Ricsin: RubyにCを埋め込むシステム

笹 田 耕 一 †

スクリプト言語 Ruby は，その記述の容易さから世界中で広く利用されているオブジェクト指向プログラミング言語である．C 言語を用いて実装されている Ruby 処理系は Ruby C API を提供しており，Ruby では記述できない機能拡張や，性能のボトルネックとなるメソッドを C で実装する，といったことが可能である．しかし，C で機能拡張を行う場合，メソッド単位で C 言語を記述する必要があり，Ruby プログラムの一部を C で記述するようなことはできない．また，Ruby から C のような言語をまたぐプログラムの呼び出しには余計なオーバヘッドが必要になるという問題がある．そこで，我々は Ruby プログラムに C プログラム片を埋め込むためのシステム Ricsin を開発した．埋め込んだ C プログラム片からは Ruby プログラムのローカル変数などのコンテキスト情報へのアクセスが可能である．本システムを用いることで，Ruby の記述性の高さと C 言語の強力な機能を組み合わせることができる．また，Ruby 処理系に専用の機構を導入することで言語間の呼び出しオーバヘッドを抑える．本発表では Ricsin のアーキテクチャについて述べ，実装し評価を行った結果を述べる．

Ricsin: A System for “C Mix-in to Ruby”

Koichi Sasada †

Scripting language Ruby is an Object-Oriented programming language used in world-wide because of its ease of description. The Ruby interpreter written in C supports ”Ruby C API”, which allows to write an extension in C and to replace a performance bottleneck method with C. However, to make an extension written in C, a whole method written in C is needed. In other words, even if only a part of program should be written in C, then whole method should be implemented in C. Moreover, an overhead of foreign function call between Ruby and C is also a problem. On this background, we develop Ricsin: a system to support ”C Mix-in to Ruby”. Ricsin enables to embed a part of C program in a Ruby program. An embedded part of C program can access to a Ruby program context such as local variables. Using Ricsin, we can combine an easy-writing nature of Ruby and powerful features of C. Moreover, we propose the method to avoid foreign function call overheads using collaboration with Virtual Machine. In this presentation, we will describe Ricsin architecture and show the implementation and the result of evaluation of Ricsin.

1. はじめに

スクリプト言語 Ruby[13)~15)] は、手軽にオブジェクト指向プログラミングを実現するための種々の機能を持つプログラミング言語である．その使いやすさ[17)] から，Ruby は世界中で広く利用されており，多くのユーザを擁すプログラミング言語となっている．

現在，もっとも広く利用されている Ruby 処理系は，松本らによる実装[15)] である．この処理系はほぼ C 言語で記述されていることから CRuby とも呼ばれる．本稿では，この CRuby を対象に議論を行う．CRuby の最新版では，筆者らによってバイトコード実行型仮想マシン（VM）が搭載される[18)] など，高速化，高機能化のための開発が継続して行われている．Ruby 処理系は長らく CRuby のみであったが，近年 Java での実装である JRuby[11)]，C#による実装である IronRuby[9)]，Ruby による実装である Rubinius[12)] などが登場している．

他の Ruby 処理系と違い，CRuby は C 言語で処理系を拡張するためのインタフェースである Ruby C API と C 拡張ライブラリの仕組みを備える．Ruby C API を利用した C プログラムにより C 拡張ライブラリを作成し，CRuby を拡張することで，Ruby のみを用いた場合には記述が不可能な機能を実現することができる．C 拡張ライブラリは，具体的には (1) C 言語で記述されたライブラリや OS のシステムコールの利用 (2) Ruby C API のみで提供される CRuby のコ

† 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

ア機能の利用 (3) C で記述することによる処理の高速化などを目的として開発される．

C 拡張ライブラリでは，Ruby C API を利用して Ruby のメソッド呼び出しと C の関数呼び出しを結び付けることで，Ruby から C の関数を呼び出すことを実現する．C 拡張ライブラリは動的リンクライブラリとして実現されており，CRuby 実行中に読み込み，動的に処理系の機能を拡張することができる．

CRuby は C 拡張ライブラリを C 言語で処理を記述しやすくするために次のような工夫を行っている．例えば，ガーベージコレクションを保守的 GC とする，C 言語で Ruby の例外処理を記述できる，C 言語で Ruby のメソッドを C の関数呼び出しで行うための Ruby C API を提供する，ジャイアントロックを設けて暗黙のスレッド切り替えに対する同期などの対処を不要とする，などである．これらの工夫により，C 拡張ライブラリは Java 処理系での JNI[5] による C 拡張などに比べて記述しやすいものとなっている．

