情報処理学会論文誌　プログラミング　Vol.5　No.3　1–22 (Aug. 2012)

CastOff：Ruby用コンパイラのライブラリとしての実装

芝 哲史[1,†1,a)]　笹田 耕一[1,†2]　平木 敬[1]

受付日 2011年12月16日, 採録日 2012年3月29日

概要：近年，様々なスクリプト言語処理系に対してコンパイラが開発されている．これらのコンパイラは処理系に組み込む形で実装されることが多く，新しくコンパイラを開発するために，処理系そのものの再実装や，処理系の大幅な改変が行われている．このため，スクリプト言語処理系に対する新たなコンパイラの開発には多大な労力をともなう．そこで我々は，スクリプト言語 Ruby の C によって実装された処理系（CRuby）の機能を活用することで，処理系に対して新たに手を加えることなく動作するコンパイラ CastOff を開発した．CastOff は，実行時コンパイル，コンパイル済みコードの再利用，プロファイル実行，アノテーションのサポート，脱最適化，再コンパイルなどの機能を持つ．これらの機能を，CastOff は CRuby の C による拡張ライブラリ（C 拡張）として CRuby にいっさいの変更を加えずに実現している．本稿では CastOff の設計と実装を述べ，CastOff の機能を Ruby の C 拡張でどう実現したかを詳細に解説する．そして，CastOff のようにライブラリとしてコンパイラを実装するために，どのような機能が必要かを議論する．

キーワード：スクリプト言語，言語処理系，コンパイラ，Ruby

CastOff: A Compiler for Ruby Implemented as a Library

Satoshi Shiba[1,†1,a)]　Koichi Sasada[1,†2]　Kei Hiraki[1]

Received: December 16, 2011, Accepted: March 29, 2012

Abstract: There are many compilers for scripting languages aiming at faster execution. In most cases, these compilers have been developed by modifying runtime system or by re-implementing runtime system. Therefore a large amount of development cost is needed to develop compilers for scripting languages. On this background, we developed CastOff, a compiler for the Ruby scripting language. By using features in Ruby, we developed CastOff without modifying the Ruby runtime system. CastOff has a lot of functions such as runtime compilation, reuse of compiled codes, profiling execution, annotation support, deoptimization and re-compilation. CastOff is a Ruby compiler with the above features provided as a Ruby's C extension library. In this paper, we show the detailed design and implementation of CastOff, how we implemented functions of CastOff by Ruby's C extension library. Moreover, we discuss functions required in the runtime system for developing a compiler as a library like CastOff.

Keywords: scripting language, programming language implementation, compiler, Ruby

1. はじめに

近年，スクリプト言語の持つ高い生産性から，様々な場面でスクリプト言語が活用されている．スクリプト言語の評価器はインタプリタとして実装されていることが多く，高い生産性を持つ反面，C や Java などの言語と比較して実行速度が低いという問題点がある．これに対し，スクリプト言語を高速に実行するためのコンパイラが，様々な言語

[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
[†1] 現在，株式会社ドワンゴ
Presently with DWANGO Co., Ltd.
[†2] 現在，Heroku, Inc.
Presently with Heroku, Inc.
[a)] shiba@rvm.jp

© 2012 Information Processing Society of Japan

に対して新たに実装されている [10], [11], [12], [13], [14].

新たに開発されるスクリプト言語用コンパイラの多くは，処理系そのものを再実装するという形で実装されている [10], [12]．これらのコンパイラは，処理系をコンパイラにあわせて開発できることから，適用できる最適化の幅が広い．しかし，標準ライブラリや GC など，様々な機能を新たに実装する必要があり，開発にかかるコストが高い．

一方で，既存の処理系を活用するという形で実装されたコンパイラも存在する [3], [13], [14]．これらのコンパイラは，生成したコードを既存の処理系の上で動作させるため，低い開発コストでほぼ完全な互換性を得ることができる [3]．特に，Python 用コンパイラである Psyco [13] や PHP 用コンパイラである PHC [14] は，処理系にいっさいの変更を加えることなくコンパイラを実装しており，既存の実行環境にほぼ影響を与えることなく，容易に導入することができる．

処理系に手を加えることなくコンパイラを開発する場合，対象とする処理系が提供する機能だけを用いてコンパイラで提供したい機能を実現する必要があるため，コンパイラの実装手法が処理系の提供する機能に強く制限される．このため，処理系が提供する機能が不十分な場合，実装手法の工夫や機能面，互換性面での妥協が必要になる．

我々は，スクリプト言語 Ruby [1] の C によって実装された処理系（以降 CRuby と呼ぶ）に対し，CastOff [2] というコンパイラを開発している．既存の実行環境に与える影響を小さくすることで，CastOff を手軽に導入できるものとするため，CastOff は処理系に対して新たに手を加えることなく，CRuby の C による拡張ライブラリ（以降 C 拡張と呼ぶ）として実装している．CastOff は，実行時コンパイル，プロファイル実行，アノテーションのサポート，脱最適化，再コンパイル，コンパイル済みコードの再利用などの機能を持つ．これらの機能を，CastOff は CRuby にいっさいの変更を加えることなく実現している．これにより，CastOff がサポートするバージョンの Ruby 処理系がインストールされた状態ならば，CastOff の導入は既存の実行環境にほぼ影響を与えることなく，容易に行うことができる．

本稿におけるコンパイラのライブラリとしての実装とは，CastOff のように，処理系にいっさい手を加えることなく実装されたコンパイラを指す．本稿では，CastOff の設計と実装を述べることで，CastOff が持つ機能を Ruby の C 拡張でどのように実現するかを詳細に解説する．そして，CastOff のようにライブラリとしてコンパイラを実装するために，処理系にどのような機能が必要かを議論する．

本研究の貢献は 3 つある．まず，スクリプト言語処理系に手を加えることなくコンパイラを開発するうえで処理系にどのような機能が必要になるかを，CastOff という実例を用いて議論したという点である．処理系に手を加えることなく実装されているスクリプト言語用コンパイラは，CastOff 以外にも存在するが，これらの先行研究では，処理系にどのような機能が必要になるかの議論がなされていない [13], [14]．次に，広く利用されているプログラミング言語 Ruby に対し，CastOff が提供する機能の実現手法を詳細にまとめた点である．CastOff と同様に，Ruby 処理系の機能を活用する Ruby 用コンパイラはすでに存在する [3], [4], [5]．CastOff はこれらのコンパイラをさらに発展させ，実行時コンパイルや再コンパイル，ユーザからのアノテーションをサポートしている．最後に，Rubygems [20] という Ruby が組み込みで持つパッケージ管理ツールを用いて，研究成果である CastOff を広く利用可能としたという点である．これは，本研究の実社会への貢献としてあげることができる．

本稿の構成を以下に示す．まず，2 章で CastOff の機能と挙動を解説し，CastOff の概観を示す．次に，3 章で CastOff が提供する各機能の設計，4 章で CastOff の実装について述べる．特に 4 章では，3 章で述べた設計をどのようにして C 拡張として実現したかを解説する．そして，5 章で CastOff のように処理系に手を加えることなくスクリプト言語用コンパイラを開発するためには，処理系がどのような機能を提供すべきかを議論する．6 章で CastOff の性能を評価し，7 章で関連研究を述べ，8 章でまとめる．

2. CastOff の概観

CastOff は，CRuby のライブラリとして実装した，Ruby から C へのコンパイラである．CastOff は，Ruby 処理系にいっさい手を加えることなく，コンパイラとしての機能を提供する．本章では，CastOff の機能や挙動を解説し，CastOff の概観を示す．

2.1 CastOff の機能

CastOff は，ユーザから与えられた情報（以降アノテーションと呼ぶ）や，プロファイル実行によって得た情報を用いてコンパイルを行う．アノテーションやプロファイルによって得た情報が間違っていることが分かった場合は，脱最適化や実行時の再コンパイルを行う．CastOff が提供する機能を以下に列挙する．CastOff は，これらの機能を CRuby 処理系にいっさいの変更を加えることなく提供する．

- 実行時コンパイル
- プロファイル実行
- 脱最適化 [18]
- 再コンパイル
- アノテーションのサポート
- コンパイル済みコードの再利用

CastOff は現在，Rubygems.org で公開している [2]．これにより，CastOff は Rubygems という Ruby 処理系組み

© 2012 Information Processing Society of Japan

表 1　cast_off の使い方
Table 1　Usage of cast_off.

	機能	cast_off への引数
(a)	対象プログラムのコンパイル	cast_off –threshold=閾値 対象プログラム 対象プログラムの引数
(b)	コンパイルしたコードの実行	cast_off –run 対象プログラム 対象プログラムの引数

```
# fact.rb
def fact(i)
  i > 1 ? (i * fact(i - 1)) : 1
end
fact(ARGV.shift.to_i)
```

図 1　サンプルスクリプト
Fig. 1　Sample script.

```
require 'cast_off'

def fact(i)
  i > 1 ? (i * fact(i - 1)) : 1
end
CastOff.compile_singleton_method(
  self, :fact, binding, :i => Fixnum
)
fact(ARGV.shift.to_i)
```

図 2　アノテーションのサポート
Fig. 2　Annotation support.

込みのパッケージ管理ツールを通して，容易にインストールすることができる．具体的には，CastOff がサポートするバージョンの Ruby 処理系がインストールされた状態ならば，`gem install cast_off` とコマンド入力するだけで CastOff をインストールすることができる．CastOff は CRuby 処理系に手を加えることなく動作するため，ユーザはプログラミング環境を変えることなく，容易に CastOff を導入し，CastOff が提供する機能を活用することができる．

2.2　CastOff の挙動

本節では，表 1 と図 1 の階乗計算のサンプルプログラムを用いて，CastOff の挙動を解説する．CastOff のプロファイラやコンパイラは Ruby のライブラリとして実装されている．ユーザは `cast_off` という Ruby で実装されたコマンドラインツールを利用し，CastOff に対し，プロファイル実行やコンパイル処理を指示する．

まず，CastOff へのコンパイル指示は表 1 (a) のように，コマンドラインツール `cast_off` へコンパイル対象のプログラムとその引数をわたすことで行う．たとえば，図 1 の fact.rb のコンパイルを指示する場合，`cast_off --threshold=5 fact.rb 10` とする．このように実行すると，CastOff は指定された引数である 10 という値を用いて fact.rb を実行し，プロファイル情報を収集する．そして，プロファイル実行が終わった後に，閾値として設定した 5 回以上実行されたメソッドをコンパイルする．今回の例では，`fact` メソッドが 5 回以上実行されるため，コンパイルの対象となる．`fact` メソッドのコンパイルでは，CastOff はプロファイル実行で収集した情報から次のような前提条件を設定し，コンパイルを行う．

前提条件 (1)：`fact` メソッドの返り値は `Fixnum` オブジェクト

前提条件 (2)：`fact` メソッドの第 1 引数 `i` は `Fixnum` オブジェクト

コンパイルを終えた後は，表 1 (b) のように指定することで，CastOff がコンパイルを済ませたコードを用いて対象プログラムを実行することができる．今回の場合，`cast_off --run fact.rb 10` とすることで，コンパイル済みのコードを読み込んで fact.rb を実行することができる．引数を 10 として実行した場合，コンパイル時と同じ挙動をするため，上記の前提条件 (1), (2) はともに崩れない．このため，コンパイル済みコードは問題なく実行できる．

対象プログラムがプロファイル情報を取得したときと異なる挙動をとり，コンパイルの前提条件が崩れた場合，CastOff は脱最適化を行い，コンパイルの入力となったオリジナルのコードへと実行を切り替える．脱最適化によって，コンパイルの前提条件が崩れたときも実行を継続することができる．たとえば，`cast_off --run fact.rb 100` と実行すると，fact メソッドは実行の途中で Bignum オブジェクト（Ruby における多倍長整数オブジェクト）を返す．このため，コンパイル時に定めた前提条件 (1) が崩れてしまい，コンパイル済みコードによる実行を継続することができなくなる．このとき，CastOff はコンパイルして得た `fact` メソッドから，コンパイル前の（オリジナルの）`fact` メソッドへと実行を切り替えることで脱最適化を行い，`fact` メソッドの実行を継続する．

CastOff は脱最適化と同時に再コンパイルを行う．脱最適化が生じると，オリジナルのコードへと実行が切り替えられるため，コンパイルを行ったことによる速度向上がなくなるだけでなく，オリジナルのコードへと実行を切り替えるオーバヘッドが生じる．このため，CastOff は脱最適化が生じる回数を減らすために，新たに前提条件を設定して再度コンパイルを行う．今回の場合，前提条件 (1) を次のような前提条件 (1') へと再設定する．これにより，もう 1 度 `cast_off --run fact.rb 100` を実行しても，再コンパイルによって生成されたコードが読み込まれ，脱最適化は発生しない．

© 2012 Information Processing Society of Japan

前提条件 (1')：fact メソッドの返り値は Fixnum もしくは Bignum オブジェクト
前提条件 (2)：fact メソッドの第 1 引数 i は Fixnum オブジェクト

