情報処理学会論文誌　プログラミング　Vol.9　No.4　16–26 (Sep. 2016)

有効範囲を既知のコールパスに限定するRuby向けの安全なクラス拡張Method seals

福室 嶺[1,a)]　千葉 滋[1,b)]

受付日 2016年1月28日, 採録日 2016年4月25日

概要：本論文では，有効範囲を既知のコールパス上に限定することで安全に利用できるクラス拡張 method seals を提案する．クラス拡張は既存のクラスを拡張するための言語機構であり，Ruby や AspectJ，C#など様々な言語に取り入れられている．クラス拡張を用いることで，既存のクラスに対してソースコードを書き換えることなくメソッドの追加や再定義を行える．しかし，クラス拡張はモジュラリティの向上に寄与する一方で，誤動作を引き起こしやすい．これはクラス拡張どうしの衝突や，クラス拡張が想定外の領域で有効になることが主な要因である．こうした問題を解決するために，プログラマのコードに対する理解度に応じてその有効範囲を段階的に拡張できるように設計されたクラス拡張 method seals を提案する．本機構では，プログラマにとって未知のパッケージのコードはブラックボックス内にあると見なし，クラス拡張が無効になる．また，ブラックボックス内から発するコールパス上ではそれ以外のコードも一時的にブラックボックス内にあるものと見なす．これにより，クラス拡張がブラックボックス内のコードに対して予期せぬ影響を与え，ひいてはプログラム全体の誤動作を引き起こすことを防げる．本研究では Ruby 処理系上に method seals を実装し，いくつかの条件下でその実行性能を評価した．

キーワード：クラス拡張，Ruby，スコープ

Method Seals: Safe Class Extension for Ruby Limiting the Scope to Known Call Paths

Ryo Fukumuro[1,a)]　Shigeru Chiba[1,b)]

Received: January 28, 2016, Accepted: April 25, 2016

Abstract: We propose method seals, a safe class extension limiting its scope to known call paths. Class extensions allow us to extend an existing class to add or modify a method without rewriting its code, and they are supported by various languages including Ruby, AspectJ, and C#. They are easy to cause unintended behavior due to collision among multiple extensions or being enabled in an unexpected domain while they contribute to improvement modularity. To address such a problem, we propose method seals designed to allow class extensions control their effective range depending on the degree of understanding about a code base of programmers. Method seals regard code in an unknown package as a black box and disable class extensions there. Furthermore, they also temporarily regard callees of such a package as being in a black box. This prevents class extensions from giving an unexpected impact on codes in black boxes and causing an unintended behavior of a whole program. We implement method seals on Ruby interpreter and evaluates its performance under several conditions.

Keywords: class extensions, Ruby, scope

1. はじめに

現代のソフトウェア開発において，プログラムのモジュラリティを向上させ，コードの再利用性を高めることは半ば常識と化しており，可能な限りソフトウェア内のコード

[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)] fukumuro@csg.ci.i.u-tokyo.ac.jp
[b)] chiba@acm.org

の重複を排し，プログラム中の複数箇所から利用されるコードはモジュールとして切り出すということは当たり前に行われている．しかしながら，再利用性の高いソフトウェアを設計することは簡単ではない．拡張性に十分配慮し設計され，実際に広く利用されているライブラリやフレームワークであっても，ユーザの要求を満たせないことは往々にしてある．

我々がクラス拡張（class extensions）と総称している仕組みを用いると，比較的容易にコードの再利用性を高めることができる．クラス拡張は直接ソースコードを書き換えることなくクラスを外部から拡張する仕組みで，メソッドの追加や再定義を行える．これを用いることで，たとえば振舞いに不満のあるライブラリであっても，直接ソースコードを書き換えることなく，その振舞いを拡張できる．クラス拡張はクラスの継承木とは独立してクラスの振舞いを拡張できるため，通常のクラスシステムのみでは再利用の難しかったコードが再利用可能になる．

クラス拡張に類する言語機構は Ruby [1] や Smalltalk [2], aspect-oriented programming [3], context-oriented programming [4], C# [5], CLOS [6], Objective-C [7] など，様々な言語やプログラミングパラダイムに取り入れられ，広く利用されている．しかし，クラス拡張は意図しない動作を引き起こしやすい．これはクラス拡張の影響が拡張対象のクラスを元々参照していたコードにまで及ぶため，クラス拡張がプログラム全体に及ぼす影響を正確に把握することが難しいことが主な要因である．

このような問題を解決するため，クラス拡張の有効範囲を制限する仕組みがこれまで数多く提案されてきた．たとえば，selector namespaces [8] や，Ruby の refinements, classboxes [9] などがある．このような手法としてはおおまかに静的にその有効範囲を決定する手法と，動的に決定する手法とが存在する．一般に静的に決定する手法は安全性は高いものの機能性が低いことが多い．たとえば，Ruby の refinements ではクラス拡張の有効範囲をレキシカルスコープ下に限定する．これはソースコードの構文構造と，クラス拡張の有効範囲が静的に対応するため，利用者にとってもその有効範囲が理解しやすく，誤動作を引き起こす可能性は低い．一方，複数のスコープにまたがってクラス拡張を有効にすることができないため，利用できるシーンが限定される．Context-oriented Programming のように動的スコープを用いる手法では，複数のスコープにまたがってクラス拡張を利用することができるため機能性は高い．一方，動的な手法では一定範囲にその有効範囲を絞ることができるものの，その絞った範囲においてはグローバルに拡張が有効になるため，有効範囲をまったく制限しないグローバルスコープと同様にクラス拡張の影響範囲を予測することが難しいことがある．このように，これらの手法は一長一短であり，いま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，安全性に配慮した有効範囲の制御が可能なクラス拡張 *method seals* を提案する．Method seals は利用者の理解の及ぶ範囲でしかクラス拡張が有効にならない．クラス拡張の影響が思わぬところに現れることがないため，クラス拡張が意図しない動作を引き起こすことが少なくなる．さらに本研究では method seals を Ruby の処理系を拡張することで実装し，またその性能の測定を行った．

