現時点では実装上の都合から 3 章で示したものとは異なるシンタックスを持つ．本実装におけるクラス拡張の定義を図 6 に，クラス拡張の有効化を図 7 に示す．クラス拡張を定義するためには seal_define メソッドを用いる．seal_define の第 1 引数には拡張対象のクラス，第 2 引数にはクラス拡張の名前をシンボルで渡す．定義したクラス拡張を有効化するためには seal_using メソッドを用いる．seal_using の第 1 引数にはクラス拡張の名前，第 2 引数には unseal 対象のクラスをリストで渡す．更にクラス拡張を利用するコードをブロックとして渡す．

```
1 seal_define Reader, :ReaderStub do
2   def_method(:read) do
3     "<div>foo</div>"
4   end
5 end
```

図 6　プロトタイプ実装におけるクラス拡張の定義

```
1 seal_using(:ReaderStub, [HTMLReader]) do
2   HTMLReader.new.read_html "dummy"
3 end
```

図 7　プロトタイプ実装におけるクラス拡張の有効化

現在の実装は Ruby 2.0 以降を対象としたライブラリという形で作成されており，全て Ruby で書かれている．デバッガやプロファイラのための API である TracePoint を用いて各メソッド呼び出し時に処理を挟むことで現在のコールパスを取得できるようにしている．また，3 章で示したアルゴリズムに従ってメソッドディスパッチを行うように拡張対象のメソッドを書き換えている．本来 method seals は refinements の自然な拡張として実現できる (何も unseal しなければ refinements と同等) ため，実用的にはそのように実装することが望ましいと考えており，今後パフォーマンスの改善とともに Ruby 処理系の拡張によって実現することを予定している．また，現在は unseal の対象 (パッケージ) はクラスのみだが，その対象をメソッドやモジュールへ広げることを検討している．

5　予備的な性能評価

4 章で述べたとおり，現在のプロトタイプは Ruby 上にライブラリという形で実現されており，デバッガ向け API を用いて各メソッドコールにイベントハン

表 1 *fib*(31)

実装種別	時間 (s)
Ruby	0.363
Method seals あり (sealed)	4.069
Method seals あり (unsealed)	9.422

ドラを設定するなど実行性能を考慮した実装にはなっていない．実行性能の改善は目下の課題であり，本章ではその予備実験として行った現実装の性能評価について報告する．

本実験ではフィボナッチ数の計算を行うプログラムを method seals を用いて実装し，その実行時間を測定した．実験プログラムでは，まず Ruby で method seals を使わずに `Fib` クラスおよび実際の計算を行う `fib` メソッドを作成し，その後で method seals で外部から `fib` を同じ実装内容で再定義した．これを用いて，元の `Fib` クラスを用いて計算した場合と method seals で再定義を行った場合のそれぞれで計測を行った．更に，method seals を用いた場合は，unseal して再定義された `fib` が呼ばれるケースと unseal せず元の `fib` が呼ばれるケースの両方を計測した．これを行ったのは，コールパス上に seal されているクラスが現れた時点でそれ以降は常に拡張が無効になることが確定し，その分の処理内容が減ることから unseal しないケースでは実行時間が短くなることが予想されるためである．実験環境はプロセッサが Intel Core i7-4578U 3.00GHz，メモリが 16GB で，Ruby のバージョンは 2.1.5p273 を用いた．

表 1 に *fib*(31) を計算した結果を示す．提案手法を用いると，Ruby で実装したものに比べて 26 倍，最終的に拡張前のメソッドが呼ばれる場合でも 11 倍の実行時間と，非常に大きなオーバーヘッドがかかることがわかる．これは全てのメソッド呼び出しごとに現在実行されている環境が seal されているかどうかの判定を行っていることや，拡張対象のメソッドがメソッドディスパッチを行うものへと書き換えられていることが大きな要因だと思われる．