また，ユーザは CastOff に対し，**図 2** のようにコンパイルを直接指示することができる．図 2 では，fact メソッドの定義直後に，fact メソッドを変数 i が Fixnum であるという前提の下でコンパイルするよう，CastOff に指示している．単一のメソッドのみをコンパイルしたい場合や，プロファイル実行を行わずにメソッドをコンパイルしたい場合は，このようにスクリプトの内部で直接コンパイルを指示する．

3. 各機能の設計

本章では，CastOff が提供する，実行時コンパイル，プロファイル実行，アノテーションのサポート，脱最適化，再コンパイル，コンパイル済みコードの再利用という機能それぞれの設計を述べる．まず，CastOff になぜこれらの機能を組み込んだかを述べ，次に個々の機能の設計を解説する．

本稿の主眼はこれらの設計を Ruby 処理系に対して適用する際にどのような工夫を行い，どのような機能が必要であったかを議論することである．CastOff の実装上の工夫は 4 章で詳細を述べ，どのような機能が必要であったかの議論は 5 章で行う．

CastOff は，スクリプト言語 Ruby 用のコンパイラである．Ruby は動的言語であるため，コンパイル時に考慮すべき様々な条件（変数の型や呼び出すメソッド）が，実行時に定まる．このため，Ruby 用のコンパイラとして高速なコードを生成するには，実行時に定まるコンパイル対象の挙動を考慮する必要がある．まず，高速化のためには，他の言語処理系 [10], [12] が行っているようにコンパイル対象が扱う型に関する情報を取得し，静的型付け言語が行うような最適化 [17] を適用することが望ましい．次に，実用性を向上させるためには，生成したコードの速度だけではなく，コンパイルにかかる時間にも配慮する必要がある．

当然，ユーザが CastOff を利用する目的は高速化にあるため，コンパイル時間の短縮は，速度を犠牲にすることなく行わなければならない．速度とコンパイル時間を両立させるには，すでに存在する数多くの JIT コンパイラ [10], [12], [13] が行っているように，実行頻度の高い箇所のみを変換するべきである．

CastOff は，出現する型や実行頻度などに関する情報を取得するために，プロファイル実行や，ユーザからのアノテーションをサポートする．なお，コンパイル対象の情報を得るための手法として，静的解析も考えられる．Ruby では，広く利用されているプログラムにおいても，Ruby 処理系の組み込みメソッドを上書きし，プログラムの意味を変化させる．たとえば，RubyGems [20] という Ruby において広く利用されているパッケージ管理ツールでは，処理系組み込みである require メソッドを自身が定義したものに上書きする [16]．このため，プログラムの意味が動的に変化する Ruby において，静的解析によって十分な情報を得るのは困難であると考え，CastOff では静的解析を用いずに，プロファイル情報とユーザからのアノテーションを用いてコンパイルを行う．

CastOff は，コンパイル時間を削減するために，なるべくコンパイル済みのコードを再利用する．コンパイル対象に変化があった場合や，プロファイル情報などのコンパイル結果を左右する情報（以降，コンパイル条件と呼ぶ）が更新された場合は，生成済みのコードと異なるコードを生成する可能性があるため，新たにコンパイルを行う．コンパイル対象やコンパイル条件に変化がない場合，コンパイルは行わずに生成済みのコードを再利用する．

CastOff は，プロファイルやユーザからのアノテーションが間違っていた場合にも実行を継続するために，実行時に脱最適化 [18] を行う．脱最適化とは，最適化の前提条件が崩れたときに，最適化を行ったコードを無効化し，安全なコードへと実行を切り替える手法である．CastOff は，プロファイル情報やユーザからのアノテーションをコンパイルの前提条件に用いる．このため，プロファイル実行時に実行されなかった箇所が実行された場合や，ユーザの与えたアノテーションが間違っていた場合に，コンパイルの前提条件が崩れ，間違った実行が行われてしまう可能性がある．これに対処するため，CastOff は脱最適化をサポートする．

最後に，CastOff は，脱最適化のオーバヘッドを削減するために，実行時の再コンパイルをサポートする．脱最適化を行う場合，脱最適化の対象となるコードから，脱最適化の適用後に実行する安全なコードへの切替えコストが発生する．速度向上という目標のためには，このような脱最適化のオーバヘッドは好ましくない．このため，CastOff は脱最適化の適用時に，脱最適化が繰り返し行われることがないよう実行時の再コンパイルを行う．

図 3 に, CastOff の全体像を示す．本章では以降, CastOff が提供する機能それぞれの設計を述べる．なお，本章では，CastOff の設計を Ruby 処理系に依存しない形で記述する．本章の設計を，Ruby 処理系が提供する機能でどう実現したかは，4 章で解説する．

3.1 実行時コンパイル

本節では，**図 4** を用いて，CastOff の実行時コンパイル機構の設計を解説する．コンパイル対象の具体例や，どのようにしてこれらの処理を実行時に行うのかは，4.1 節で解説する．

まず，(1) ユーザがコマンドラインツールなどのイン

© 2012 Information Processing Society of Japan

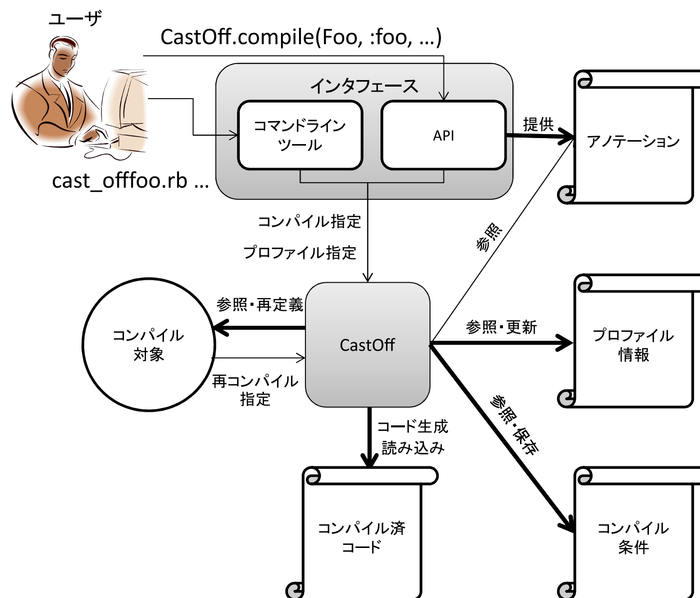


図 3　CastOff の全体像
Fig. 3　Overview of CastOff.

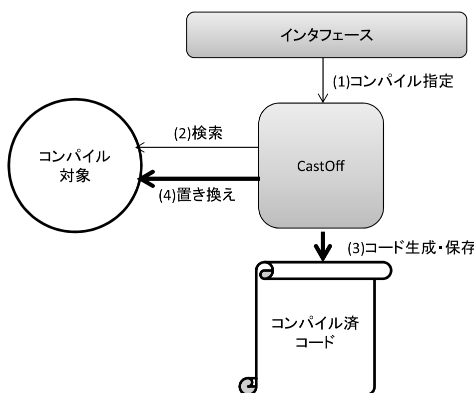


図 4　実行時コンパイルの設計
Fig. 4　Design of runtime compilation.

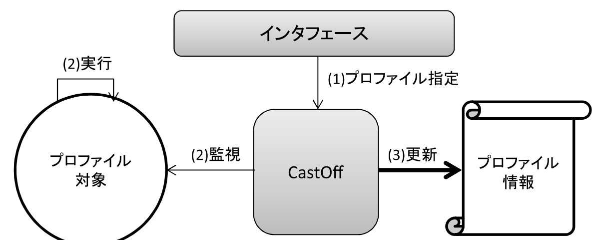


図 5　プロファイル実行の設計
Fig. 5　Design of profiling execution.

タフェースを通して CastOff にコンパイルを指示すると，CastOff に対し，コンパイル対象（メソッドや，ソースファイルなど）の識別子（クラスとメソッド名の組や，ファイル名と行番号の組など）がわたされコンパイルが指示される．すると，(2) CastOff はわたされた識別子を用いてコンパイル対象を検索し，その定義を参照する．そして (3) コードを生成し，コンパイル済みコードの再利用（3.6 節）のために保存する．最後に，(4) コンパイル対象を生成したコードへ置き換える．

これらの処理は (2) の検索処理と (4) の置き換え処理を除けば，実行時に行うことは容易である．(2) の検索処理と (4) の置き換え処理は，コンパイル対象や識別子，および生成したコードのフォーマット（機械語やバイトコード列など）次第で処理系のサポートが必要となる．以下，コンパイル対象の単位がメソッドである場合を例に解説する．

まず，コンパイル対象の検索処理の場合，識別子がファイル名と行番号の組ならば，ファイルを検索し指定された行を参照するだけでよいため，処理系のサポートは特に必要ない．しかし，識別子がクラスとメソッド名の組であった場合，処理系がクラスとメソッド名からのメソッド検索の機能を提供している必要がある．

次に，コンパイル対象の置き換え処理の場合，生成したコードのフォーマットが対象とする言語のソースコードならば，生成したコードをそのまま処理系に読み込ませればよい．実行時のコードの読み込みは，多くのスクリプト言語処理系がサポートしている．しかし，生成するコードが対象とする言語のソースコードではない場合，生成するコードによるメソッド定義を，対象とする処理系がサポートしている必要がある．たとえば，生成するコードが機械語である場合，処理系が機械語によるメソッド定義をサポートしている必要がある．生成するコードによるメソッド定義を，対象とする処理系がサポートしていない場合，コンパイル対象の置き換え処理をライブラリとして実装することは困難である．

3.2　プロファイル実行

本節では，図 **5** を用いて，CastOff のプロファイル実行

© 2012 Information Processing Society of Japan

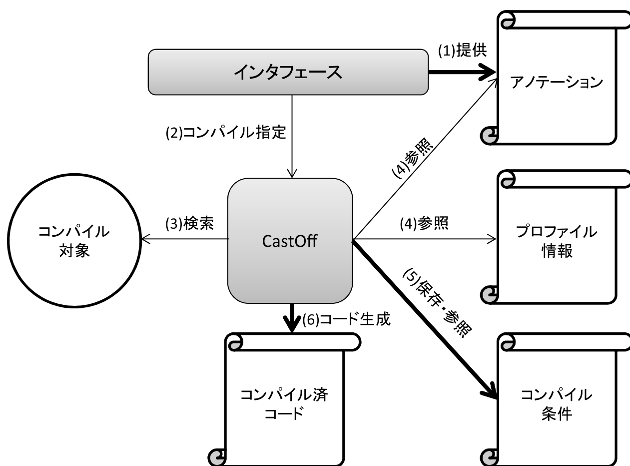


図 6　アノテーションの活用
Fig. 6　Utilizing annotation.

の設計を解説する．プロファイル用コードを具体的にどのように実行させるかは，4.2 節で解説する．

まず，(1) ユーザが CastOff にプロファイルを指示すると，(2) CastOff は，プロファイル用命令の挿入や，処理系が提供するイベントフックの API を用いてイベントハンドラを設定することで，対象プログラムを監視する．そして，対象プログラムを実行することで，挿入したプロファイル用命令や設定したイベントハンドラを通し，プロファイル情報を取得する．最後に，(3) 取得した情報を用いてプロファイル情報を更新する．プロファイル実行時に与えられた引数によって，対象プログラムの挙動が異なる可能性がある．このため，取得したプロファイル情報は追記式で管理し，プロファイル実行をまたいでプロファイル情報を蓄える．

本設計では，処理系が提供するイベントフックの API と，プロファイル対象コードの書き換えを併用する．参考文献 [8] では，スクリプト言語処理系が提供する API を活用し，パフォーマンスプロファイラを構築している．参考文献 [8] で用いている API はメソッドの起動，終了などの特定のイベントをフックするためのものであり，プロファイル対象の情報を取得できる箇所が，提供される API に制限される．このため，プロファイル対象の詳細な情報（たとえば特定の命令を実行する直前の特定の変数の型情報など）を取得するには不向きである．以上の理由により，十分な情報を得るためにはイベントフックの API だけでは不十分であると考え，プロファイル対象コードの書き換えによるプロファイル用命令の挿入も行う．

3.3　アノテーションのサポート

本節では，図 6 を用いて，CastOff のアノテーションのサポートに関する設計を解説する．ユーザは図 2 のように，CastOff が提供する API を通してアノテーションを提供し（図 6 (1)），CastOff はコンパイル時に提供されたアノテーションを参照する．

まず，(2) CastOff はコンパイルを指示されたとき，(3) コンパイル対象を検索し，その定義を参照する（3.1 節を参照）．次に，(4) プロファイル情報やユーザから与えられたアノテーションを参照し，コンパイル条件としてこれら 2 つの情報をとりまとめる．そして，(5) コンパイル済みコードの再利用（3.6 節）のために，コード生成を行う前にコンパイル条件を保存する．最後に，(6) コンパイル条件を用いてコード生成を行う．

```
        ...
        /* 変数 a が Fixnum オブジェクト
           であることを確認するためのガード */
(1)     if (!guard(local0_a, Fixnum)) {
          /* ガードによる検査に失敗 */
(2)       goto safe_code;
        }
        ...
        safe_code:
        /* 安全に実行できるコード */
        ...
```

図 7　ガードによる検査
Fig. 7　Check by guard.