しかし，C 拡張ライブラリには C 言語で記述できる最小単位を Ruby のメソッド単位としているため，次の 2 つの問題がある．第一に，プログラムのごく一部を C 言語で記述する必要がある場合でも，その部分をメソッドとして括りだし，C 拡張ライブラリを生成するために別のファイルに移すという作業が必要になる点である．プログラムの部分を別のファイルに移すため，手間もかかり，プログラム全体の見通しも悪くなる．第二に，Ruby のメソッドとして C の機能を呼び出すため，Ruby のメソッドフレームの生成などの処理が発生し，呼び出しごとにオーバヘッドが発生する．

そこで，我々は Ruby スクリプトに C 言語による記述を埋め込むための Ricsin というシステムを開発した．Ricsin は，**図 1** のような Ruby と C を混在させたプログラムを受け取り，CRuby で実行可能な C 拡張ライブラリを生成する．Ricsin では，メソッド単位ではなく，Ruby スクリプトに C の文の直接埋め込みを可能とすることで，C 言語を利用する Ruby プログラムの開発を容易にした．C プログラム片からは Ruby の変数などのコンテキスト情報に直接アクセスできるようにすることで Ruby と C の混在プログラムの記述性を向上した．また，CRuby の VM を改造し，C プログラム片を直接呼び出すことにしたため，メソッド呼び出しのオーバヘッドも不要とした．

類似システムは存在するが，本研究の貢献は次の通りである．まず，Ruby のような動的スクリプト言語に C 言語の文を直接埋め込むための言語仕様を提案

```
# Ruby での記述
def open_fd(path)
  fd = __C__(%q{
    /* C での記述 */
    return INT2FIX(open(RSTRING_PTR(path), O_RDONLY));
  })
  raise 'open error' if fd == -1
  yield fd
ensure
  raise 'close error' if -1 == __C__(%q{
    /* C での記述 */
    return INT2FIX(close(FIX2INT(fd)));
  })
end
```

図 1 Ricsin による Ruby と C 混在プログラムの例
Fig. 1 An Example of Ruby and C Mixed Program

した点，そして記述性を高めるために Ruby（埋め込み先言語）のコンテキスト情報へ直接アクセスするための手法，および記法を提案した点である．そして，VM 命令の追加により，C 言語機能を呼び出すためにメソッド呼び出しのようなオーバヘッドを排除し，高速な呼び出しを可能とした点である．また，実社会の貢献として，広く利用されている Ruby の機能拡張を容易にする実用的なシステムを実装し利用可能としたという点が挙げられる．

本稿では，この Ricsin システムについて設計と実装，評価を述べる．以下，2 章で現状と問題分析を行い，3 章で Ricsin システムの概要を述べる．4 章で Ricsin での Ruby，C 混在プログラムの記述手法を説明し，5 章で実装について述べる．6 章で評価を行い，7 章で本システムの妥当性に関する議論を行う．8 章で関連研究を述べ，9 章でまとめる．

2. Ruby に C を埋め込む手法の検討

1 章で述べた通り，CRuby では C 拡張ライブラリにより処理系を拡張することができるが，記述性や性能に問題が残る．

本章では Ruby スクリプトに C 言語によるプログラム片を埋め込む方式を検討するため，現在の C 拡張ライブラリの記述方式を述べ，その問題点を述べる．また，他言語での例を示し，既存の CRuby の仕組みの問題点についてまとめる．

2.1 C 拡張ライブラリの開発

CRuby は C 拡張ライブラリによって処理系を拡張することができる．

C 拡張ライブラリでは，基本的に 1 つ以上の C ソースファイルに次のものを定義する．

(1) 1 個のロード時の初期化関数
(2) 0 個以上の C メソッドを定義する関数

初期化関数は C 拡張ライブラリ呼び出し時に一度だけ呼ばれる．name という拡張ライブラリの場合，`Init_name()` という名前の関数として定義しておき，CRuby の動的リンク機能により呼び出される．