以降，2 章ではクラス拡張の詳細とその問題点について述べ，クラス拡張の有効範囲を制御する必要性について述べる．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．また，同時に method seals の制限事項についても触れる．4 章では，我々が拡張した Ruby 処理系の概要と，拡張した処理系に対して行った性能測定について報告する．5 章では関連研究について述べ，6 章では本論文のまとめと今後の課題について述べる．

2. クラス拡張とその問題点

クラス拡張はプログラムの再利用性を向上させる一方で，誤動作を引き起こしやすいという問題がある．クラス拡張を用いることで，複数のクラスにまたがる関心事（横断的関心事，cross-cutting concern）[3] をモジュールとして切り出すことが可能になり，プログラム全体のモジュラリティの向上が期待できる．あるいは，組み込みクラスやサードパーティライブラリなど直接ソースコードを編集する権限がない場合でも外部からクラスの挙動を拡張することができるため，コードの再利用性が高まる．たとえば Ruby ではオープンクラスがクラス拡張に相当する．**図 1** はオープンクラスを用いてテスト用のスタブを作成する例である．ここでは HTML ファイルの解析を行う `HTMLReader` クラスのテストを行うため，外部ファイルの読み込みを行う `Reader` クラスの `read` メソッドを再定義し，定数文字列を返すように振舞いを変更している．`HTMLReader` のテストではこの定数文字列に対して正しく解析が行われるかをテ

```
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  # Redefine `Reader#read`
9  class Reader
10   def read(path)
11     return "<div>mytext</div>"
12   end
13 end
14
15 Reader.new.read "non-existing-file"
16 # => "<div>mytext</div>"
17 HTMLReader.new.read_html "non-existing-file"
18 # => #<HTMLElement:0x3daf382 div[mytext]>
```

図 1　オープンクラスによるスタブの作成例
Fig. 1　Creating a stub with open classes.

```
1  class CSSReader
2    def read_css(path)
3      r = Reader.new
4      parse_css r.read path
5    end
6  end
```

図 2　CSS ファイルを解析するクラス
Fig. 2　A class to parse CSS files.

ストすればよい．オープンクラスではクラスを拡張するための特別な構文は用意されておらず，クラス定義を行った際に同名のクラスがすでに存在していた場合はそのクラスへの拡張となる．ただし，オープンクラスはグローバルかつ破壊的に有効になるため，図 1 の利用例は現実的ではない．たとえば，プログラム中のどこかに `HTMLReader` と同様に `Reader` クラスを用いて CSS ファイルの解析を行う `CSSReader` クラス（**図 2**）が存在したとする．この場合，`Reader` クラスへの変更の影響は `CSSReader` クラスにまで及んでしまう．`CSSReader` の存在，あるいはその実装詳細についての知識をクラス拡張の利用者が持っていない場合，そのクラス拡張が `CSSReader` に影響を及ぼすということを予測することはできない．このようなクラス拡張の影響範囲に関する認識性の低さはクラス拡張が抱える特徴的な問題点である．したがって，クラス拡張を安全に利用するためには，利用者にとってその影響範囲が明確であるということが必要不可欠である．このように，クラス拡張によって引き起こされる問題の原因の多くは，利用者がクラス拡張を適用する際にその影響範囲を正しく予測できていないことによる．図 1，図 2 で示した例でも，クラス拡張の利用者が，クラス拡張の影響が `CSSReader` へ及ぶことさえ予測できていれば，少なくとも事前にプログラムを工夫することで実行時に問題が起きることを防止できる．

これまで述べてきたように，Ruby のオープンクラスのようなグローバルに有効になるクラス拡張は，利用者にとってその影響範囲が明確であるとはいえない．セマンティクスそのものは単純で利用者にとっても明確だが，新たにクラス拡張を定義した際にそれがプログラム全体にどのような影響を与えるかということは明確ではない．これを正確に把握するためには利用者がプログラムすべてについて理解している必要がある．拡張対象のクラスが自身の作成したクラスであればまだよいが，組み込みクラスやサードパーティライブラリに対する拡張であった場合，その拡張対象のクラスがプログラム中でどのように利用されているかをすべて把握するのは容易なことではない．

Ruby の refinements のようにクラス拡張の有効範囲を静的スコープ下に制限する手法はその影響範囲が利用者にとって明確で安全性が高いが，拡張対象のメソッドがスコープをまたいで間接的に呼び出される場合に拡張を有効にすることはできない．**図 3** は refinements を用いて 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．実際に Ruby の mathn モジュール内で同様の変更がオープンクラスを用いて行われている．Refinements はオープンクラスとは異なり，クラス拡張を定義しただけではその拡張が有効にならず，`using` 宣言（正確にはメソッド）を用いて明示的に有効化する必要がある．図 3 の例では 10 行目の `using` 宣言により 11 から 13 行目で Fixnum に対するクラス拡張が有効になる．`using` で有効にしたクラス拡張の有効範囲はその `using` 宣言を含むレキシカルスコープ内に限定され[*1]，その外部にはいっさい影響を与えない．このような手法は安全性という点では有効だが，間接的な呼び出しに対応できないため利用できる場面は限られている．たとえば，**図 4** は refinements を用いて図 1 の例を書き直したものである．図 4 では，`using` 宣言と同一スコープにある `Reader#read` メソッドの呼び出し（19 行目）では正しく拡張が適用されたものが呼び出される．しかし，`HTMLReader#read_html` メソッドを経由

```
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 3　Refinements によるビルトインクラスの書き換え
Fig. 3　Extending a built-in class with refinements.

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

図 4　Refinements によるスタブの実装（意図どおりには動作しない）
Fig. 4　Creating a stub with refinements (Does not work as intended).

[*1] 正確にはもう少し変則的なスコープを持つ．たとえば，Ruby ではメソッド内からメソッドの定義元である外側のスコープのローカル変数を参照することはできないが，外側のスコープで呼ばれた `using` の有効範囲はメソッド定義内にまで及ぶ．