3.4　脱最適化

本節では，CastOff の脱最適化機構の設計を解説する．脱最適化は，実装している最適化や処理系が提供する機能への依存度が大きいため，本節では脱最適化のタイミングと流れを簡単に解説するだけにとどめる．CastOff が脱最適化処理を具体的にどのように行うかは，4.4 節で解説する．

CastOff は，プロファイルやアノテーションによって得られた情報を前提条件としてコード生成を行う．プロファイル情報はプロファイル時に出現した情報しか含まず，アノテーションはユーザの間違いによって誤りを含む可能性がある．そこで，CastOff は**図 7** のように，(1) 生成したコード中に前提条件が保たれているかどうかの検査（以降，ガードと呼ぶ）を挿入する．(2) ガードによる検査に失敗した場合，前提条件が崩れても実行を継続できる，安全なコード（コンパイルの入力となったソースファイルやバイトコード列など）へと実行を切り替える．

また，適用する最適化次第では，ある時点でのメソッド定義に生成したコードが依存する場合がある．たとえば，脱仮想化 [21] を適用したコードは，ある時点でのメソッド定義に依存する．つまり，**図 8** のように，(1) CastOff がコードを生成し，(2) 生成したコードでコンパイル対象を置き換えた時点で，コンパイル対象といくつかのメソッド（脱仮想化の対象となったメソッド）との間に依存関係が生じる．これに対し，CastOff は，置き換えたコードが依存するメソッドを監視する．そして，(4) 依存しているメ

© 2012 Information Processing Society of Japan

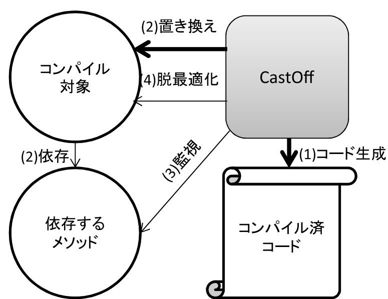


図 8　メソッド定義のフック
Fig. 8　Hook of method definitions.

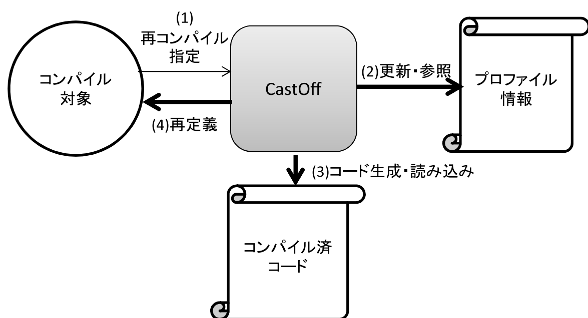


図 9　再コンパイルの設計
Fig. 9　Design of re-compilation.

ソッドが上書きされたことを検出し，脱最適化を適用する．

3.5　再コンパイル

本節では，図 9 を用いて，CastOff の再コンパイル機構の設計を解説する．再コンパイルは，脱最適化が繰り返し発生することを防ぐことを目的に，脱最適化が発生したときに行う．

脱最適化が発生すると，図 9 (1) のように，コンパイル済みコードで置き換えたコンパイル対象が CastOff へ再コンパイルを要求する．このとき，脱最適化の原因が CastOff へわたされる．たとえば，ある変数 `a` の型が `Fixnum` オブジェクトであることを前提としていた場合，a に Bignum オブジェクトがわたると脱最適化が発生する．このとき，変数 `a` に `Bignum` オブジェクトがわたったために脱最適化が発生したという情報が，CastOff へわたされる．

再コンパイルが指示されると，(2) CastOff はわたされた脱最適化の原因を用いて，プロファイル情報を更新する．変数 `a` に `Bignum` オブジェクトがわたったことが脱最適化の原因だった場合，変数 `a` に `Bignum` オブジェクトがわたるという情報を用いてプロファイル情報を更新する．そして，(3)，(4) 更新されたプロファイル情報を用いて実行時コンパイルを行う．再コンパイル後のコードは脱最適化の原因となった条件を組み込んでコンパイルされているため，同じ原因で繰り返し脱最適化が発生することを防ぐことができる．

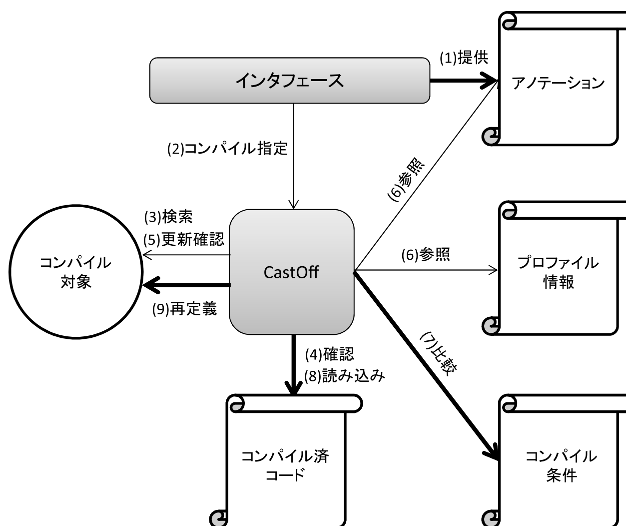


図 10　コンパイル済みコードの再利用に関する設計
Fig. 10　Design of reusing compiled codes.

3.6　コンパイル済みコードの再利用

CastOff はメソッドが定義されたとき（主にプログラムの起動時），コンパイル済みコードが存在するかどうかを確認し，利用可能である場合は読み込む．また，本章のはじめに述べたように，実用性を向上させるためには，生成したコードの速度だけではなく，コンパイルにかかる時間にも配慮する必要がある．そこで，CastOff はコンパイルが指定されたときに，すでにコンパイル済みのコードが存在するかどうかを確認する．そして，利用可能であるならばコンパイルを行わずに，生成済みのコードを利用する．本節では，図 10 を用いて，CastOff のコンパイル済みコード管理を解説する．

まず，(1)，(2) CastOff にアノテーションがわたされ，コンパイルを指示されると，(3) 3.1 節で述べたように，コンパイル対象を検索し，その定義を参照する．次に，(4) コンパイル対象に対し，コンパイル済みのコードがすでに存在するかどうかを確認する．コンパイル済みのコードが存在した場合は，(5) コンパイル対象の定義がコンパイル済みコードを生成したときから変化していないかを確認する．この確認は，コンパイル対象のソースファイルの最終更新日時や，バイトコード列の内容などから行えばよい．そしてさらに，(6) ユーザからわたされたアノテーションとプロファイル情報からコンパイル条件を生成し，(7) コンパイル済みコードを生成したときのコンパイル条件と比較する．コンパイル対象に変化がなく，コンパイル条件が同一であった場合，コンパイルを行ってもすでに生成済みのコードと同じコードを生成するため，(8)，(9) コンパイル処理を行わずに，コンパイル済みのコードをただちに読み込む．

4.　実装

本章では，3 章で述べた CastOff の各機能について，そ

© 2012 Information Processing Society of Japan

の実装を解説する．また，現時点での CastOff には Ruby 処理系と非互換な点があるため，CastOff の非互換性とそれが生じる理由についても述べる．アノテーションのサポートや再コンパイル，コンパイル済みコードの再利用については，実装について解説すべき点はないため，本章での解説の対象とはしない．本稿では以降，Ruby の Foo クラスのインスタンスメソッド bar を `Foo#bar`，特異メソッド [6] baz を `Foo.baz` と記述する．

4.1　実行時コンパイル

3.1 節で述べた実行時コンパイルの設計を，CastOff は**図 11** のように実装している．本節では，図 11 を用いて，CastOff の実行時コンパイルの実装を解説する．

まず，図 11 (1) のように，CastOff にコンパイルが指示されると，CastOff にコンパイル対象メソッドの識別子として，クラスとメソッド名の組がわたされる．CastOff にコンパイルが指示されるタイミングは，**表 2** に示す 4 点である．すると，(2) CastOff はクラスとメソッド名の組を用いてメソッド検索を行い，コンパイル対象の定義を参照する．実行の経過によって，同じクラスとメソッド名に対し，異なる定義が得られる可能性があるが，コンパイル時に参照するのはメソッド検索時に発見した定義となる．つまり，CastOff によるコンパイルの対象となるのは，Ruby 処理系にすでに読み込まれているメソッドである．そして (3) Ruby 処理系の仮想マシンが生成した，コンパイル対象のバイトコード列を入力として，C ソースコードを生成する．ここで生成する C ソースコードは，CRuby の C 拡張のフォーマットに則ったものである．そして，(4)，(5)，(6) CRuby が提供する拡張ライブラリのビルド機構やローダを用いて，C ソースコードから共有ライブラリファイルを生成し，読み込む．このとき，共有ライブラリのローダが，コンパイル対象を CastOff が生成した C 拡張へと置き換える．

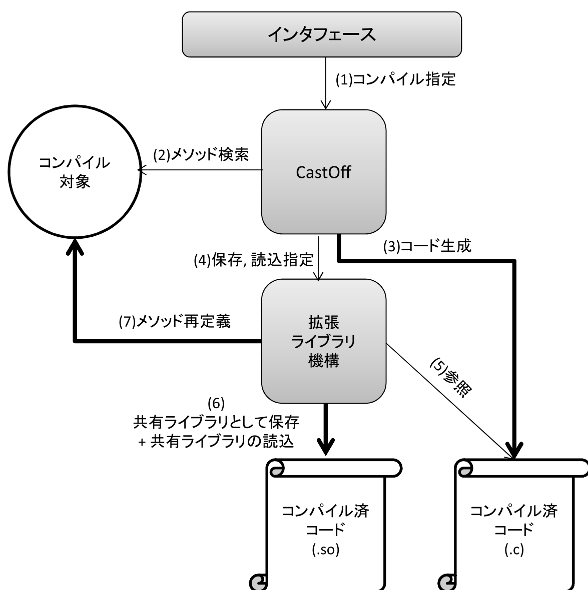


図 11　実行時コンパイルの実装
Fig. 11　Implementation of runtime compilation.

3.1 節の実行時コンパイルの設計を実装するためには，コンパイル対象やコンパイル対象の識別子，そして生成したコードのフォーマットを決める必要がある．CastOff は，コンパイル対象を Ruby のメソッド，コンパイル対象の識別子をクラスとメソッド名の組，生成したコードのフォーマットを CRuby の C 拡張としている．コンパイル対象の検索には CRuby のメソッド検索機構を使用し，コンパイル対象の置き換えには C 拡張のローダを使用する．CastOff の実装でこのような選択をしたのは，次の理由によるものである．

まず，コンパイル対象をメソッドとしたのは，CRuby がメソッドに対し，検索や置き換え，定義のフックなどの機能を提供しているためである．メソッドの置き換えは，`rb_define_method` などの関数を，生成した C 拡張から呼び出すだけで行うことができる．また，メソッド定義のフックは，`Object#singleton_method_added` や，`Module#method_added` を用いることで容易に行うことができる．メソッド検索については，拡張ライブラリに対して関数が公開されていないが，Ruby 処理系のソースコードを参考に，容易にその処理を再現することができる．

コンパイル対象をメソッド全体ではなく，メソッドの一部にすることも考えられる．しかし，CRuby ではダイレクトスレッデッドコードや `switch` 文による命令ディスパッチを行う [7] ため，バイトコード列の一部の処理のみを CastOff が生成したコード（今回は C 拡張のコード）へ置き換えることが難しい．

次に，コンパイル対象の識別子をクラスとメソッド名の組にしたのは，CRuby が提供するメソッドの検索や置き換え，定義のフック機能の入力が，クラスとメソッド名の組であるためである．また，図 2 のようにユーザが CastOff にコンパイルを直接指示する場合に，クラスとメソッド名

表 2　CastOff にコンパイルが指示されるタイミング
Table 2　Compilation direction timings to CastOff.

CastOff が提供するメソッド（`CastOff.compile` など）を通してコンパイルを直接指示されたとき
プロファイル実行を行うとき
プロファイル実行が終わったとき
脱最適化時にともなう再コンパイルが行うとき

© 2012 Information Processing Society of Japan

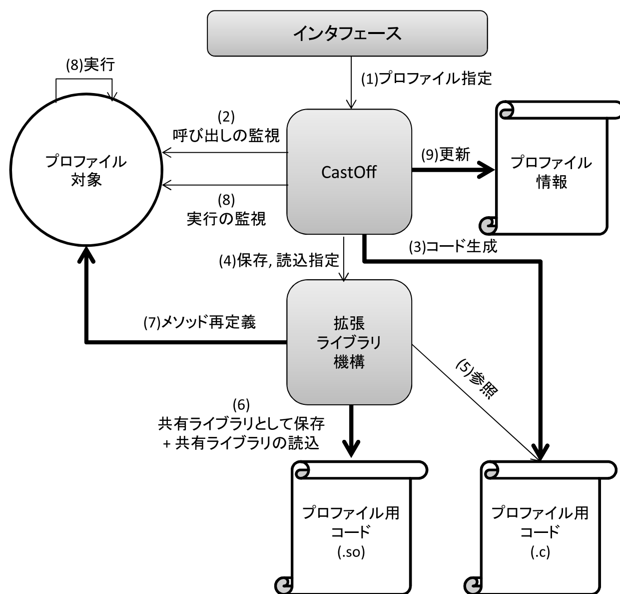


図 12　プロファイル実行の実装
Fig. 12　Implementation of profiling execution.