初期化関数で Ruby C API によってクラス定義やメソッド定義を行い，CRuby の機能を拡張する．

Ruby C API によるメソッド定義は，C の関数を Ruby から呼び出すことのできるメソッドとして関係づけることによって行われる．つまり，定義したメソッドを呼び出すと，C の関数が呼び出される．現在の CRuby では，とくに断りのない限り，C 関数を実行中はスレッド切り替えが起こらないことを保証しているため，並列実行における同期を行う必要はない[19]．ここで登録するメソッドを C メソッドと呼ぶことにする．

Ruby C API には，このようなクラス定義，メソッド定義を行う処理以外にも，数値や配列，文字列などの代表的なオブジェクトの生成や操作をする関数やマクロが多数用意されている．例えば，C の int 型の値を Ruby の Fixnum オブジェクトへ変換する場合は `INT2FIX()` マクロを利用する．

Ruby のオブジェクトは，C 言語では `VALUE` という型で扱われる．また，`nil` や `true`，`false` などの特殊な Ruby オブジェクトの値を示す `Qnil`，`Qtrue`，`Qfalse` が提供されている．

CRuby は保守的ガーベージコレクションを行うことで，マシンスタックもしくはマシンレジスタ上にのみポインタのある Ruby オブジェクトも特別な記述行わずにマークされることを保証している☆．

C 拡張ライブラリを作成するときには，慣例として extconf.rb という Ruby スクリプトを作成し，mkmf.rb というライブラリの機能を利用して Makefile を生成する．extconf.rb には，必要となるライブラリやヘッダファイルを mkmf.rb の機能を利用して指定する．生成された Makefile を用いてビルドを行う．

2.2 既存の C 拡張ライブラリの問題点

C 拡張ライブラリには次の問題点がある．第一に，メソッド単位で C 言語の処理を記述する必要があるという点である．

Ruby から C の機能を利用するためには，C メソッドにして呼び出さなければならない．つまり，メソッドより小さい粒度では C 言語の機能を利用することができないことになる．プログラム中に C 言語で記述する必要のある部分がごく一部であった場合，その部分をメソッドとして切り出し，その切り出した部分を C ソースファイルで実装しなければならない．これは，プログラム全体の見通しを悪くする．

この問題に対応するため，Ruby には RubyInline[3] というライブラリが存在する．このライブラリを利用すると，Ruby スクリプトの中に C の関数を記述し，それを C メソッドとして登録することができる．このライブラリを利用することで別のファイルへ処理を記述するという手間と見通しの悪さを改善することはできるが，C のプログラムが必要となる箇所に直接 C の文を埋め込むような，より密接な言語間の連携はできない．また，メソッドとして処理を切り出す必要がある点も同様に問題である．

また，性能の問題がある．C の機能は Ruby のメソッド呼び出しの機構のもとで行われるため，C で実装した機能を利用するごとにメソッドフレームの生成のようなオーバヘッドが発生する．また，Ruby C API を利用した Ruby のメソッド呼び出し機能を利用することで C から Ruby で定義した機能を利用することができるが，Ruby の VM の起動のようなオーバヘッドが同様に発生するのも問題である．

3. Ricsin の設計

2 章で述べた現在の C 拡張ライブラリ開発の問題点である次の 2 点，Ruby スクリプトに C の文を気軽に埋め込むことが出来ないこと，および C の機能を利用するためにメソッド呼び出しのオーバヘッドが必要になる点を解決するため，我々は Ricsin システムを開発した．

Ricsin は，Ruby スクリプトに C プログラムを埋め込んだプログラムを入力として受け取り，CRuby で実行可能な C 拡張ライブラリを出力する．埋め込む C プログラム片はメソッドよりも小さい単位で混在させることが可能である．また，C のプログラム片をメソッド呼び出しではなく，VM の命令によって直接呼び出すようにしているため，高速な呼び出しが可能である．

本章では Ricsin の利用方法や例を紹介する．また，Ricsin によって行われる具体的な変換について述べ，Ricsin システムの全体像を示す．

3.1 Ricsin の利用方法

本節では Ricsin をどのように利用するのか，その概要を説明する．

☆ 実際は C コンパイラの最適化によってポインタがマシンスタックもしくはマシンレジスタから存在しなくなる場合もありマーク漏れが生じることがある．この問題は，本研究の成果を利用しづらくする可能性があるため，今後検討していきたい．