```
1  class HTMLReader
2    def read_html(filepath)
3      parsed = parse_html Reader.new.read(filepath)
4      css_reader = CSSReader.new
5      styles = parsed.get_css_paths.map do |p|
6        css_reader.read_css p
7      end
8      parsed.set_style styles
9    end
10 end
11
12 with ReaderStub do
13   # Activate the extension in this block
14   HTMLReader.new.read_html "non-existing-file"
15   # => #<HTMLElement:0x3daf382 div[mytext]>
16 end
```

図 5　動的スコープを用いたスタブの実装
Fig. 5　Creating a stub with dynamic scope.

して呼び出す場合（21 行目）では拡張前のものが呼び出される．これは先述したように，`ReaderStub` の有効化宣言（18 行目）が，拡張対象である `Reader#read` メソッドの呼び出し（4 行目）から見て参照可能なスコープ内に位置していないことによる．

一方，COP のようなクラス拡張の有効範囲を動的スコープに基づいて決定する手法は，複数のスコープにまたがる間接的な呼び出しでも利用可能だが，グローバルスコープと同様，クラス拡張の影響範囲の認識性には難がある．**図 5** は，`HTMLReader` を CSS ファイルの解析も行うように拡張し，これに対して `Reader` クラスのスタブを適用する例である．ただし，Ruby にはこのような機能は存在しないため，この例は擬似コードであることに注意されたい．この例では `with` 文を用いてクラス拡張 `ReaderStub` を有効にすることを宣言しており，またその有効範囲は続くブロック内に限定されている．このブロック内では拡張対象のメソッドが間接的に呼び出されても関係なく，つねにクラス拡張は有効になる．したがって，図 4 の例とは異なり，`HTMLReader` から呼ばれた `Reader#read` メソッドについてもクラス拡張が適用されたものになる．しかし，同時に `CSSReader` 内で呼ばれている `Reader#read` メソッドにもクラス拡張が適用されてしまう．グローバルスコープと同様に，クラス拡張の利用者が `CSSReader` の実装詳細についての知識を持っていなければ，この影響を事前に予測することはできない．

3. Method seals：既知のコールパスでのみ有効なクラス拡張

影響範囲が予測可能で，かつ間接的に呼び出されるメソッドに対しても有効なクラス拡張 *method seals* を提案する．従来のクラス拡張に類する言語機構の多くは，安全性について十分に考慮されていないか，あるいは安全性のために機能性を犠牲しているが，method seals は機能性をできるだけ損なわずに安全性を担保することを目指している．

Method seals ではクラス拡張の有効化と同時に「知っているところ」を明示できる．ここでの「知っているところ」とは，クラス拡張の利用者（`using` 宣言を用いてクラス拡張を利用しようとするプログラマ）が「コードを読んで理解した部分（パッケージ）」を意味している．「知っているところ」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．したがって，クラス拡張は必ずクラス拡張の利用者の理解の及ぶ範囲でしか有効にならず，意図せず誤動作を引き起こす危険性を軽減できる．なお，ここでのパッケージとは何らかの基準でまとめられたコードの集まりを意味しており，具体的に Ruby でパッケージに該当する概念としてはクラスやメソッド，モジュールがあげられる．ただし，4 章で示す現在の我々の実装ではクラスのみをサポートしており，すなわちパッケージとクラスは等価である．仮に method seals を Ruby 以外の言語上に実装する場合，そのホスト言語の持つモジュール機構の存在によってパッケージとして扱う単位は変化しうる．

パッケージ名	::=	クラス名 | モジュール名 | メソッド名
using 宣言	::=	“using” クラス拡張名
	|	“using” クラス拡張名 “,” “[” (パッケージ名 “,”)* パッケージ名 “]”
クラス拡張	::=	“module” クラス拡張名 (“refine” クラス名 “do” メソッド宣言* “end”)* “end”

図 6　using 宣言とクラス拡張の構文規則
Fig. 6　Syntax of the using declaration and class extension.

Method seals では，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．デフォルトではすべてのパッケージは seal されており，クラス拡張の有効範囲を広げるためには unseal するパッケージを明示していく必要がある．

Method seals は Ruby の refinements の自然な拡張になっており，拡張の有効化宣言は refinements と同様 using 宣言を用いるが，加えて unseal するパッケージを指定できる．using 宣言の構文規則を**図 6** に示す．ただし，先述のとおり現在の我々の実装ではクラスのみがパッケージとして受理可能である．パッケージを省略した場合は refinements と同様に振る舞うため，たとえば図 3 の refinements の例はそのまま動作する．

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．
(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