ならば容易に指定できるというのも，コンパイル対象の識別子をクラスとメソッド名の組とした理由の 1 つとしてあげられる．たとえば識別子にソースファイルと行番号の組を使用することも考えられるが，ソースファイルはカレントディレクトリを考慮して指定するか，フルパスで指定しなければならない．また，ソースファイルを編集するたびに，CastOff にわたす行番号を修正する必要があり，煩雑である．

最後に，生成するコードのフォーマットを C 拡張としたのは，開発の容易さと可搬性，そして性能のためである．C 拡張ではなく直接機械語を生成することも考えられるが，レジスタアロケーション [17] などの最適化をすべて CastOff のために一から実装するのは容易ではない．コンパイル速度の面でも直接機械語を生成したほうが有利であるが，CastOff では開発コストと可搬性の理由から，C 拡張を採用している．コンパイル速度に対する工夫は，プロファイル実行により閾値を超えた回数実行されたメソッドのみをコンパイルすることにとどめている．また，CastOff が生成するコードを最適化を適用したバイトコードにするという手法も考えられるが，バイトコードよりも機械語を生成したほうが命令ディスパッチなどの仮想マシンのオーバヘッドを削減できると考え，今回は採用していない．

4.2　プロファイル実行

3.2 節で述べた実行時コンパイルの設計を，CastOff は**図 12** のように実装している．本節では，図 12 を用いて，CastOff のプロファイル実行の実装を解説する．

まず，図 12 (1) のように，CastOff にプロファイルが指定されると，(2) CastOff は CRuby が提供する API を用いてメソッド呼び出しを監視し，メソッドの呼び出し回数をプロファイルする．そして次に，ユーザから設定された閾値などを用いて頻繁に実行されるメソッドを判断し，そのメソッドの実行を監視する．メソッドの実行の監視は，(2)，(3)，(4)，(5)，(6)，(7) 実行時コンパイルを行い，プロファイル用の命令を含んだコードへと，プロファイル対象のメソッドを置き換えることで行う．そして，(8) 対象プログラムを実行することでプロファイル用の命令を実行し，対象コードのプロファイル情報を取得する．ここで取得するプロファイル情報は，変数，およびメソッドの返り値の型情報である．現在の CastOff では，これら以外の情報は収集していない．最後に，(9) 取得した情報を用いてプロファイル情報を更新する．

プロファイル情報を追記式にする理由や，プロファイルに処理系が提供するイベントフックの API とプロファイル対象コードの書き換えを併用する理由は，3.2 節で述べた．本節では以降，プロファイル対象コードの書き換えを，C 拡張を生成することで行う理由を述べる．

プロファイル対象コードの書き換えを C 拡張を生成することで行うのは，CastOff が生成する C 拡張にプロファイル用の命令を挿入するのが容易であるためである．CastOff が生成する C 拡張には任意のコードを埋め込むことが可能であるため，プロファイルによって参照したい情報（ローカル変数の型情報など）を容易に参照することができる．

バイトコード列書き換えによって，プロファイル命令をバイトコード列中に挿入することも考えられるが，プロファイル用のコードをバイトコード列中に挿入するのは Ruby 処理系では容易ではない．Ruby 処理系には，任意の C の関数を呼び出す `opt_call_c_function` というバイトコードが存在するが，このバイトコードは拡張ライブラリから使用するのは難しい．Ruby 処理系は `gcc` 環境ではダイレクトスレッデッドコード [7] を使用するため，`opt_call_c_function` 命令をバイトコード列書き換えによって挿入するには，`opt_call_c_function` のアドレスを取得する必要がある．また，プロファイル用のメソッドを定義し，メソッド呼び出しのバイトコードによってそれを呼び出すという手法も考えられる．しかし，この方法を用いる場合，ローカル変数などの情報を得るためにはプロファイル用のメソッドで呼び出し元のメソッドフレームを参照する必要がある．これは，実装が煩雑であるだけでなく実行時のオーバヘッドも大きい．

4.3　アノテーションのサポート

3.3 節で述べたように，CastOff はユーザからのアノテーションをサポートする．アノテーションによって指定できるのは，変数やメソッドの返り値のクラス情報，副作用やオブジェクトの破壊，脱出に関する情報である．本節では CastOff がサポートするアノテーションの形式を，それぞれ解説する．

© 2012 Information Processing Society of Japan

```
(1): アノテーションの例
CastOff.compile(
  Foo, :bar, binding, # Foo#bar のコンパイル指定
  :a => [Fixnum, Float], # a のクラスは Fixnum もしくは Float
  :@b => Array, # @b のクラスは Array
  Fixnum => {:- => [Fixnum, Bignum]} # Fixnum#- は Fixnum もしくは Bignum を返す
)

(2): ローカル変数や引数のクラス情報の指定のフォーマット
:ローカル変数名 => クラス or
:ローカル変数名 => [クラス 1, クラス 2, ...] or
[:ローカル変数名 1, :ローカル変数名 2, ...] => クラス or
[:ローカル変数名 1, :ローカル変数名 2, ...] => [クラス 1, クラス 2, ...]

(3): インスタンス変数のクラス情報の指定のフォーマット
:@インスタンス変数名 => クラス or
:@インスタンス変数名 => [クラス 1, クラス 2, ...] or
[:@インスタンス変数名 1, :@インスタンス変数名 2, ...] => クラス or
[:@インスタンス変数名 1, :@インスタンス変数名 2, ...] => [クラス 1, クラス 2, ...]

(4): メソッドの返り値のクラス情報の指定のフォーマット
クラス => {:メソッド名 1 => 返り値のクラス,
           :メソッド名 2 => [返り値のクラス 1, 返り値のクラス 2, ...], ...}

(5): 副作用やオブジェクトの破壊，脱出に関する指定方法
CastOff.set_method_information(オブジェクト, メソッド名, メソッドの形式,
                               :destroy_reciever => true or false,
                               :destroy_arguments => true or false,
                               :escape_reciever => true or false,
                               :escape_arguments => true or false,
                               :side_effect => true or false)
```

図 **13**　アノテーションの形式

Fig. 13　Annotation format.

4.3.1　変数やメソッドの返り値のクラス情報の指定

CastOff がサポートするアノテーションで最も重要なのは，コンパイル対象のメソッドで使用する，変数やメソッドの返り値のクラス情報である．これは，図 **13** (1) や図 2 のように，`CastOff.compile` や `CastOff.compile_singleton_method` の引数として指定する．たとえば，`Foo#bar` というメソッドに対し次のような型情報を指定する場合は，図 13 (1) のように指定する．

- 引数 a のクラスは Fixnum もしくは Float
- インスタンス変数 @b のクラスは Array
- `Fixnum#-` の返り値は Fixnum もしくは Bignum

より具体的には，変数のクラス情報は，図 13 (2)，(3) のように指定し，メソッドの返り値のクラスは，図 13 (4) のように指定する．クラス変数やグローバル変数のクラスの指定方法も，図 13 (2)，(3) と同様である．図 13 (2)，(3)，(4) から分かるように，変数のクラス情報の指定は，変数の名前や名前の配列に対し，とりうるクラスやその配列を対応付けて指定する．そして，メソッドの返り値のクラスは，クラスに対し，メソッド名とその返り値のクラスを保持するハッシュを対応付けて指定する．なお，CastOff では，変数の型情報はメソッドに対して指定する．このため，メソッド内の各ブロックに対し，変数の型情報を別々に指定することはできない．

また，CastOff は，`CastOff.compile` や `CastOff.compile_singleton_method` の引数として，ユーザから Binding オブジェクトを受け取る．そして，受け取った Binding オブジェクトからコンパイル対象のコンテキストを参照することで，定数の型情報を取得する．Binding オブジェクトは，主に `eval` メソッドの第 2 引数として使用するオブジェクトであり，Binding オブジェクトを生成したコンテキストで，`eval` による文字列評価を実行することができる．定数の型情報をローカル変数などの型情報と同様の方式で指定することも考えられたが，Binding オブジェクトによって一括で指定できたほうが使い勝手が良いと考え，本方式を採用した．

© 2012 Information Processing Society of Japan

4.3.2　副作用やオブジェクトの破壊，脱出に関する指定

CastOff は，メソッドの副作用や，メソッドが引数やレシーバのオブジェクトを破壊，もしくは脱出させるかどうかの情報を，アノテーションとして受け付ける．このアノテーションは，図 13 (5) のように `CastOff.set_method_information` を用いて，レシーバの破壊や副作用などの各条件を，それぞれ true か false で指定する．

4.4　脱最適化

本節では，CastOff の脱最適化機構の実装を解説する．脱最適化発生のタイミングが，ガードによる検査に失敗したとき，およびコンパイル済みコードが依存するメソッドが上書きされたときの 2 点であることは，3.4 節で述べた．本節では，この 2 点のタイミングで行う脱最適化について，それぞれ実装を解説する．

4.4.1　ガードによる検査に失敗したときの脱最適化

CastOff は，3.4 節で述べたように，ガードを用いてコンパイルの前提条件が崩れていないかどうかを確認する．CastOff が前提条件として使用するのは，メソッドの定義と型情報の 2 点のみである．この 2 点の前提条件のうち，メソッドの定義に関する前提条件の確認は，コンパイル済みコードに埋め込んだガードではなく，メソッド再定義のイベントハンドラで行う．このため，ガードが確認するのはローカル変数や一時変数などの型情報のみである．

CastOff に実装しているガードでは，図 **14** (1) のように，C の if 文によって候補となる型（図 14 (1) では Fixnum と Bignum）を順に確認する．CastOff は，このようなガードを，確認する変数の分だけ生成する．出現しうる型が膨大な数となった場合，ガードによる検査のコストが大きくなるため，型の候補が一定を超えた時点で型情報を不定とする手法も考えられる．しかし，型情報の個数の閾値をどのように定めればいいのかは明らかではないため，現状の CastOff にはこのような手法は取り入れていない．

ガードによる検査（図 14 (1)）に失敗すると，CastOff は安全に実行できるコードへ実行を切り替えることで脱最適化を行う．ここでの実行の切替え先は，コンパイルの入力となった Ruby メソッドである．コンパイルの入力となった Ruby メソッドは，プログラムにどのような変化があったとしても正しく動くように生成されている．このため，CastOff が生成した関数を実行せずに，コンパイルの入力となったメソッドへと実行を切り替えれば，脱最適化を行うことができる．コンパイルの入力となった Ruby メソッドは，Ruby 処理系に解放されないよう，コンパイル対象メソッドの置き換え時に退避させておく．安全に実行できるコードを CastOff が生成するという手法も考えられるが，ここではコンパイル時間を削減するために，コンパイルの入力となった Ruby メソッドを利用している．

```
      …
(1) if (guard(local0_a, Fixnum)) {
      /* ガードによる検査に成功 */
    } else if (guard(local0_a, Bignum)) {
      /* ガードによる検査に成功 */
    } else {
      /* ガードによる検査に失敗 */
      recompile(sign, local0_a, "a");
(2)   pc = 2;
      goto deoptimize;
    }

    /* 確認する変数の分だけ，ガードが存在 */
    if (guard(local1_b, String)) {
      /* ガードによる検査に成功 */
    } else if (guard(local1_b, Symbol)) {
      /* ガードによる検査に成功 */
    } else {
      /* ガードによる検査に失敗 */
      recompile(sign, local1_b, "b");
      pc = 2;
      goto deoptimize;
    }
    …
  deoptimize:
(3) context[0] = local0_a;
    context[1] = local1_b;
   …
(4) return original_code(context, pc);
```

図 14　ガードによる検査と脱最適化
Fig. 14　Check by guard and deoptimization.

ガードによる検査に失敗したときの脱最適化は，図 14 のように行う．まず，(2) ガードに失敗した箇所がどこであるかの目印として，プログラムカウンタを設定し，(3) ローカル変数や一時変数などの実行コンテキストを収集する．ここで設定するプログラムカウンタは，コンパイルの入力となった Ruby メソッドの命令列のうち，ガードに失敗した命令の位置を指す．そして，(4) CastOff の脱最適化器にプログラムカウンタと実行コンテキストをわたす．CastOff の脱最適化器は，わたされた実行コンテキストを用いて，Ruby のメソッドの実行に必要なメソッドフレームを構築し，Ruby 処理系の評価器を呼び出す．これにより，入力となった Ruby メソッドを，わたされたプログラムカウンタの位置から実行することができる．

脱最適化時に収集する実行コンテキストは，脱最適化に備えてあらかじめ計算し，保持しておく必要がある．これは，本脱最適化手法における，最適化への制限である．

4.4.2　メソッドが上書きされたときの脱最適化

CastOff は脱仮想化を行うことで，C で定義されたメソッドの呼び出しを，単純な C の関数呼び出しへ置き換える．このため，3.4 節で図 8 を用いて解説したように，コンパ

© 2012 Information Processing Society of Japan

```
    …
   local_0 = fptr_Foo_bar(args);
    …
```

図 15　Foo#bar メソッドの呼び出し
Fig. 15　Method invocation of Foo#bar.

```
    # Foo#bar が上書きされたとき,
    # この処理を呼び出し,
    # 関数ポインタを再設定する
    …
(1) if (c_method(Foo, bar)) {
(2)   fptr_Foo_bar = get_fptr(Foo, bar);
    } else {
(3)   fptr_Foo_bar = method_dispatcher;
    }
    …
```

図 16　関数ポインタの設定
Fig. 16　Function pointer setter.