3.1.1 rcb ファイルの用意

まず，Ruby スクリプトに C プログラムを混在させたプログラムを記述したファイルを用意する．通常の Ruby スクリプトや C プログラムと区別するために，ファイルの拡張子を rcb とし，これを rcb ファイルと呼ぶ．

例えば，図 1 で示した Ruby と C を混在した rcb ファイルを用意する．記述手法の詳細は次章で述べるが，`__C__(%q{...})` の `...` の部分が C プログラム片である．rcb ファイルは，基本的に Ruby スクリプトとして構文解析が可能である．このとき，Ruby プログラムとして見ると `__C__(...)` はメソッド呼び出しとなり，`%q{...}`は文字列リテラルにあたる．Ricsin によって`__C__`が記述されていた箇所で文字列に記述されている C プログラム片を実行するように変換される．

`__C__`を識別子として C の式を直接埋め込むことができるので，プログラムの見通しをよくする．これは，手続きを値として取り扱う時，メソッドや関数などの形で別の場所で定義したものを参照する C や C++，Java のような言語に比べ，処理をクロージャとして必要な箇所に記述できる Lisp や Ruby のような言語のほうが記述性，可読性が高いという点と同様である．

C プログラム片は，C 拡張ライブラリの記述とまったく同等の記述をする．例えば，`VALUE` 型で Ruby オブジェクトを取り扱い，Ruby C API を利用して CRuby の内部機能を利用する．保守的 GC により，マシンスタック上にポインタが存在する Ruby オブジェクトはマークされる．

`__C__`以外の，Ricsin とは関係ないメソッド呼び出しなどを含む Ruby プログラムはすべて通常の CRuby で実行するときと同様に実行される．

3.1.2 変換の実行と利用

Ricsin 変換器は rcb ファイルを受け取ると，`__C__` などを解析した結果をもとに C ソースファイルを生成する．生成された C ソースファイルは，C 拡張ライブラリとしてビルドするために必要なヘッダファイル，ロード時に実行するための初期化関数を含む．

生成された C ソースファイルは実行環境に付属する C 言語ビルド環境（コンパイラ，リンカ等）を利用してビルドし，Ruby 用 C 拡張ライブラリを生成する．もし，rcb ファイルと同じ位置に `extconf.rb` というファイルがあればそれを実行して `Makefile` を生成し，ビルドを行う．つまり，もしコンパイラやリンカ等に指定しなければならないオプション，例えば挿入するべきヘッダファイルやリンクするべきライブラリがあったときには，`extconf.rb` ファイルにそれを利用する旨を記述することで，オプションを利用してビルドすることができる．なければ，デフォルトの Makefile を生成しビルドする．

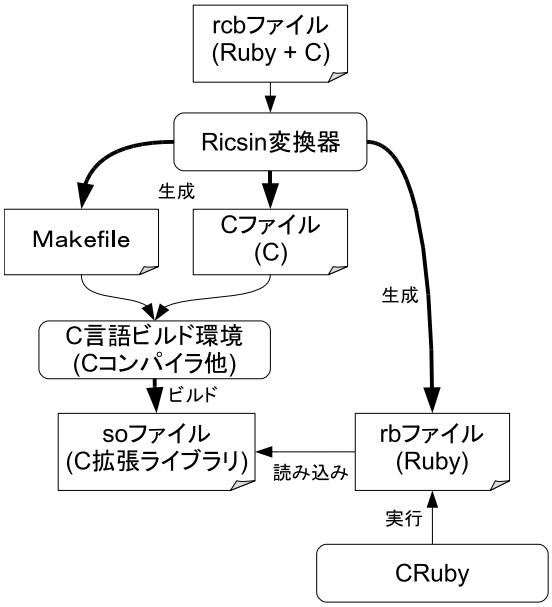

図 2 Ricsin の全体像
Fig. 2 Overview of Ricsin

生成された C 拡張ライブラリをロードすることで，rcb ファイルに記述されていた処理が CRuby に登録される．登録された処理は Ruby プログラムから起動することができる．なお，生成された C 拡張ライブラリを読み込み，登録された処理を起動するだけの Ruby スクリプトも同時に生成する．rcb ファイルに記述されたプログラムを実行するためには，その生成された Ruby スクリプトを従来の Ruby スクリプトと同様に実行する．プログラムを配布する場合は，生成された C 拡張ライブラリと Ruby スクリプトのみを配布すればよい．