```
1  using ReaderStub, [HTMLReader]
2  Reader.new.read "non-existing-file"
3  # => "<div>mytext</div>"
4  HTMLReader.new.read_html "non-existing-file"
5  # => #<HTMLElement:0x3daf382 div[mytext]>
```

図 7　Method seals におけるクラス拡張の有効化
Fig. 7　Activating a class extension in method seals.

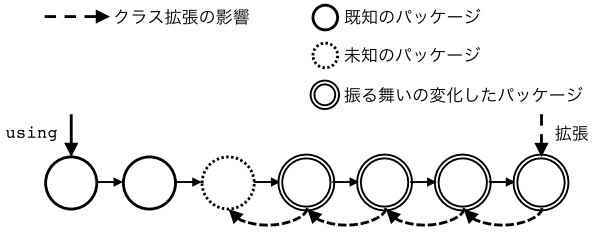

図 8　伝搬してゆくクラス拡張の影響
Fig. 8　Propagation of influence of a class extension.

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．

Method seals における using 宣言の例を**図 7** に示す．この例では，クラス拡張 `ReaderStub` を有効化するとともに，そのクラス拡張について unseal するパッケージ，すなわち `HTMLReader` を指定している．この例がどのように動作するかを順を追って見ていく．なお，ここでの `HTMLReader` は図 5 で示した，CSS ファイルの読み込みに対応したものとする．まず，using 宣言を含むパッケージ，すなわちここではトップレベルスコープにおいてクラス拡張 `ReaderStub` が有効になる (a)．したがって，図 7 の 2 行目の `Reader#read` メソッド呼び出しは拡張されたものになる．次に，4 行目の `HTMLReader#read_html` が呼び出されるとき，`HTMLReader` が unseal パッケージリストに含まれているかを確認する (b)．この例では `HTMLReader` は unseal パッケージリストに含まれているため，`HTMLReader` 内でもクラス拡張 `ReaderStub` が有効になり，`HTMLReader` クラス内で直接呼び出される `Reader#read` メソッドの呼び出し（図 5 の 3 行目）はクラス拡張が適用されたものになる (b1)．一方，`HTMLReader` から `CSSReader#read_css` が呼び出されると（図 5 の 6 行目），再び新たなパッケージのメソッド呼び出しとなるので，(b) の処理が繰り返される．`CSSReader` は unseal パッケージリストに含まれていないため，`CSSReader` 内ではクラス拡張が無効になり，`CSSReader` から呼び出される `Reder#read` メソッド（図 2 の 4 行目）は拡張前のものになる (b2)．仮に `CSSReader` がさら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．

3.1　クラス拡張から保護される「知らないところ」

Method seals を用いることで，「知らないところ」にクラス拡張が影響を及ぼすことによって混入するバグを防ぐことができる．2 章で述べたように，クラス拡張の問題はその影響範囲の予測が難しく，意図しない動作を引き起こしやすいという点にある．理想的にはクラス拡張の利用者の意図したままにクラス拡張の有効範囲が定まれば，利用者にとってこれ以上の明確さはないように思える．しかし現実には，元来主観的な利用者の「意図したところ」を簡潔な言語機構で表現することは困難である．そこで我々は，利用者が「意図したところ」を柔軟に表現できるような言語機構を考案することは目指さず，多少冗長になりがちでも安全側に倒すことで，少なくとも「知らないところ」にクラス拡張が影響を及ぼすことだけは防ぐことを目指した．これには，仮にクラス拡張が一部の既存のコードに対して望ましくない影響を及ぼす場合であっても，その事実さえ利用者が理解していれば，多くの場合は何らかの対策を行えるだろうという期待が根底にある．

さらに我々は「知らないところ」の定義に，未知のパッケージだけでなくそのパッケージを経由して呼ばれるすべてのパッケージも含めるようにした．これらのパッケージは，たとえクラス拡張の利用者がそのソースコードを読んでおりその振舞いを理解していたとしても，利用者が振舞いを理解していない未知のパッケージを経由して呼ばれている間に限り「知らないところ」として扱われ，クラス拡張の影響から保護される．これはクラス拡張による変更の影響は必ずコールパス上の未知のパッケージへ伝搬する可能性があるからである．

図 8 にクラス拡張の影響が伝搬してゆく様子を示す．図 8 は左端の `using` 宣言を含むパッケージから拡張対象のメソッドを含む右端のパッケージに至るまでのコールパスを表している．まず拡張対象のメソッドを含む右端のパッケージがクラス拡張の影響を受け，その振舞いが変化する．そして，そのパッケージの振舞いの変化によって，そのパッケージを参照していたパッケージの振舞いが変化する．そしてまたその振舞いの変化の影響が参照元のパッケージへと，というように，クラス拡張の影響はその拡張対象のクラスの呼び出し元，さらにその呼び出し元へと次々伝搬していく．したがって，図中に示すようにコールパス上に未知のパッケージが存在していた場合には，そのパッケージにもクラス拡張の影響が伝搬する可能性がある．もちろん，クラス拡張の影響が必ずパッケージの振舞いを変えるとは限らないので，伝搬が途中で止まることもありうる．しかし，重要なことは影響が伝搬する可能性があることであり，我々の目指す安全性という観点から，これを無視することはできない．そして，未知のパッケージへ伝搬したクラス拡張の影響が，害のあるものかを正確に把握するためには，未知のパッケージの実装詳細を調べるほかな

```
1  rows = []
2  rows << ['One', 1]
3  rows << ['Two', 2]
4  rows << ['Three', 3]
5  table = Terminal::Table.new :rows => rows
6
7  # > puts table
8  #
9  # +-------+---+
10 # | One   | 1 |
11 # | Two   | 2 |
12 # | Three | 3 |
13 # +-------+---+
```

図 9　Terminal Table の出力例
Fig. 9　An example output of Terminal Table.