イル済みコードが依存するメソッドの再定義を監視する．そして，依存しているメソッドが上書きされたことを検出し，脱最適化を適用する．

CastOff においてメソッドが上書きされたときに無効化するべきなのは，脱仮想化を適用し，メソッド呼び出しを C の関数呼び出しに置き換えた箇所のみである．本項では以降，C の関数呼び出しに置き換えた箇所をどのように脱最適化するかについて解説する．

CastOff は脱仮想化を適用したメソッド呼び出しを，**図 15** のように，関数ポインタを用いた C の関数呼び出しへと置き換える．図 15 では，メソッド呼び出しで呼び出す先が `Foo#bar` であると判断し，脱仮想化を適用している．図 15 で `Foo#bar` の呼び出しに用いている関数ポインタ `fptr_Foo_bar` は，**図 16** のように初期化する．図 16 では，まず，(1) `Foo#bar` が C で定義されたメソッドかどうかを確認する．そして，(2) `Foo#bar` が C で定義されたメソッドならば，`Foo#bar` の実体である C の関数ポインタを `fptr_Foo_bar` へと設定する．(3) `Foo#bar` が C で定義されたメソッドではなかった場合，Ruby 処理系が提供するメソッドディスパッチャの関数を `fptr_Foo_bar` へと設定する．

このように，CastOff は C で定義されたメソッドの呼び出しを C の関数呼び出しへと置き換え，メソッド呼び出しを高速化する．しかし，この手法では，C で定義されたメソッドが実行の途中で Ruby で定義されたメソッドへと置き換わった場合に問題が生じる．上記の例の場合，`Foo#bar` が C で定義されていた場合，`fptr_Foo_bar` には C で定義された `Foo#bar` の実体が入る．ここで，実行の途中で `Foo#bar` が Ruby で定義されたメソッドへと置き換わった場合，C で定義された `Foo#bar` ではなく，Ruby で定義された `Foo#bar` を呼び出す必要がある．しかし，`fptr_Foo_bar` は C で定義された `Foo#bar` の実体を指しているため，`Foo#bar` が置き換えられた後も C で定義された `Foo#bar` が呼び出されてしまう．

そこで，CastOff では，メソッドの上書きをフックしたときに，このような問題が発生する箇所を認識する．そして，問題が発生する箇所それぞれに対し，図 16 に示す関数ポインタの初期化処理を，再度呼び出す．これにより，問題が発生する関数ポインタがメソッド上書きのタイミングですべて更新され，置き換えた後のメソッドを呼び出せるようになる．

Ruby には，C で定義されたメソッドの呼び出し方法が複数あるため，メソッド上書きによってメソッドの呼び出し方法が変わってしまった場合，C の関数の引数の順序などを変更する必要がある．これに対し，CastOff の実際の実装では，呼び出し方法の違いを吸収するための関数を生成し，それを呼び出すことで引数の順序などの違いを吸収している．この処理は実装における非常に細かい点であるため，本稿ではこれ以上の説明は行わない．

4.5　コードの再利用

本節では，CastOff のコンパイル済みコードの再利用の実装を解説する．基本的な流れは 3.6 節で述べたとおりである．本節では特に，コンパイル対象の定義がコンパイル済みコードを生成したときから変化していないことの確認を，CastOff がどのように行うかを解説する．

CastOff は，コンパイル対象の定義がコンパイル済みコードを生成したときから変化していないことを確認するために，ファイルの最終更新日時を用いる．図 11 や図 12 にあるように，CastOff のコンパイル済みコードは共有ライブラリとして保存される．CastOff は，コンパイル済みコードの最終更新日時と，コンパイルの入力となるメソッドが記述されているソースファイルの最終更新日時を比較する．このとき，コンパイル済みコードの最終更新日時の方が新しかった場合は，コンパイルを行ってからソースファイルが変更されていないということが確認できる．

4.6　コード生成

CastOff は，Ruby 処理系の仮想マシンのバイトコード列から中間表現を生成し，コントロールフローグラフ（以降 CFG と呼ぶ）を構築する．そして，構築した CFG を用いて手続き内のデータフロー解析を行い，C ソースコードを生成する．この際に適用する最適化手法を以下に列挙する．本節では，以下に列挙する最適化それぞれについて，最適化の概要と適用条件を簡単に解説する．

- 脱仮想化
- 無用コード除去 [17]
- 冗長な文字列リテラル複製の削減
- ブロックインライニング

© 2012 Information Processing Society of Japan

- C と Ruby 間の型変換の削減

4.6.1　脱仮想化

脱仮想化とは，メソッドの呼び出し先を確認する最適化手法である．CastOff は脱仮想化を用いることで，各メソッド呼び出しにおいて，どのメソッドが呼び出されるかを確認する．そして，4.4.2 項でも述べたように，呼び出す先のメソッドが C で定義されていた場合，メソッドの呼び出しを C の関数ポインタを用いた呼び出しへ置き換える．脱仮想化を使用し，メソッド呼び出しを C の関数ポインタを用いた呼び出しへ置き換えることで，メソッドディスパッチのオーバヘッドを削減することができる．

CastOff は脱仮想化を適用するために，プロファイルやユーザからのアノテーションによって得た型情報を，CFG 上に伝播させる．そして，プログラムの各点において，各変数がどのような型を取る可能性があるかを確認する．型を確認することができれば，メソッド名と組み合わせることで，呼び出す可能性のあるメソッドを確認することができる．

メソッドが上書きされた場合の挙動などは，4.4.2 項で述べたとおりである．また，あるメソッド呼び出し箇所において，レシーバが複数の型をとりうる場合，メソッド呼び出し箇所の直前に型を確認するコードを挿入し，C の if 文によって使用する関数ポインタを選択している．

4.6.2　無用コード除去

無用コード除去とは，使用されることのない値を生成するコードを除去する最適化である．CastOff が生成するコードでは，Ruby のインスタンス変数などの参照に，Ruby 処理系が提供する C の関数を使用する．これらの関数定義は，CastOff が生成するコードの外部に存在するため，呼び出しの除去を C コンパイラに期待することができない．このため，CastOff は，コード生成を行う際に，無用コード除去を適用する．

無用コード除去を適用する際には，脱最適化で使用するローカル変数や一時変数などの実行コンテキストが変化しないよう，注意する必要がある．CastOff では，脱最適化で使用する実行コンテキストが最適化によって変化しないよう，ガードが実行コンテキストの各値を参照する．これにより，無用コード除去により実行コンテキストが変化することを防いでいる．無用コード除去の手法そのものは既知であるため，本項ではこれ以上の解説は行わない．

4.6.3　冗長な文字列リテラル複製の削減

CastOff は，文字列リテラルの複製処理のうち，冗長と判断できるものを削除する．Ruby では文字列を破壊的に変更することができるため，処理系は，文字列リテラルが破壊されないように，文字列リテラルを含む式を評価するたびに，評価する文字列リテラルの複製を行う．リテラルから複製した文字列が破壊されないような場合，このような複製処理は冗長である．リテラルの複製処理の除去は，オブジェクトの割当て，解放コストの削減だけでなく，GC の回数の削減にもつながるため，Ruby プログラムの高速化のために効果的である．

リテラルの複製を削減するためには，生成する各リテラルについて，破壊的変更を行う可能性のあるメソッドで使用される可能性があるかどうかを確認すればよい．破壊的変更を行う可能性のあるメソッドで使用されないことが確認できれば，リテラルを複製せずに，使いまわすことができる．これを確認するために，CastOff は CFG をたどり，各リテラルに対して以下の点が満たされるかどうかを確認する．

(1) 代入文やメソッド呼び出しによって，インスタンス変数や配列オブジェクトなどに格納され，大域的に参照されるような状態にならない．
(2) `String#force_encoding`（文字列オブジェクトのエンコーディングを破壊的に変更する）などのメソッドによって，破壊的な変更を加えられない．

上記の点を確認するために，まず，CFG 上の代入文をたどり，インスタンス変数などに格納されないことを確認する．そして，リテラルの値がメソッドのレシーバや引数として使用されるとき，そのメソッドが上の条件をともに満たすかどうかを，ユーザから得たアノテーション（4.3 節）を使用して確認する．リテラルの値が使用される CFG 上のすべての点において上記の条件が確認できた場合にのみ，対象リテラルの複製処理を除去する．

4.6.4　ブロックインライニング

Ruby のイテレータ（`Fixnum#times` のように，ブロックを起動するメソッド）には，Ruby で定義されたものと C で定義されたものの 2 種類がある．C から Ruby の処理を呼び出すには Ruby 処理系の評価器を呼びなおす必要があるため，C のイテレータから Ruby で記述されたブロックの呼び出しは，Ruby のイテレータから Ruby で記述されたブロックの呼び出しよりもオーバヘッドが大きい．

そこで，CastOff は C のイテレータを高速化するために，呼び出し元のメソッド，C のイテレータ，およびイテレータが起動するブロックを，単一の C の関数へ変換する（ブロックインライニング）．このブロックインライニングにより，ブロック呼び出しにともなうオーバヘッドを除去する．CastOff がブロックインライニングを行うのは，イテレータが C によって実装されていた場合のみである．イテレータが Ruby によって実装されていた場合には，ブロックインライニングは行っていない．

CastOff の行うブロックインライニングでは，C のイテレータも含めてインライニングを行う．このため，ブロックインライニングを行うためには，Ruby 処理系の C のイテレータそれぞれに対し，インライニング用のコードを手作業で実装する必要がある．現在，CastOff では，特に利用することが多く，かつ手作業での実装が容易であるメソッド

© 2012 Information Processing Society of Japan

（`Fixnum#times`，`Array#each`，`Array#map`，`Array#map!`）のみを，ブロックインライニングの対象としている．

4.6.5　C と Ruby 間の型変換の削減

Ruby 処理系では，Fixnum オブジェクトや Float オブジェクトなどの計算時に，Ruby が内部的に扱うデータ構造から C のリテラルへの変換や，その逆の変換を行う（以降，本稿では型変換と呼ぶ）．CastOff は，このような型変換のうち，冗長と判断することができるものを削減する．

たとえば `1.0 + 2.0 + 4.0` という式を評価した場合，加算を行うために内部的に Ruby の Float オブジェクトから C の double 型への変換（Unboxing と呼ぶ）が行われる．そして，計算結果として得られる C の double 型の値を Ruby オブジェクトとして扱うために，Float オブジェクトへの変換（Boxing と呼ぶ）が行われる．つまり，Float オブジェクトに対する加算のたびに，Unboxing が 2 度行われ，Boxing が 1 度行われる．`1.0 + 2.0 + 4.0` では 2 度の加算にともない，4 度の Unboxing と 2 度の Boxing が行われる．Float オブジェクトに対する Boxing では，Float オブジェクトの割当て処理をともなうため，オーバヘッドが大きい．これらの型変換処理を削減することができれば，オブジェクト割当ての回数を減らし，GC の実行回数も削減することができる．

`1.0 + 2.0 + 4.0` という式では，式の評価後にも使用されるのは，最後の `3.0 + 4.0` によって生成される 7.0 のみである．最初の `1.0 + 2.0` によって生成される 3.0 という Float オブジェクトは，`1.0 + 2.0 + 4.0` の内部でのみ使用され，他からは使用されない．このため，`1.0 + 2.0` で得られる 3.0 を Boxing せずに C の double 型として保持し，そのまま 4.0 との加算で用いれば，3.0 に対する Boxing と Unboxing を除去することができる．

表 3　型変換の削減の対象となるメソッド
Table 3　Target methods of type conversion elimination.

クラス	メソッド名
Fixnum	-@, to_f, +, -, *, >, >=, <, <=, ==, !=, ===
Float	-@, to_f, to_i, +, -, *, /, >, >=, <, <=, ==, !=, ===

CastOff は**表 3** に示すメソッドに対し，引数が C の long 型や double 型，Ruby の Fixnum オブジェクトや Float オブジェクトがわたった場合の処理をそれぞれ実装している．そして，表 3 に示すメソッドの計算で生じる C の long 型や double 型に対し，使用される箇所が表 3 に示すメソッドに限られていた場合は，Boxing を行わない．これによって，Fixnum オブジェクトや Float オブジェクトに対する型変換の回数を削減している．

4.7　非互換性

現時点での CastOff には Ruby 処理系と非互換な点がある．本節では，CastOff の非互換性とそれが生じる理由について述べる．

4.7.1　定数上書き

CastOff は，コンパイルしたコードの読み込み時，もしくは最初に実行したときに定数解決を行い，1 度定数解決を行った後はすべて同じ値を使用する．このため，定数の再定義が行われても，再定義前の値を使用し続ける．これは，性能上の理由によるものである．

Ruby は拡張ライブラリに対し，定数解決のために `rb_const_get` という関数を提供している．`rb_const_get` を使用すれば最新の定数値を参照することができるが，`Foo::Bar::Baz` という定数を解決するために，Foo，Bar，Baz それぞれに対して `rb_const_get` を呼び出す必要がある．Ruby において定数は頻繁に使用されるため，定数解決のたびに `rb_const_get` を用いて定数解決を行うことは，性能面で好ましくない．また，Ruby の定数は上書き可能であるものの，文字どおり値を固定して使われることが多い．CastOff では，Ruby の定数を書き換えるようなプログラムは稀であると考え，定数の再定義に対する互換性よりも速度を優先し，最初に取得した定数値を使い続ける．

4.7.2　メソッド上書き

CastOff は，実装規模が小さく，かつ上書きされることも稀であると考えられるメソッド（**表 4**）の上書き時は，脱最適化ではなく例外を発生させる．これも，定数上書きの非互換性と同様に，性能上の理由によるものである．

表 4　inline 宣言の対象となるメソッド
Table 4　Target methods of inline declaration.