これらの入出力対象となるファイルと，その操作をまとめた図を**図 2** に示す．Ricsin 変換器は rcb ファイルを読み込み，C ソースコード（C ファイル）経由での C 拡張ライブラリ（so ファイル）生成と，Ruby スクリプト（rb ファイル）の生成を行う．実行時には生成された rb ファイルを実行することで，so ファイルが読み込まれ，rcb ファイルに記述したプログラムが実行される．

3.1.3 Ricsin の利用シーン

Ricsin では次のような利用シーンを想定している．

第一に，従来の C 拡張ライブラリの開発を Ricsin

を用いて行うという例が挙げられる．Ricsin を用いることで，C の機能を利用する C 拡張ライブラリのより効率的な開発が可能となる．

既存の C ライブラリの単純なラッパを作成するような場合には SWIG[1] などのシステムを利用して機械的に生成することも可能である．しかし，C ライブラリの機能を，Ruby のプログラミングモデルに適したインターフェースをもったライブラリとして提供するためには，複雑な C の記述を行わなければならないことが多い．例えば，例外処理やイテレータを用いたライブラリ設計において，C での複雑な記述が必要になる．この問題に対して，従来は単純な C ライブラリのラッパを C 拡張ライブラリとして開発し，それを利用する Ruby スクリプトを記述するという開発スタイルがとられることが多い．

Ricsin を用いることで，Ruby スクリプトに C プログラム片を埋め込み，Ruby の言語機能を用いつつ C のライブラリ機能を用いることにより，そのような冗長な開発手順を踏む必要がなくなる．

第二に，Ruby スクリプトの性能ボトルネックを段階的に C プログラムで記述することで，必要に応じた性能向上を行うという使い方がある．一般的に，ソフトウェアの実行時間はプログラムの一部に集中していることが知られている．この一部をプロファイラ等で発見し，該当箇所を C プログラム片で置き換えることでソフトウェアの実行時間の短縮を試行することができる．

高速化のために C プログラムを利用する場合，従来はそのボトルネック部分をメソッドとしてまとめ，そのメソッドを C プログラムで書き直し，そのスクリプトとは別途 C プログラムを記述した C ソースファイルを用意し，C 拡張ライブラリとしてビルドして利用する，という手順が必要であった．このような手間が必要となるため，C 拡張ライブラリを利用する高速化は気軽に行うことができなかった．

Ricsin では，Ruby スクリプトの一部を C に手軽に置き換えることができるため，試行錯誤を行いながら必要な性能に応じて C 言語化を進めることが可能になる．また，Ricsin がツールとして C 拡張ライブラリのビルドまで自動で行うため，従来必要であった C 拡張ライブラリのビルドの手間を省くことができる．

さらに，高速化に関する性能面でも利点がある．まず，Ruby スクリプトに埋め込むため，そのコンテキストに関する情報を利用した最適化が可能である．例えば，ある変数には整数が格納されていることが確実であるとプログラム記述者が判断すれば，C プログラム片ではそれを前提とした処理を記述することができる．一般的に，C メソッドを記述する場合，メソッドという単位の一般性から，引数などにどのような型の値が格納されているかを静的に検知する方法はない．そのため，すべての変数について型のチェックを行う必要がある．Ricsin で C プログラム片を埋め込んだ場合，メソッドと違いその箇所でしか利用されないことがわかっているため，その箇所に関する知識を利用した最適化が可能になる．

また，C プログラムで記述する範囲をメソッド呼び出しの単位でまとめる必要がないので，C メソッドの呼び出しで必要となるメソッドフレーム作成などのオーバヘッドは不要になる．また，次章で述べる `__Ccont__`による埋め込みを利用することで，C から Ruby プログラムを呼び出すオーバヘッドが削減可能となる．

なお，どのような Ruby スクリプトをどのように書き換えれば高速化するかというのは CRuby の実装に強く依存するため自明ではないが，例えば数値計算部分を C に書き換えると高速化することが多い．

その他，Ruby スクリプトに容易に C プログラム片を記述できることから，CRuby の Ruby C API のテストを記述するのに利用できるのではないかという提案もされている．つまり，Ruby で開発された使いやすいテストフレームワーク上で，CRuby の内部機能のテストを記述することができる．