6 関連研究

クラス拡張に類する言語機構や関連研究は数多く存在する．2 章で述べたように，Ruby [7] のオープンクラスは既存クラスに対して外部からメソッドの追加や再定義が可能だが，破壊的に行われ一度適用すると元に戻すことは出来ない．また，Ruby の refinements はクラス拡張の有効範囲制御を行うことができるが，間接呼び出しで有効にすることは出来ない．Selector namespaces は Modular Smalltalk [10] で導入された概念で，クラス拡張をモジュール化し，それを特定のスコープにインポートすることでクラス拡張の有効範囲をそのスコープ下に限定することができる．Ruby の refinements によく似ており，間接呼び出しで有効にすることは出来ない．Context-Oriented Programming [5] は実行時に文脈に応じてクラスの振る舞いを変えることができるプログラミングパラダイムである．文脈は動的スコープで有効無効が切り替えられるため，間接呼び出しでも有効だが，全ての文脈について各クラスの振る舞いの変更をプログラマが把握することは難しく，理解の浅さから意図しない動作を引き起こしかねない．Classboxes [2] は *classbox* というモジュール内にクラス宣言とクラス拡張を定義することができる．ここで定義したクラスやクラス拡張はその classbox 内だけで有効だが，他の classbox からもインポートすることで自身の classbox 内で利用できるようになる．間接呼び出しでも有効 (この機能を *local rebindings* と呼ぶ) だが，一度インポートしたクラス拡張はその classbox 内で常に有効になるため，プログラマがその classbox 内の実装詳細をインポートしたものも含めて全て理解していなければ，意図しないクラス拡張の適用が起こりうる．Method shelters [1] は本研究の前身研究の 1 つで，*method shelter* というモジュールを導入している．Method shelters は Classboxes によく似ており，method shelter 内にクラスやクラス拡張を定義したり，他の method shelter のものをインポートすることができる．Classboxes と大きく違う点は，method shelter は内部が *exposed chamber* と *hidden chamber* という 2 つの部分に分かれており，どちらにクラスやクラス拡張を定義する

かによって，外部からのインポートを許可するかどうかを制御できるところにある．しかし，どちらの chamber に定義するかはインポート先の実装者に委ねられているため，やはりインポートする側がインポート先の実装について理解していないと依然として意図しない挙動を引き起こしうる．Method shells [9] は本研究のもう 1 つの前身研究である．Method shells は method shelters が複雑化しすぎたことを踏まえ，chamber を廃止し，別途 2 種類のインポート方法を用意することでよりシステムを簡素にした．また，インポート元でどのようにインポートするかを制御できるようになったため，安全性が高まった．しかし，単純なシンタックスを獲得した一方で依然として複雑なセマンティクスを持っており，暗黙のうちに複雑なメソッドディスパッチが行われるためやはりプログラマが全体像を把握することは難しく，意図しない挙動を引き起こしうる．本研究はこれを踏まえ，暗黙的な解決を諦めることでより単純で安全なセマンティクスを得ることを目指したものである．

7 まとめ

本研究では，クラス拡張を安全に利用するために，プログラマのコードに対する理解度に応じて段階的に有効範囲を拡張できる言語機構 method seals を提案した．一般に，クラス拡張は既存のクラスを拡張するという性質上，拡張対象のクラスがプログラム上のどこで使われているかを完全に把握した上で利用しなければ不意にプログラムを破壊しかねない．サードパーティライブラリの利用や多人数による開発を考慮すると，プログラム全体を把握することは難しく，クラス拡張をプログラマが把握している領域だけで有効にするとしたほうが現実的である．提案手法ではプログラマのコードに対する理解をそのまま明示することを強制することでより安全にクラス拡張を利用できるようにすることを目指した．

今後の課題としては，提案手法をより実用的なプログラムに対して適用することで基本的なアイディアの有用性を検証する，実用化に向けてパフォーマンスの改善を図る等が考えられる．

参考文献

[1] Akai, S. and Chiba, S.: Method shelters: avoiding conflicts among class extensions caused by local rebinding, *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, ACM, 2012, pp. 131–142.

[2] Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R.: Classboxes: Controlling visibility of class extensions, *Computer Languages, Systems & Structures*, Vol. 31, No. 3(2005), pp. 107–126.

[3] Goldberg, A. and Robson, D.: *Smalltalk-80: the language and its implementation*, Addison-Wesley Longman Publishing Co., Inc., 1983.

[4] Hansson, D. H.: Ruby on Rails, http://rubyonrails.org/.

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

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: *Aspect-oriented programming*, Springer, 1997.

[7] Matsumoto, Y.: Ruby Programming Language, http://www.ruby-lang.org/.

[8] Microsoft Corporation: C# Language Specification, https://msdn.microsoft.com/library/ms228593.

[9] Takeshita, W. and Chiba, S.: Method Shells: avoiding conflicts on destructive class extensions by implicit context switches, *Software Composition*, Springer, 2013, pp. 49–64.

[10] Wirfs-Brock, A. and Wilkerson, B.: A overview of modular smalltalk, *ACM SIGPLAN Notices*, Vol. 23, No. 11, ACM, 1988, pp. 123–134.