クラス，モジュール	メソッド名
String	<<, +, ==, ===, !=, empty?
Array	[], []=, length, size, empty?, last, first, each, map, map!
Fixnum	&, +, -, *, <=, <, >=, >, ==, !=, ===, -@, to_f, times
Float	+, -, *, /, <=, <, >=, >, ==, !=, ===, -@, to_f, to_i
Class	new
Kernel	===, nil?
NilClass	nil?
Object	===
BasicObject	!

© 2012 Information Processing Society of Japan

CastOff は，4.4.2 項で述べたように，脱仮想化によってメソッド呼び出しを C の関数ポインタによる呼び出しへと置き換える．そして，脱仮想化の対象となるメソッドが上書きされた場合，関数ポインタを更新することで脱最適化を適用する．しかし，Fixnum オブジェクトに対する加減乗除などのメソッドのように実装規模が小さいメソッドは，C レベルでインライニングしたときに，C コンパイラによる最適化を期待することができる．CastOff は，表 4 のように実装規模が小さく，上書きされることも稀であると考えられるメソッドに対しては，速度のために，関数ポインタによる間接呼び出しではなく inline 宣言した C の関数の直接呼び出しを使用する．表 4 に示すメソッドのうち，特に `Fixnum#times`，`Array#each`，`Array#map`，`Array#map!` の 4 つのメソッドはブロックを受け取るため，ブロックインライニング（4.6 節）の対象としている．4.4.2 項で述べた脱最適化では関数ポインタの更新のみを行うため，C の関数の直接呼び出しにしてしまったメソッド呼び出しでは，再定義後も再定義前のメソッドが呼び出されてしまう．このため，現在の CastOff では，これらのメソッドが上書きされたとき例外を発生させる．Fixnum オブジェクトに対する加減乗除などのメソッドを上書きすることは稀であると考えられるため，CastOff ではこのようなメソッドの再定義に対する互換性よりも速度を優先している．

4.7.3 組み込みメソッド

CastOff は，Ruby で定義されたメソッドを C 拡張で定義したメソッドへと変換する．このため，Ruby で定義した場合と C 拡張で定義した場合で挙動が異なるメソッドは非互換となる．このようなメソッドとしては，`Method#arity` や `Proc#arity` などの Ruby 処理系の組み込みメソッドがあげられる．`Method#arity` と `Proc#arity` は，それぞれメソッドと Proc の引数に関する情報を取得するメソッドである．Ruby 処理系はメソッドや Proc を取り扱うとき，対象となるメソッドや Proc が，Ruby で定義したものか C 拡張で定義したものかによって処理を分けている．このため，メソッドや Proc に関する情報（引数やローカル変数の名前など）を扱うメソッドでは，Ruby で定義していた場合と C 拡張で定義していた場合でそれぞれ異なる結果が返される．

また，CastOff を使用した場合，Ruby の継続（Continuation）を扱うメソッドとの非互換性が生じる．この非互換性の理由も，継続を扱うときに呼び出す実行コンテキストの保存処理で，Ruby で定義したメソッドと C で定義したメソッドを分けて扱っているためである．

5. 議論

4 章で述べたように，CastOff の実行時コンパイル，プロファイル実行，アノテーションのサポート，脱最適化，再コンパイル，コンパイル済みコードの再利用という機能は，Ruby 処理系の C 拡張，メソッド再定義，メソッド検索，メソッド呼び出しのフックや Binding オブジェクトという機能を用いて実現している．このように，Ruby 処理系が提供する機能によって，CastOff の機能を実現することができている．

しかし，いくつかの Ruby 処理系の内部実装や機能面での不足が，4.7 節で述べた非互換性などの，CastOff の実装上の問題につながっている．CastOff の実装で問題となった点と，原因となった Ruby 処理系の機能を**表 5** にまとめる．本章では，4 章で解説した CastOff の実装，および表 5 をふまえて，CastOff のようにライブラリとしてコンパイラを実装するために，処理系がどのような機能を提供するべきかを議論する．

5.1 動的機能の監視

最適化には，ある処理で扱う対象が固定されていることを前提に適用するものが存在する．たとえば，脱仮想化などの最適化は，呼び出されるメソッドが固定されていることを前提条件とした手法である．Ruby のような動的言語では，これらの最適化を単純に適用することが難しい．なぜなら，動的言語が提供するメソッド上書きなどの機能を使用することで，最適化の対象（脱仮想化におけるメソッドの呼び出し先など）を，容易に実行時に書き換えることができてしまうためである．動的言語を対象としたコンパイラで新たに最適化を実装するときは，実装する最適化の前提条件が，動的な機能によって崩されないかどうかをつねに意識する必要がある．

処理系が動的な機能に対するフックを提供していた場合，

表 5 CastOff の実装で問題となった Ruby 処理系の機能
Table 5 Functions of Ruby runtime which cause problem on CastOff development.

Ruby 処理系の機能	CastOff への影響
定数上書きをフックする機能の不足	定数上書きに関する非互換性. CastOff では，定数を上書きしても無視される.
C のメソッドと Ruby のメソッドの扱いの違い	`Method#arity` などのメソッドに対する非互換性. Ruby の継続（Continuation）に対する非互換性.
処理系の内部機能の非公開	特定バージョンの Ruby 処理系への強い依存. 非公開機能を補完するための実装コスト.

© 2012 Information Processing Society of Japan

最適化の前提条件が崩れたかどうかの確認は，動的な機能が使用されたタイミングでのみ行えばよいため，性能にほぼ影響を与えることなく行うことができる．CastOff の場合，脱仮想化の前提条件の確認のために，メソッドの再定義という動的な機能に対し，`Object#singleton_method_added` や `Module#method_added` という Ruby 処理系組み込みのメソッドを使用し，メソッド再定義をフックする．そして，4.4 節で述べたように，Ruby のメソッド再定義を監視し，必要に応じて脱最適化を適用する．

動的な機能に対するフックが直接与えられていなくても，動的な機能の処理の実体がメソッドとして定義されていれば，そのメソッドをコンパイラ側で上書きすることで，動的な機能が使用されたかどうかを監視することができる．たとえば CastOff では，Ruby の Mix-in という，クラスに対して動的にメソッドを加える機能を監視するために，その機能の実体となる `Module#include` や `Module#extend` というメソッドを上書きしている．

動的な機能が使用されていても大丈夫なように保守的なコードを生成するという手法も考えられる．しかし，動的な機能という，使用されることが稀である機能のために，頻繁に実行する機能の性能が大きく損なわれてしまう可能性がある．Ruby 処理系は Ruby の定数操作に対するフックを提供しておらず，定数操作の実体はメソッドではないため，定数の再定義という動的な機能はフックすることはできない．定数の再定義に対処するためには，定数の参照処理を毎回実行する必要がある．4.7 節で述べたように，定数の参照処理を毎回実行することは性能上好ましくない．また，Ruby の定数は上書き可能であるものの，文字どおり値を固定して使われることが多く，再定義されることは稀である．このため，CastOff では，定数の再定義に対する互換性よりも，速度を重視し，1 度参照した値を使用し続けるという方針を採用している．

以上より，CastOff のように処理系に手を加えることなく効果的なコンパイラを開発するためには，処理系が動的な機能に対するフックを提供するべきである．Ruby のメソッド上書きに対する `Object#singleton_method_added` や `Module#method_added` のように，フックするためのメソッドが提供されていた場合や，Ruby の Mix-in に対する `Object#include` や `Module#include` のように，動的な機能の実体が上書き可能なメソッドであった場合は，性能に与える影響を抑えながら脱最適化を適用することができる．

最後に，動的な機能に対するフックでは，コールバックを複数設定できることが望ましい．`Object#singleton_method_added` や `Module#method_added` のように，メソッド上書きのフックをメソッドとして登録する場合，CastOff が登録した `Object#singleton_method_added` などがコンパイル対象のプログラムによって上書きされてしまう可能性がある．このため，現在の CastOff では，`Object#singleton_method_added` や `Module#method_added` の上書きを制限している．

5.2　メソッドの種類による場合分け

4.7.3 項で述べたように，CastOff に生じる非互換性の原因の 1 つは，Ruby 処理系が内部的に Ruby に関するデータと C に関するデータを大きく区別して扱っているという点である．メソッドや Proc に関するデータが，Ruby で定義したものと C で定義したものによって処理を分けられているため，`Methor#arity` や `Proc#arity` に非互換性が生じている．

このような非互換性が生じるのは Method オブジェクトや Proc オブジェクトだけではない．Ruby 処理系がメソッド呼び出し時に構築するメソッドフレームに対する処理も，Ruby と C によって処理が分けられている．たとえば，文字列評価を行う `eval` メソッドでは，呼び出されたときに Ruby のメソッドや Proc が対応付けられたメソッドフレームを探し，見つけたメソッドフレームに保持された実行時コンテキスト（ローカル変数など）を参照する．このため，CastOff がコンパイルしたメソッドから `eval` メソッドを呼び出した場合，本来使用されるべきメソッドフレームが無視されてしまう．

CastOff が生成したコードを C のメソッドとして登録するのは，Ruby の処理として登録するよりも C のメソッドとして登録したほうがメソッド呼び出しなどの処理が高速となるためである．CastOff が生成したメソッドや Proc を Ruby で記述した処理として登録するという手法も考えられる．しかし，これを行うためには CastOff が生成した C の関数をバイトコードから呼び出す必要があり，メソッドの呼び出し速度などが低下する．つまり，互換性の面では Ruby で記述された処理として登録する必要があり，速度の面では C の処理として登録する必要がある．

以上の点をふまえて，CastOff のようなコンパイラを開発するうえでは，処理系に対し，メソッドなどの挙動を特定の処理ごとに細かく設定できることが望ましい．Ruby の処理，C の処理という大きな粒度で処理を分けられてしまうと，本節や 4.7.3 項で述べているように，速度と互換性を両立することができなくなってしまう．このため，たとえばオプショナル引数やデフォルト引数を持つかどうかや，メソッドフレーム上に実行時コンテキストを持つかどうかなどの設定を，フラグなどで細かく指定できることが望ましい．

5.3　処理系の内部機能の公開

CastOff の実装において最も煩雑かつ困難であった点の 1 つが，処理系がライブラリに対して非公開としているグローバル変数や関数の存在である．たとえば，4.1 節で述べたように，CastOff は実行時コンパイルの実装にメソッ

© 2012 Information Processing Society of Japan

ド検索の機能を使用する．しかし，メソッド検索の機能はRuby 処理系内部に関数として実装されているものの，拡張ライブラリに対して関数が公開されていない．このため，CastOff ではRuby 処理系のソースコードを参考に，メソッド検索の機能を再現している．

また，4.4 節で述べたように，CastOff の脱最適化器は，わたされた実行コンテキストを用いてRuby のメソッドの実行に必要なメソッドフレームを構築し，Ruby 処理系の評価器を呼び出す．しかし，Ruby 処理系の評価器はRuby 処理系内部に関数として実装されているものの，メソッド検索の機能と同様に拡張ライブラリに対して公開されていない．また，Ruby 処理系の評価器そのものをCastOff で再現することは非常に煩雑である．このため，CastOff では，Ruby 処理系の実装を参考にメソッドフレームを書き換え，Ruby 処理系の評価器を間接的に呼び出している．このメソッドフレーム書き換え処理は，Ruby 処理系の内部実装に強く依存する処理である．

ここで留意するべきは，CastOff が必要とした機能は，処理系内部にすでに存在していたという点である．CastOff では，処理系が非公開としている機能を自身で補うために，実装が複雑化している．処理系が非公開にしていた関数を公開するだけで，CastOff のようなコンパイラの開発を容易にすることができる．

Ruby 処理系がCastOff の実装で必要になった機能を公開していないのは，これらの機能の仕様が先のバージョンで変更される可能性があるためである．CastOff が必要とする機能は，メソッド検索やメソッドの同一性の判定，仮想マシンが管理する世代番号や，Ruby 処理系の評価器など，Ruby 処理系の内部的な機能が中心である．これらの機能の仕様は，Ruby 処理系の今後の発展にともない変更される可能性がある．このため，Ruby 処理系公式のAPI としてライブラリに提供することが難しい．