4. Ricsin の記法

本章では，Ricsin による rcb ファイルの記法について述べる．ここで，Ruby の言語仕様，および C の言語仕様を拡張した記法を Ricsin 記法と称する．Ricsin 記法とは，Ruby 言語を拡張して C プログラム片を埋め込むための Ruby レベル Ricsin 記法と，C 言語を拡張して Ruby コンテキスト情報へアクセスするための C レベル Ricsin 記法の 2 つからなる．

rcb ファイルは通常の Ruby スクリプトとして有効な文法となるように Ricsin 記法を設計した．この方式をとることで，rcb ファイルを事前に Ruby 文法チェッカ等にかけ，変換前に記述ミスなどをチェックすることが可能となる．ただし，そのまま CRuby で処理しても正しく動作しないため，Ricsin によって変換する必要がある．

なお，Ricsin 記法による C 埋め込みを利用しない限り，rcb ファイルは通常の Ruby スクリプトと全く同じ表現力をもつ．

表 1 Ricsin 記法で用いるセレクタ一覧
Table 1 Selectors List of Ricsin Syntax

セレクタ	機能
__C__	C プログラム片を埋め込み，実行する
__Cdecl__	C での宣言，定義を埋め込む
__Cinit__	ロード時に必要な初期化処理を埋め込む
__Cb__	C で Ruby のブロックを記述する
__Ccont__	C プログラム片に Ruby プログラム片を埋め込むための特別な記述

4.1 Ruby レベル Ricsin 記法

Ruby スクリプトに C を埋め込む Ruby レベル Ricsin 記法は，基本的に C プログラム片を記述した文字列を特定セレクタ（メソッド呼び出しに用いるシグネチャ）のメソッド呼び出しに渡すという表現で行う．この特別なセレクタの一覧とその効果を**表 1** に掲載する．

4.1.1 __C__による埋め込み

まず，最も基本的な埋め込み記述である__C__について述べる．__C__に渡された文字列は 0 個以上の C の文として解釈され，その箇所で実行するように変換される．C プログラム片は return 文によって値を返すことができ，これが__C__の返値となる．__C__の返値は，Ruby スクリプトでは通常のメソッド呼び出しの返値と同じように利用可能である．C プログラム片に return 文がなければ，__C__の返値は nil となる．

rcb ファイルは実行前に変換するため，__C__などの引数に文字列リテラルしか指定することは出来ない．たとえば，変数経由で渡したり，Ruby の言語仕様にある埋め込み文字列を指定することなどはできない．また，レシーバを指定した場合，Ricsin 記法ではなく通常のメソッド呼び出しとして解釈される．もし，Ruby スクリプトとして__C__メソッドを利用したい場合，レシーバを陽に指定すれば，通常の__C__というメソッド呼び出しとして扱われる．

なお，__C__というセレクタ名は多くの C コンパイラでサポートされているアセンブラ埋め込み文が__asm__というキーワードを利用していることから名付けた．

4.1.2 __Cdecl__による宣言・定義の埋め込み

埋め込む C プログラム片で利用する関数やマクロを定義，もしくは宣言したい場合は，__Cdelc__にその定義，宣言を記述する（**図 3**）．__Cdecl__で指定した C プログラムは，変換後の C ソースコードの冒頭に出現順に配置されるため，他の C 埋め込み文で利用する定義や宣言を記述することができる．__Cdecl__では他と違い，C の文のみを書くことはできない．__Cdecl__は複数記述可能で，それらは変換後の C プログラムの冒頭部に出現順に配置される．

```
__Cdecl__ %q{
  #define ANSWER 42
  int func(int x) {
    return ANSWER + x;
  }
}

puts(__C__('return INT2FIX(func(ANSWER));'))
#=> 84 と表示
```

図 3 __Cdecl__の利用例
Fig. 3 Usage of __Cdecl__

```
__Cinit__ %q{
  initialize_process();
} # ロード時に一度だけ実行される
```

図 4 __Cinit__の利用例
Fig. 4 Usage of __Cinit__

4.1.3 __Cinit__による初期化コードの埋め込み

C ライブラリの初期化など，ロード時に一度だけ行う処理がある場合，__Cinit__を利用することができる（**図 4**）．__C__と同様に C の文を記述することができる．このプログラム片はプログラムのロード時に一度だけ実行される．__Cinit__は複数記述可能であり，それらはロード時に一度だけ実行される．