```
1  module FullWidthLength
2    refine String do
3      def length
4        # Returns total width of str taking
5        # full-width chars into account.
6      end
7    end
8  end
```

図 10　Terminal Table を全角文字に対応させるためのクラス拡張
Fig. 10　A class extension to add support for full-width characters to Terminal Table.

い．したがって，「知らないところ」に拡張の影響が及ぶことを避けるという本研究の方針に則し，未知のパッケージが現れた時点でクラス拡張を無効にする必要がある．

3.2　利用例

より現実的な method seals の利用例として，Terminal Table[*2] を拡張する例を取り上げる．Terminal Table は与えられたコレクションを人間が読みやすい表形式へテキストフォーマットで出力するためのライブラリで，コマンドラインアプリケーションの実装などに用いられている．**図 9** は Terminal Table のドキュメントより抜粋した Terminal Table の出力の一例である．

Terminal Table は平仮名や漢字のような全角文字には対応しておらず，全角文字も半角文字と同様に幅の計算が行われるために表示が崩れてしまうという問題がある（バージョン 1.5.2 現在）．等幅フォント環境で正しく表示するためには，全角文字は半角文字の 2 倍の幅を持つものとして計算してやる必要がある．実際に Terminal Table の実装を調べると，文字幅の計算には Ruby 標準の `String` の `length` メソッドを用いていることが分かる．したがって，この `length` メソッドを書き換え，全角文字を考慮に入れるように振舞いを変えるクラス拡張を**図 10** のように作成すればよい．

なお，この例に対してオープンクラスを用いることは適当でない．拡張対象である `String` クラスは Ruby の標準クラスであり，その振舞いを変更したことによる影響は広範に及ぶ．したがって，意図しない動作を引き起こす公算が大きい．また同様に，refinements を用いることも適当でない．2 章で述べたように，refinements ではクラス拡張をスコープを跨いで有効にすることはできない．

[*2] https://github.com/tj/terminal-table