しかし，これらの機能が公開されていれば，CastOff の実装にかかる労力を削減することができたのは事実である．これらの機能は，処理系の特定バージョンに依存してもよいので，何らかの形（C の関数やグローバル変数のシンボルを入力とし，指定した関数やグローバル変数のアドレスを返すなど）で提供するべきである．特定バージョンに依存する形で機能が提供されたとしても，CastOff 側で機能を再現するより実装コストは低い．なぜなら，CastOff 側で必要となった機能を再現した場合，結局は処理系の特定バージョンに依存してしまうためである．たとえば，4.1 節で述べたように，メソッド検索の機能はRuby 処理系から提供されていないため，Ruby1.9.3 処理系のソースコードを参考に，CastOff 側で再現している．このため，CastOff が使用するメソッド検索の機能は，Ruby1.9.3 処理系に依存している．

6. 評価

本章では，**表 6** に示すベンチマークを用いて，CastOff による性能向上，およびプロファイル実行とコンパイルの速度を評価する．本評価では，マシンにCPU IntelCore2Quad 2.66 GHz，メモリ4 GB を，OS にGNU/Linux 2.6.31 32-bit を，コンパイラにGCC4.4.1 最適化オプション-O3 を使用し，Ruby 処理系にはruby 1.9.3p0（2011-10-30 revision 33569）を使用した．性能向上の評価では，CastOff を用いた実行と，CastOff を用いない，Ruby 処理系による通常の実行でそれぞれ3 度実行し，それぞれの最小の実行時間を比較する．

なお，性能向上の評価におけるCastOff の実行では，次のような前処理を行う．まず，あらかじめベンチマーク実行と同一の条件でベンチマークをプロファイル実行し，コンパイルを行う．そして，脱最適化と再コンパイルが生じなくなるまで繰り返しベンチマークを実行する．このため，CastOff のベンチマーク実行では，脱最適化やコンパイルにかかる時間は含まない．また，プロファイル実行とコンパイルの速度の評価では，上記の前処理にかかる時間を評価に用いる．

本章ではまず，CastOff によって性能に影響を受ける箇所について簡単に解説する．そして，その解説をふまえて，ベンチマークの実行結果を分析する．

6.1 CastOff によって性能に影響を受ける処理

CastOff によって速度に影響を受ける処理は，主に3 つある．1 つ目がRuby 処理系の仮想マシンの処理，2 つ目がFloat オブジェクト（Ruby における浮動小数点数を扱

表 6　ベンチマーク一覧
Table 6　List of benchmarks.

ベンチマーク	内容	備考
Mandelbrot	マンデルブロ集合	浮動小数点演算が中心
Tarai	たらいまわし関数	整数演算，メソッド呼び出しが中心
Sieve	エラトステネスのふるい	配列処理，整数演算が中心
Renderer	レイトレーシング	浮動小数点演算，インスタンス変数操作が中心
Nbody	N 対問題	浮動小数点演算，インスタンス変数操作が中心
Empty	空のスクリプト	起動時間の計測に使用
RDoc	ドキュメント生成	1 万行規模

© 2012 Information Processing Society of Japan

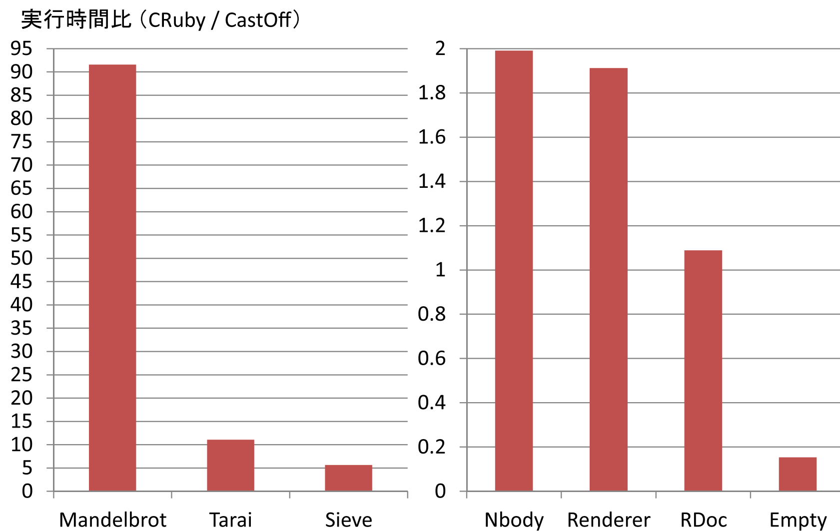


図 17 ベンチマークの実行結果
Fig. 17 Benchmark result.

表 7 ベンチマーク実行の絶対時間（秒）
Table 7 Absolute time of benchmark execution (sec).

ベンチマーク	Ruby	CastOff	C	プロファイル実行＋コンパイル
Mandelbrot	49.445	0.54	0.343	343.973
Tarai	77.139	6.955	1.482	1,817.267
Sieve	94.619	16.78	1.113	1,805.288
Renderer	10.682	5.585	未計測	110.145
Nbody	7.378	3.706	未計測	97.49
Empty	0.022	0.143	未計測	0.142
RDoc	128.515	117.996	未計測	1,225.114

うオブジェクト）や文字列リテラルに対するオブジェクト割当て，3 つ目がプログラムの起動時間である．

まず，CastOff は，Ruby で記述されたメソッドを C に変換する．このとき，Ruby のローカル変数操作を C のローカル変数操作に変換，メソッド呼び出しを C の関数呼び出しに変換するなど，様々な工夫を加えている．これにより，Ruby 処理系の仮想マシンにおける，バイトコードディスパッチやメソッドディスパッチ，ローカル変数などの実行時コンテキストの参照処理を高速化している．

次に，CastOff は Ruby の文字列リテラルや，Float オブジェクトに対し，冗長と判断できる割当てを除去する．Ruby は文字列リテラルを評価するたびに文字列リテラルを複製し，String オブジェクト（Ruby における文字列を扱うオブジェクト）を生成する．これに対し，CastOff は文字列リテラルが破壊されないと判断できた場合，文字列リテラルから String オブジェクトを生成せずに，同一の String オブジェクトを使用する．また，Ruby は Float オブジェクトの加減乗除を行うたびに Float オブジェクトを生成する．これに対し，CastOff はインスタンス変数やメソッドの返り値などにならない Float オブジェクトを C の double 型として保持することで，Float オブジェクトの割当てを削減する．これにより，特に Float オブジェクトが中心となるプログラムは CastOff を用いることで特に高速化することが期待できる．

最後に，CastOff は主に Ruby を用いて実装しており，コマンドラインツール `cast_off` に与えられた引数の処理や，Ruby で記述された `cast_off` のコンパイラ本体の読み込み時間なども，単純なスクリプトでは無視できないオーバヘッドとなる．このため，非常に実行時間の短いようなプログラムを対象とした場合，CastOff の初期化時間のほうが対象プログラムの実行時間よりも長くなってしまい，速度が低下する可能性がある．

6.2 ベンチマークの実行結果

本節では，**図 17** や**表 7** に示すベンチマークの実行結果を用いて，CastOff による性能向上，およびプロファイル実行とコンパイルの速度を評価する．まず，性能向上について評価し，次にプロファイル実行とコンパイルの速度を評価する．

6.2.1 性能向上の評価

ベンチマークの実行結果を図 17 に示す．図 17 より，CastOff によって Empty ベンチマークを除き，今回用いた

© 2012 Information Processing Society of Japan

マイクロベンチマークを高速化できていることが分かる．

まず，図 17 では，浮動小数点演算が中心のベンチマークである Mandelbrot が最も高速化できており，91.56 倍高速化している．これは，6.1 節で述べたように，CastOff が Float オブジェクトの処理を C の double 型の処理へと変換しているためである．Mandelbrot ベンチマークは，一時的な Float オブジェクトを数多く生成する．CastOff はこれらの一時的な Float オブジェクトの処理を C の double 型の処理に変換することで，Mandelbrot ベンチマークにおける Float オブジェクトの割当て，解放処理をほぼすべて除去している．これにより，浮動小数点演算が中心である Mandelbrot ベンチマークの速度が大きく向上している．

図 17 で Mandelbrot ベンチマークの次に高速化しているのは Tarai と Sieve ベンチマークであり，それぞれ 11.09 倍，5.63 倍高速化している．Tarai ベンチマークと Sieve ベンチマークの高速化度合いが大きいのは，これらのベンチマークでは，実行時間のほぼすべてが Ruby 処理系の仮想マシンによって費やされるためである．数値に対する加減乗除や配列の参照などの単純な処理では，Ruby 処理系の仮想マシンによる処理が中心となる．6.1 節で述べたように，CastOff が高速化することができるのは Ruby 処理系の仮想マシンの処理であるため，Tarai ベンチマークや Sieve ベンチマークは CastOff による高速化度合いが大きい．

Mandelbrot，Tarai，および Sieve ベンチマークを C によって記述し，実行した場合の実行時間を表 7 (C) に示す．表 7 より，Mandelbrot ベンチマークの実行時間は C の 1.6 倍程度に収まっているものの，Tarai では約 4.7 倍，Sieve ベンチマークでは約 15 倍，C のほうが高速である．これは，ガード（4.4 節）によるオーバヘッドや，次に述べるオーバヘッドによるものである．

まず，Tarai ベンチマークでは，Fixnum の範囲に収まっているかどうかの検査や，Ruby の Fixnum オブジェクトから C の long 型への変換が生じる（計算で生成した値が表 3 以外のメソッドで使用されるため，4.6 節の型変換の削減対象とならない）．そして，Sieve ベンチマークでは，配列参照や配列への代入に，Ruby 処理系が提供する配列操作の関数を使用する．この配列操作の関数では，配列参照の添字が配列のサイズ内に収まっているかどうかの検査や，配列に対する代入が許可されているかどうかの検査，配列の要素が共有されていないかどうかの検査を行う．

Tarai ベンチマークではメソッド呼び出しと整数演算が，Sieve ベンチマークでは配列処理が，それぞれ中心となる処理である．このため，これらのベンチマークではいまだに C との大きな速度差が生じていると考えられる．

Renderer ベンチマークと Nbody ベンチマークはともに，Mandelbrot ベンチマークと同様に浮動小数点演算が中心である．しかし，Mandelbrot ベンチマークが 91.56 倍高速化しているのに対し，Renderer ベンチマークと Nbody ベンチマークの高速化の度合いはそれぞれ 1.91 倍，1.99 倍である．これは，Renderer ベンチマークや Nbody ベンチマークでは，生成した Float オブジェクトをインスタンス変数などから参照するため，CastOff が 6.1 節で述べた Float オブジェクト割当ての削減に失敗しているためである．今後，オブジェクトの割当てと解放を高速に行えるような仕組みを導入することで，オブジェクト割当ての削減に失敗したオブジェクトも高速に扱えるようにしたいと考えている．

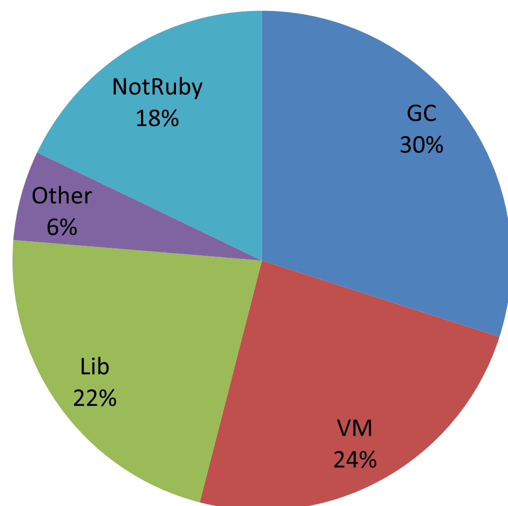


図 18　RDoc ベンチマークのプロファイル結果
Fig. 18　Profiling result of RDoc benchmark.