```
1  using FullWidthLength,[Class, Terminal::Table,
2                         Array, Integer]
3  table = Terminal::Table.new :rows => rows
4  puts table
```

図 11　クラス拡張 FullWidthLength の有効化
Fig. 11　Activating the FullWidthLength extension.

このクラス拡張を利用するためには**図 11** のようにする．Unseal しているパッケージのリストは Terminal Table のバージョン 1.5.2 時点のコードを元に作成した．解析の結果，`Terminal::Table` クラス内で呼びだされている `String` の `length` メソッドを拡張すればよいと分かったので，直感的には `Terminal::Table` クラスのみ unseal すればよいように思えるが，実際にはさらに `Class` や `Array` などのクラスも加えて unseal する必要がある．これは文字幅の計算がコンストラクタ内で行われているために `Class` の `new` メソッドを経由することや，`Array#map` のようなブロックを受け取って実行するようなメソッドを経由することによる．

3.3　制限事項

Method seals にはいくつかの制限事項が存在する．本節ではこれらの制限事項について述べ，また，考えうる対策について議論する．

3.3.1　無害なクラス拡張に対する冗長性

明らかに無害なクラス拡張に対してはクラス拡張を有効にする手順が冗長になりすぎるという問題がある．無害なクラス拡張の例としては，自身が作成したクラスに対する拡張など，影響範囲が明らかである場合や，拡張前の挙動と下位互換性を持つように丁寧に設計されたクラス拡張などがあげられる．

ただし，この種のクラス拡張には method seals を用いる必要がない．利用者にとって本当に安全であるという確信があるクラス拡張であれば，グローバルスコープや動的スコープ[*3] を用いればよい．安全性よりも利便性を優先したいというニーズは現実には往々にしてあることで，method seals のような安全よりの仕組みとオープンクラスのような利便性よりの仕組みを使い分けていくことが現実的である．

3.3.2　`using` 間の干渉

複数の `using` 宣言が干渉しあうとき，本章で示したセマンティクスだけではその安全性を保証できないことがある．Method seals では unseal によって既知のパッケージを宣言するが，一度既知と宣言されたパッケージが，他のクラス拡張によって後から振舞いが変化した場合，もはやそのパッケージは既知とはいえないのではないかという懸

[*3] ただし Ruby には動的スコープは存在しない．

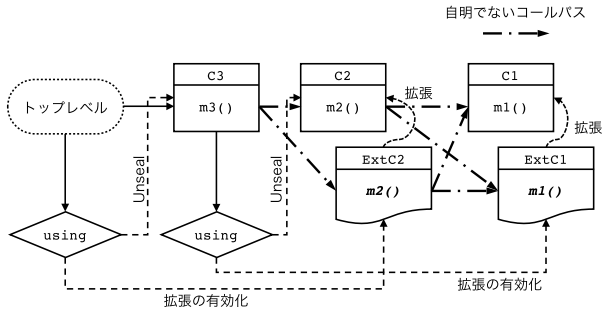

図 12　複数の using 宣言間の干渉
Fig. 12　Interference among multiple using declarations.

念がある．たとえば図 **12** において，クラス C3 の設計者はクラス拡張 ExtC1 を用いてクラス C1 を拡張している．このとき，クラス C2 を unseal している．しかし，クラス C3 の利用者はさらにクラス拡張 ExtC2 を用いてクラス C2 を拡張している．すでに提示したセマンティクスに基づくと，それぞれの using は正しく unseal パッケージを指定しているため，すべて拡張されたメソッドが呼び出されることになる．しかし，これはもとのソースコードを読んだことにより既知と宣言されたクラス C2 の振舞いが後から変化してしまうことになり，「知らないところ」に拡張の影響が及ぶことを避けるという本研究の目標が達成できていないといえる．

この問題に対応する 1 つの方法としては，seal/unseal の対象にクラス拡張そのものも含める方法が考えられる．図 12 の例に即して述べると，トップレベルの using において，クラス拡張 ExtC1 を unseal 対象に入れない限り，クラス拡張 ExtC2 が有効ならないようにする，ということである．ExtC1 を unseal するということは，ExtC2 の利用者は ExtC1 の実装について理解しているという前提が生まれ，ひいては ExtC1 と ExtC2 の相互作用によって C2 にどのような影響を与えるかの理解があるということになる．したがって，クラス拡張 ExtC2 を有効にしても問題ない，という考え方である．この方法の問題としては，クラス拡張の数が増えていくと，unseal しなければならないクラス拡張の数も増えていき，過剰に複雑化する恐れがある点があげられ，まだ議論の余地があるといえる．

3.3.3　unseal されたパッケージに対するコードの変更

一度 unseal された後で，そのパッケージのソースコードが別のプログラマによって変更されてしまった場合，そのパッケージはもはや既知とはいえないという懸念がある．この問題は 3.3.2 項で述べたものとよく似ている．ただし，3.3.2 項の問題とは異なり，この問題はソースコードそのものが静的に変化してしまうケースであり，より対処が難しい．ただし，この問題は本研究特有の問題ではなく，aspect-oriented programming や context-oriented programming など，クラス拡張に類する言語機構は，外部から既存のクラスに対して変更を加えるという性質上，全般的に拡張対象のコード変更に対して弱い．しかしながら，method seals ではコールパス上に現れるパッケージの構成にまで依存しているので，特にコードの変更の影響を受けやすいといえる．

この問題について，現時点で明確な対策はないが，既存のコードベースに対してハッシュ値を取ってそれをもとに変更を検知することなどの対策が考えられる．ただし，ソースコードの変更は実行時以前の出来事であるから，これを言語内に完結した機能として実現することは困難であり，外部ツールによる支援に頼る方法が現実的である．Aspect-oriented programming（AOP）の分野では，クラス拡張（アスペクト）の有効範囲を可視化するなどして AOP によるプログラミングを支援する開発環境についての研究が多数提案されている [10], [11], [12]．同様のアプローチを用いてクラス拡張の有効範囲の視覚化するなどして，こうした問題を防止することも考えられる．

4.　Ruby 処理系の拡張とその性能評価

本研究では，3 章で提案した method seals を Ruby 上に実装した．最も一般的に使われている Ruby の公式処理系である MRI（Matz' Ruby Implementation）を拡張することでこれを実現している．ベースとなっているバージョンは Ruby 2.1.4（r48285）である．また，この拡張した処理系に対して性能評価実験を行った．

4.1　実装

我々の実装では 3 章で示したアルゴリズムをほぼそのまま素朴に実装している．クラス拡張を有効にしてから拡張対象のメソッドが呼ばれるまでの処理の流れは大まかに次のとおりである．

(a) using 宣言でクラス c に対するクラス拡張 e を有効化．このとき，e と unseal パッケージのリストに関する情報を現在のレキシカルスコープに保存する．
(b) メソッド呼び出しが行われると新たにスタックフレームを作成し，これをコールスタックに積む．このとき，
(b1) 現在のレキシカルスコープまたはスタックフレームに e 関する情報が存在するか調べ，存在する場合はさらに呼び出すメソッドが属するパッケージが unseal パッケージリストに含まれているかについて調べる．含まれている場合は新たにコールスタックに積むスタックフレームに e に関する情報を埋め込む．以降，拡張対象のクラス c に到達するまで (b) を繰り返す．
(b2) e に関する情報が見つからないか，呼び出すメソッドが属するパッケージが unseal パッケージリストに含まれていない場合はスタックフレームには e に関する情報は埋め込まれない．
(c) c のメソッドが呼び出される．このとき行われるメソッドルックアップにおいて，現在のスタックフレー

表 1　クラス拡張を用いないメソッド呼び出しのパフォーマンス
Table 1　The method call performance without class extensions.

	イテレーション数/秒	標準偏差
標準処理系	9.280×10^6	2.7%
拡張処理系	9.038×10^6	1.2%

ムに e に関する情報があった場合は e を有効にする．

4.2　実装上の制限事項

3 章で述べたように，現在の我々の実装ではクラスのみがパッケージとして指定できる．今後，モジュールやメソッドをパッケージとして指定できるように実装を拡張することで，より柔軟な粒度でクラス拡張の有効範囲を制御できるようになるはずである．