他のベンチマークがすべて高速化しているのに対し，Empty ベンチマークは CastOff を通して実行することで速度が低下している．これは，6.1 節で述べたように，コマンドラインツール `cast_off` に与えられた引数の処理や，Ruby で記述された `cast_off` のコンパイラ本体の読み込み時間のオーバヘッドによるものである．現状の CastOff は，実行時間が極端に短いプログラムを扱うのに適していないといえる．

最後に，RDoc ベンチマークは 8.91%高速化している．RDoc ベンチマークは，1 万行規模の Ruby プログラムである RDoc を使用するベンチマークである．このことから，CastOff は現実的なプログラムにも適用できるコンパイラであるといえる．RDoc ベンチマークの高速化度合いは 8.91%と小さい．

これは，RDoc ベンチマークのボトルネックに，現在の CastOff によって高速化できる処理以外のものが存在するためである．**図 18** に，RDoc ベンチマークを CRuby を用いて実行し，oprofile [23] というツールでプロファイルを行った結果を示す．図 18 では，VM が CRuby の仮想マシン関連の処理，GC が CRuby の GC 関連の処理，Lib が文字列処理や正規表現などの CRuby のライブラリの処理，Other が CRuby のその他の処理，NotRuby が CRuby

© 2012 Information Processing Society of Japan

以外での処理時間の割合を示している．図 18 より，RDoc のベンチマークでは，Ruby 処理系の仮想マシンの実行時間は 24%であり，76%が仮想マシン以外の処理に費やされていることが分かる．

CastOff が高速化できるのは，特定のオブジェクトに対する処理や，Ruby 処理系の仮想マシンの処理が中心であることは 6.1 節で述べた．しかし，RDoc では Ruby 処理系の仮想マシンの処理だけでなく，GC や Ruby 処理系の組み込みメソッドの処理にも多くの時間が費やされる．

4.6 節で述べた「冗長な文字列リテラル複製の削減」や「C と Ruby 間の型変換の削減」により，CRuby のライブラリの処理や GC 関連の処理時間を削減することは可能である．しかし，CastOff がこれらの手法を適用できるのは，文字列リテラルや，Fixnum オブジェクト，Float オブジェクトなどの特定のオブジェクトに限られるため，RDoc のように様々な種類のオブジェクトを扱うプログラムへの影響は小さい．さらなる最適化を実装し，現状の CastOff では扱えない処理も高速化の対象とすることが，CastOff の今後の課題の 1 つである．

6.2.2 プロファイル実行とコンパイルの速度の評価

CastOff のプロファイル実行とコンパイルにかかった時間を，表 7（プロファイル実行 + コンパイル）に示す．表 7 より，CastOff におけるプロファイル実行とコンパイルのオーバヘッドが大きいことが分かる．これは，実装の完成度による影響が大きい．プロファイル実行とコンパイルのうち，オーバヘッドが特に大きいのはプロファイル実行である．

まず，現状ではプロファイル用コード生成時に，型情報を収集するコードを削減する工夫を行っていないため，型情報の収集処理がプログラム中のいたるところで実行される．プログラム中の各点でどのような型情報を収集するかを伝播させ，すでにプロファイル用のコードが実行されていると分かった場合にプロファイル用コードの生成をスキップすることで，この問題は解決できると考えている．

また，CastOff は，メソッドの呼び出し回数をプロファイルするために，メソッド呼び出しを監視する（4.2 節）．このメソッド呼び出しの監視処理で，CastOff は Ruby で記述した集計処理を実行する．このため，呼び出す対象のメソッドが小さい場合，メソッドの呼び出し回数の集計処理がボトルネックとなる場合がある．現状では CastOff の大半の処理を Ruby によって記述しているため，このようにボトルネックとなりうる処理は，段階的に C などの高速な言語による記述に置き換えようと考えている．

コンパイルのオーバヘッドもプロファイル実行のオーバヘッドほどではないが大きい．これは，先述のように，CastOff の大半の処理を Ruby によって記述しているためである．コンパイル処理は脱最適化にともなう再コンパイル時にも発生するため，最悪の実行時間に与える影響は大きい．

本評価の計測時には，プロファイル実行とコンパイルを終えた直後の実行で，RDoc ベンチマークに 17 回の脱最適化と再コンパイルが生じている[*1]．この結果，実行時間が約 193 秒となった．これに対し，脱最適化と再コンパイルが生じない場合の RDoc ベンチマークの実行時間が約 118 秒である（表 7）．つまり，17 回の脱最適化と再コンパイルにより，約 75 秒のオーバヘッドが生じていることが分かる．

このため，C による記述に置き換え，コンパイルの速度を向上させるだけでなく，コンパイルの回数をできるだけ減らすことが望ましい．これに対し，脱最適化にともなう再コンパイルが繰り返し発生するような場合は，インタプリタ実行に切り替えてプロファイル情報をとり，まとめて再コンパイルを行うなどの工夫が必要であると考えている．

7. 関連研究

本研究と同様に，既存の処理系に手をくわえることなく動作するスクリプト言語用コンパイラはすでに存在する [13], [14]．しかし，これらの先行研究では，処理系にどのような機能が必要になるかの議論がなされていない．本稿では，スクリプト言語処理系に手を加えることなくコンパイラを開発するうえで処理系にどのような機能が必要になるかを，CastOff という実例を用いて議論している．また，先行研究 [13] は Python を，先行研究 [14] は PHP を対象としている．これに対し，本研究では Ruby を対象にコンパイラの設計と実装手法をまとめた．

CastOff に先んじて，Ruby から C への変換を行うコンパイラはいくつも存在する [3], [4], [5], [19]．Ruby2C [19] は，コンパイル対象の Ruby スクリプトに強い制限を加え，解析し，実行前に C 言語ソースコードへ変換する．Ruby2C が解析によって得られる情報を使用するのに対し，CastOff は，ユーザからの情報やプロファイルによる情報を活用する．これにより，解析が困難なプログラムに対しても，CastOff は容易に適用することができる．Ruby2C と CastOff をコード生成の面で比較すると，Ruby2C は基本的に，Ruby2C に登録された変換対象となる Ruby の構文に対して等しい C へと変換を行う（たとえば Ruby の == や [] に対して C の == や []）．対象となる C の構文がなかった場合は，CastOff のように Ruby のメソッドディスパッチャを使用する．これに対し，CastOff はロード時にメソッド検索を行い，C の関数ポインタを取得することで，CastOff にメソッドを登録することなく，広く脱仮想化を適用している．

五嶋らによるコンパイラ [4]，yarvaot [5]，我々によるコ

*1 CastOff がメソッドの実行を監視し，型情報を収集するのは，メソッドが閾値を超えた回数実行されてからであるため，型情報の収集を開始する前の情報が抜け落ちてしまう．

© 2012 Information Processing Society of Japan

ンパイラ [3] は，CastOff と同様に，RubyVM から得られるバイトコード列を C に変換する．CastOff はこれらのコンパイラをさらに発展させ，実行時コンパイルや再コンパイル，ユーザからのアノテーションをサポートしている．

JRuby [9]，Rubinius [10]，MacRuby [11] は，Ruby プログラムの高速化を目標の 1 つとして掲げた Ruby 処理系である．これらのプロジェクトは，処理系や評価器そのものを新しく開発するプロジェクトである．これに対し，CastOff は，ライブラリとしてすでに存在する Ruby 処理系を拡張する形で，Ruby プログラムの高速化を目指した．

プロファイル情報をコンパイルに活用するスクリプト言語用コンパイラとしては，TraceMonkey [22] があげられる．TraceMonkey は実行のたびにプロファイル情報の収集とコンパイルを行う．このため，TraceMonkey のユーザは，プロファイルやコンパイルのための実行と，実際の実行を区別することなく利用することができる．これに対し，CastOff ではプロファイル情報の収集やコンパイルは，実際の実行を行う前にユーザがあらかじめ明示的に行うことを前提としている．

本研究の目標の 1 つとして，既存の処理系をベースとして Ruby プログラムの実行を高速化するという点があげられる．Yeti [15] という Java 用のコンパイラでは，コンパイラを持たない処理系を拡張し，JIT コンパイラを組み込むための手法についてまとめている．これに対し，本研究では処理系に手を加えることなく，ライブラリとしてコンパイラを実装するという手法を採用し，そのために処理系が提供すべき機能について論じた．

8. おわりに

本稿では，Ruby 処理系に手を加えることなく，C 拡張ライブラリとして実装した CRuby 用コンパイラ CastOff について，設計と実装を詳細に解説した．そして，CastOff の実装をふまえて処理系が提供すべき機能を議論した．CastOff は，実行時コンパイル，プロファイル実行，アノテーションのサポート，脱最適化，再コンパイル，コンパイル済みコードの再利用などの機能を持つ．CastOff は，これらの機能を CRuby 処理系にいっさいの変更を加えることなく提供する．CastOff を用いることで，ユーザは容易に Ruby プログラムの実行を高速化することができる．

評価の結果，浮動小数点演算が中心のベンチマークを最大で 91.56 倍高速化することができた．また，1 万行規模のプログラムである RDoc を 8.91%高速化することができており，CastOff が現実的なプログラムにも適用できるコンパイラであることが確認できた．しかし，RDoc では，CastOff の高速化の対象である Ruby 処理系の仮想マシンの処理だけでなく，GC や Ruby 処理系の組み込みメソッドの処理にも多くの時間が費やされる．このため，RDoc の高速化度合いは 8.91%と小さい．

これをふまえて，CastOff の今後の課題として，Ruby 処理系の VM だけでなく，GC やライブラリまで含めたオーバヘッドの削減が必要である．短寿命オブジェクトに対して CastOff 側で高速に割当て，解放を行うなどの工夫や，メソッドインライニングやデータフロー最適化を適用することで，さらなる速度向上を目指したい．

謝辞　本研究は科学研究費補助金（課題番号 21220001）の助成を受けたものである．

参考文献

[1] オブジェクト指向スクリプト言語 Ruby，入手先 ⟨http://www.ruby-lang.org/ja/⟩.

[2] cast_off — RubyGems.org — your community gem host, available from ⟨http://rubygems.org/gems/cast_off⟩.

[3] 芝　哲史, 笹田耕一, 卜部昌平, 松本行弘, 稲葉真理, 平木　敬：実用的な Ruby 用 AOT コンパイラ，情報処理学会第 80 回プログラミング研究会 (2010. 8).

[4] 五嶋宏通, 笹田耕一, 三好健文, 稲葉真理, 平木　敬：Ruby 用仮想マシンにおける AOT コンパイラ，情報処理学会論文誌 プログラミング，Vol.2, No.1, pp.21–21 (2009).

[5] shyouhei's ruby at shyouhei/yarvaot – GitHub, available from ⟨http://github.com/shyouhei/ruby/tree/shyouhei%2Fyarvaot⟩.

[6] 青木峰郎：Ruby ソースコード完全解説，インプレス (2002).

[7] 笹田耕一，松本行弘，前田敦司，並木美太郎：Ruby 用仮想マシン YARV の実装と評価，情報処理学会論文誌：プログラミング，Vol.47, No.2, pp.57–73 (2006).

[8] 須永高浩，笹田耕一：Ruby 用リアルタイムプロファイラの設計と実装，情報処理学会論文誌 プログラミング，Vol.4, No.3, pp.1–15 (2011).

[9] JRuby.org, available from ⟨http://jruby.org/⟩.

[10] Rubinius: Use Ruby, available from ⟨http://rubini.us/⟩.

[11] MacRuby ≫ Home, available from ⟨http://www.macruby.org/⟩.

[12] v8 — V8 JavaScript Engine — Google Project Hosting, available from ⟨http://code.google.com/p/v8/⟩.

[13] Rigo, A.: Representation-based just-in-time specialization and the psyco prototype for python, *PEPM '04, Proc. 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2004).

[14] Biggar, P. and de Vries, E. and Gregg, D.: A practical solution for scripting language compilers, *SAC '09, Proc. 2009 ACM symposium on Applied Computing* (2009).

[15] Zaleski, M., Brown, A.D. and Stoodley, K.: YETI: a graduallY Extensible Trace Interpreter, *VEE '07, Proc. 3rd international conference on Virtual execution environments* (2007).

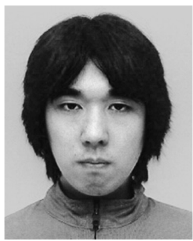
[16] Furr, M., An, J.-H. (David) and Foster, J.S.: Profile-guided static typing for dynamic scripting languages, *OOPSLA '09, Proc. 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (2009).

[17] Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc. (1986).

[18] Holzle, U., Chambers, C. and Ungar, D.: Debugging optimized code with dynamic deoptimization, *PLDI '92, Proc. ACM SIGPLAN 1992 conference on Programming language design and implementation* (1992).

© 2012 Information Processing Society of Japan

[19] RubyForge: ruby2c — ruby to c translator: Project Info, available from ⟨http://rubyforge.org/projects/ruby2c/⟩.
[20] RubyGems.org — your community gem host, available from ⟨https://rubygems.org/⟩.
[21] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A study of devirtualization techniques for a Java Just-In-Time compiler, *OOPSLA '00, Proc. 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2000).
[22] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M. and Franz, M.: Trace-based just-in-time type specialization for dynamic languages, *PLDI '09, Proc. 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009).
[23] OProfile — A System Profiler for Linux (News), available from ⟨http://oprofile.sourceforge.net/news/⟩.



芝 哲史　（正会員）

1985年生まれ．2009年千葉大学理学部数学･情報数理学科卒業．2011年東京大学大学院情報理工学系研究科博士前期課程創造情報学専攻修了．2012年同研究科博士後期課程創造情報学専攻退学．2012年株式会社ドワンゴ入社（現職）．プログラミング言語処理系の研究に興味を持つ．

笹田 耕一　（正会員）

2004年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006年同大学院工学教育部博士後期課程電子情報工学専攻退学．博士（情報理工学）（東京大学情報理工学系研究科，2007年）．2006年東京大学情報理工学系研究科助手，2008年同講師，2012年 Heroku, Inc. にて Ruby 処理系の開発に従事（現職）．システムソフトウェア，特に並列処理システム，言語処理系に関する研究に興味を持つ．

平木 敬　（正会員）

東京大学理学部物理学科，東京大学大学院理学系研究科物理学専門課程博士課程退学，理学博士．工業技術院電子技術総合研究所，米国 IBM 社 T.J.Watson 研究センターを経て，現在，東京大学大学院情報理工学系研究科勤務．数式処理計算機 FLATS，データフロースーパーコンピュータ SIGMA-1，大規模共有メモリ計算機 JUMP-1 等多くのコンピュータシステムの研究開発に従事，現在は超高速ネットワークを用いる遠隔データ共有システム Data Reservoir システムの研究，超高速計算システム GRAPE-DR の研究を行っている．

© 2012 Information Processing Society of Japan