また，Ruby のコールスタックを用いて実装しているため，マルチスレッドプログラミングとの親和性は低い．現在の実装では親スレッドが保持する method seals に関する情報を子スレッドに引き継がないため，クラス拡張を別スレッドで有効にすることができない．

4.3　性能測定

Method seals ではメソッド・ルックアップやスタックフレームの拡張を行っており，パフォーマンスへの影響が広範に及ぶと予測されるため，これを計測するための実験を行った．実験環境は CentOS release 6.2, Intel(R) Xeon(R) CPU E5-2687W 3.10 GHz，メモリは 64 GB である．

4.3.1　通常のメソッド呼び出しにおけるオーバヘッド

まず，method seals を用いない場合のメソッド呼び出しのオーバヘッドを計測した．空のメソッドを 10 秒間繰り返し呼び出し，1 秒あたりのイテレーション回数の平均を計算した．この結果を**表 1** に示す．この結果から，我々が拡張した処理系には，3%程度のオーバヘッドが存在することが分かった．Method seals では新たにスタックフレームを積む際に毎回，現在クラス拡張が `using` されていないかを確認し，必要に応じて前のスタックからクラス拡張に関する情報を引き継ぐ処理を行う．これにより，method seals を使わない場合も含め，すべてのメソッドコールにオーバヘッドが発生する．

4.3.2　クラス拡張を用いた場合のメソッド呼び出しのオーバヘッド

次に，クラス拡張を利用した場合のメソッド呼び出しのオーバヘッドを計測した．空のメソッドに対し，クラス拡張を用いて同様に空のメソッドに再定義し，この再定義されたメソッドを繰り返し呼び出すことでパフォーマンスを計測した．またこれを，オープンクラスを用いた場合，refinements を用いた場合，method seals を用いた場合のそれぞれについて計測した．Method seals を用いる場合については最終的に拡張されたメソッドが呼ばれる場合（unsealed）と，拡張前のメソッドが呼ばれる場合（sealed）の 2 通りについて計測した．

表 2　クラス拡張を用いたメソッド呼び出しのパフォーマンス
Table 2　The method call performance with class extensions.

	イテレーション数/秒	標準偏差
Open class	9.162×10^6	0.9%
Refinements	9.220×10^6	1.9%
Method seals (unsealed)	4.628×10^6	0.3%
Method seals (sealed)	7.099×10^6	0.4%

この計測結果を**表 2** に示す．この結果が示すように，オープンクラスと refinements を用いた場合では表 1 に示したクラス拡張を用いない場合とくらべてもほとんど実行速度に変化が見られない．一方，method seals を用いた場合はそうでない場合に比べて大きく性能が劣化している．特に，最終的に拡張されたメソッドが呼び出される場合は，2 倍以上のオーバヘッドが存在する．これはコールパス上の全パッケージと unseal されたパッケージとの比較処理のオーバヘッドが大きいことによると思われる．ただし，我々の実装では，`using` 呼び出しの時点で unseal されたクラスには印を付けておき，明らかに seal されているパッケージがコールパス上に現れたらそれ以降 method seals に関する処理のほとんどを行わないようにしている．拡張前のメソッドが呼び出される場合ではいくらかパフォーマンスの劣化が穏やかなのは，このことが有効に働いていると思われる．

また，method seals を用いた場合に共通して現れるオーバヘッドは，インラインメソッドキャッシュの有無によるものが大きいものと思われる．Ruby（MRI）には前回のメソッド・ルックアップの結果をキャッシュすることによってメソッドディスパッチを高速化する仕組みがある．クラスが再定義された場合にステートカウンタをインクリメントし，このステートカウンタがキャッシュが作成されたときのものと同じである限りそのキャッシュが利用され続ける．しかしながら，method seals ではクラスが再定義されない場合でもたどったコールパス次第で呼び出すべきメソッドが変化しうるため，既存のアルゴリズムでは正しいキャッシュが読み込まれないことがある．そこで我々の現状の実装では拡張対象のメソッドに関してのみインラインメソッドキャッシュを無効にしている．

実際にインラインメソッドキャッシュの影響を調査するため，全体としてインラインメソッドキャッシュを無効にしたうえで同様の計測を行った結果を**表 3** に示す．表 2 の結果と比較すると，オープンクラスと refinements に関しては大きくパフォーマンスが劣化した一方，method seals を用いた場合では比較的小さな変化にとどまった．今回のベンチマークプログラムでは，再定義されたメソッドの呼び

表 3　クラス拡張を用いたメソッド呼び出しのパフォーマンス（インラインキャッシュ無効時）

Table 3　The method call performance with class extensions (Inline method caches are disabled).

	イテレーション数/秒	標準偏差
Open class	7.954×10^6	1.5%
Refinements	6.392×10^6	1.1%
Method seals (unsealed)	4.416×10^6	0.6%
Method seals (sealed)	6.720×10^6	0.3%

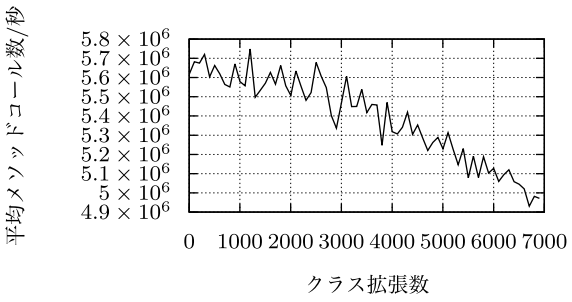

図 13　クラス拡張数によるメソッド呼び出しパフォーマンスの変化

Fig. 13　Transition of the method call performance by the number of class extensions.

出しのみを計測対象としているため，method seals を用いた例ではインラインメソッドキャッシュの恩恵をほぼまったく受けていなかった．したがって，全体としてインラインメソッドキャッシュを無効にしても結果に変化がないのは期待どおりといえる．また，refinements はインラインメソッドキャッシュを有効にしていた場合ではオープンクラスに匹敵する性能を見せていたが，インラインメソッドを無効にした場合ではオープンクラスよりも大きく劣る結果となっている．これは元々 refinements によって拡張されたメソッドの呼び出しは一般のメソッド呼び出しよりもコストが大きく，そのためインラインメソッドキャッシュの恩恵がより大きかったものと思われる．

4.3.3　同一コールパス上のクラス拡張数によるパフォーマンスの変化

using によって有効化されたクラス拡張が同一コールパス上に複数現れる場合，クラス拡張ごとに処理を行う必要があるため，クラス拡張が増えれば増えるほどパフォーマンスが劣化することが予測される．これを確認するため，同一コールパス上のクラス拡張の数を変化させたときに観測されるパフォーマンスの変化を計測した．

この実験では，指定した数のクラス拡張を用いるベンチマークプログラムを自動生成し，生成したプログラムごとにパフォーマンスを測定した．各プログラムの測定についてはこれまでの条件と同様である．

この実験によって得られた結果を図 **13** に示す．クラス拡張数が増えるにつれ，パフォーマンスが低下する傾向にあることが分かる．ただし，一般に同一コールパス上のクラス拡張の数はそこまで大きくならず，我々の想定するユースケースでは多い場合でも 100 に満たないと思われる．クラス拡張が 1000 ある場合でも 10%に満たないオーバヘッドであり，クラス拡張数の増加による速度低下が実用上問題になることは考えにくい．

5.　関連研究

クラス拡張に類する言語機構には様々なものがある．必ずしも本研究のようにクラス拡張の有効範囲制御を主目的とはしていないが，多くの場合，提供される機能の一部として有効範囲の制御を行う仕組みが含まれている．

先述したように，Ruby や MultiJava [13] の Open classes をはじめとするグローバルに有効になるタイプのクラス拡張は，その影響範囲の認識性が低く，意図しない動作を引き起こしやすい．

また，Ruby の refinements や selector namespaces [8] のようにレキシカルスコープ下にクラス拡張の有効範囲を制限するものは，影響範囲の認識性が高く，意図しない動作を引き起こす可能性が低いが，一方で複数のスコープをまたいで間接的に呼ばれるメソッドを拡張することはできず，利便性に難がある．

Context-oriented programming（COP）[4] は文脈に依存したクラスの振舞いをモジュールとして切り出し，実行時に文脈に応じて振舞いを切り替えるプログラミングパラダイムである．COP では，文脈依存の振舞いは間接的に呼び出されたメソッドに対しても有効だが，method seals のように影響範囲の認識性を向上させる仕組みはなく，文脈の切替えがプログラム全体にどのような影響を与えるかを把握することは容易ではない．

AspectJ [14] は aspect-oriented programming [3] のための Java 向けの拡張である．AspectJ ではクラス拡張（advice）の有効範囲を指定するための手段（pointcut）を多数提供しており，非常に柔軟な有効範囲の制御が可能である．AspectJ の提供する `cflow` オペレータを用いることで，Method seals のようにコールスタックの状態に基づいて拡張を切り替えることもできる．ただし，method seals と同等のセマンティクスを得るためには，複雑な条件式を自身で書く必要があり，また影響範囲の認識性を保証する仕組みはない．

Classboxes [9] では `classbox` というモジュール内にクラスとクラス拡張を定義し，その中にクラス拡張の影響を限定する．また，他の `classbox` のクラス拡張やクラスをインポートして利用することもできる．さらに，*local rebindings* と呼ばれる仕組みにより，間接的に呼び出されたメソッドに対しても拡張が有効になる．ただし，`classbox` 内で有効にしたクラス拡張はその `classbox` 内ではつねに有効になるため，グローバルスコープや動的スコープと同様に，その影響範囲の認識性には難がある．

Method shelters [15] は本研究の前身研究の 1 つであ

る．Classboxes とよく似ており，methodshelter というモジュール内にクラスとクラス拡張を定義することができ，さらに他の methodshelter をインポートすることができる．Classboxes と違う点は，methodshelter 内が *exposed chamber* と *hidden chamber* と呼ばれる 2 つの領域に分かれており，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．これにより，classboxes よりも細かい粒度でクラス拡張の有効範囲を制御できるようになったが，この制御はあくまで methodshelter の定義時に行うものであり，外部から制御できない．したがって，いくらかリスクは軽減しているものの，依然として影響範囲の認識性に難がある．

Method shells [16], [17] は本研究のもう 1 つの前身研究であり，method shelters とは異なりインポートする側が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．しかし，method shells は 2 種類のインポート宣言によって構築されたモジュール構造に基づき複雑なメソッド・ディスパッチを行うため，実際にどのメソッドが呼ばれるかを把握するためにはモジュール構造を正確に把握する必要があり，やはりクラス拡張の影響範囲の認識性が高いとはいい難い．

なお，本研究で提案する method seals の基本的なアイディアは既発表のものであるが [18]，本論文では既発表の内容に加え，論旨をより明確にし，さらに method seals の限界に関する議論や新たな処理系の実装とその実験についての報告を行った．

6. まとめ

本研究では，コールパス上の未知のパッケージの有無によりクラス拡張の有効範囲を制御する言語機構 method seals を提案した．クラス拡張はコードの再利用性を高めるが，外部からクラスを拡張するというその性質上，その影響範囲の予測が難しく意図しない動作を引き起こしやすいという問題があった．Method seals では，クラス拡張が未知のコードに影響を及ぼすことによって起こりうる潜在的な問題を回避するために，クラス拡張の影響範囲を既知のコールパスに限定する．未知のコードに対してクラス拡張が影響を及ぼすことがないため，意図しない動作を引き起こす危険性を軽減できる．さらに本機構を Ruby 処理系を拡張することで実装し，その性能の測定を行った．

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はかるなどがあげられる．また，3.3 節で述べたような method seals における制限事項について何らかの対策を講じる必要がある．

参考文献

[1] Flanagan, D. and Matsumoto, Y.: *The ruby programming language*, O'Reilly Media, Inc. (2008).

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

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

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

[5] Hejlsberg, A., Wiltamuth, S. and Golde, P.: *C# Language Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003).

[6] Paepcke, A.: *Object-oriented programming: the CLOS perspective*, MIT press (1993).

[7] Pinson, L.: *Object-Oriented Programming with Objective-C*, Addison-Wesley Longman Publishing Co., Inc. (1991).

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

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

[10] Clement, A., Colyer, A. and Kersten, M.: Aspect-oriented programming with AJDT, *ECOOP Workshop on Analysis of Aspect-Oriented Software* (2003).

[11] Hon, T. and Kiczales, G.: Fluid AOP Join Point Models, *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA, pp.712–713, ACM (online), DOI: 10.1145/1176617.1176687 (2006).

[12] Bragdon, A., Zeleznik, R., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, Jr., J.J.: Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance, *Proc. SIGCHI Conference on Human Factors in Computing Systems, CHI '10*, pp.2503–2512, ACM (online), DOI: 10.1145/1753326.1753706 (2010).

[13] Clifton, C., Leavens, G.T., Chambers, C. and Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, *SIGPLAN Not.*, Vol.35, No.10, pp.130–145 (online), DOI: 10.1145/354222.353181 (2000).

[14] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An overview of AspectJ, *ECOOP 2001—Object-Oriented Programming*, pp.327–354, Springer (2001).

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

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

[17] 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, No.3, pp.12–21 (2014).

[18] 福室　嶺，千葉　滋：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，日本ソフトウェア科学会第 32 回大会予稿集 (2015).

福室 嶺

2014 年早稲田大学基幹理工学部情報理工学科卒業．2016 年東京大学情報理工学系研究科創造情報学専攻修了．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 年同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）．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の研究に従事．