4.1.4 __Cb__による C でのブロックの記述

Ruby ではメソッド呼び出しの引数のひとつとして，処理のまとまりを表現するブロックを渡すことができる．__Cb__を利用することで，このブロックを C プログラム片として記述可能である．**図 5** の (1) で示す Ruby スクリプトを，__Cb__を利用して C によるブロック記述を行ったのが (2) である．

__Cb__は，値をブロックとして渡すという&で始まる Ruby の特殊メソッド呼び出し形式として記述する．return 文はブロックの返値として扱われ，指定がなければ nil が返値となる．

従来，C でブロック処理を記述するには図 5 の (3) で示すような rb_block_call() という Ruby C API を利用する煩雑な方法しかなかったため，書きづらかった．__Cb__を利用することで，C で簡潔に記述することができるようになった．

4.1.5 __Cont__による C プログラム中への Ruby プログラム片の埋め込み

__Ccont__での C プログラム片埋め込みは特殊である．

例えば，C プログラム片に Ruby プログラム片を埋め込みたい場合を考える．いくつか実現手段はあるが，もっとも簡単な実現方法としては，その埋め込まれた

```
# (1) Ruby で記述した場合
[1, 2, 3].map{|arg|
  arg * 2
}

# (2) __Cb__ を利用した場合
[1, 2, 3].map(&__Cb__(%q{
  /* C で記述した場合 */
  return INT2FIX(FIX2INT(arg) * 2);
})) #=> [2, 4, 6] を返す

# (3) 従来の Ruby C API のみを利用して C で記述した場合
VALUE func_i(VALUE arg) {
  return INT2FIX(FIX2INT(arg) * 2);
}
VALUE map_double(VALUE ary) {
  /* ary に [1, 2, 3] が格納されているとする */
  return rb_block_call(
    ary, rb_intern("map"),
    0, 0, func_i);
}
```

図 5 __Cb__の利用例
Fig. 5 Usage of __Cb__

Ruby プログラム片を実行する Ruby VM をさらに起動するという手法である．実際，Ruby C API で C から Ruby のメソッド呼び出しを行う場合，そのように実行される．しかし，C から Ruby プログラムへの遷移は VM の起動を含めオーバヘッドがかかる．そこで，処理を VM に戻し Ruby プログラム片を実行し，また C プログラム片の実行を継続するという記述を可能とした．

__Ccont__に渡された C プログラム片は，同じスコープ上のその他の__Ccont__で指示された C プログラム片と連結される．そして，__Ccont__で挟まれた Ruby スクリプト片との実行を交互に行う**図 6** に示すような記述が可能になる．

この例の場合，(a) を実行し，変数 v が `nil` でないことを確認する．その後，Ruby スクリプト (b) を実行し，C レベルの (c) へ戻る．(c) では変数 v の値を出力し，(d) に移る．(d) は (a) の `while` 文の終端であるため，(a) へ戻る．(b) によって，v は `nil` が代入されているため，このプログラムはループを抜け終了する．

なお，この Ruby レベル変数 v への参照については次節で詳説する．

C プログラム片へ Ruby プログラムを埋め込みたい場合，__C__を複数並べるということも考えられるが，C の制御フロー中に Ruby プログラム片を埋め込む図 6 のような場合にはこれでは対応できない．そのため，__Ccont__を利用する．

__Ccont__を並べた記述は読みづらいため，行頭が

```
v = true
__Ccont__('while (v != Qnil) {') # (a)
  v = nil                        # (b)
__Ccont__('  rb_p(v);')          # (c)
__Ccont__('}')                   # (d)
```

図 6 __Ccont__の利用例
Fig. 6 Usage of __Ccont__

```
v = true
#C while (v == Qnil) { /* (a) */
     v = nil            # (b)
#    rb_p(v);          /* (c) */
#C }                   /* (d) */
```

図 7 `#C` 記法の利用例
Fig. 7 Usage of `#C` Notation

`#C cexpr` という形であれば，通常はコメントと解釈されるところを__Ccont__('cexpr') と書かれている，と解釈する記法も用意した．この記法を使うと，**図 7** のように見やすくなる．

一度 Ruby スクリプト片を実行すると，C 言語のローカル変数の内容は保存されないという制限がある．そのため，C プログラム片の各実行で共有する変数がある場合は，Ruby レベルのローカル変数に格納するなどの処置が必要になる．たとえば，図 6 の例では，Ruby レベルローカル変数 v に共有する情報をおいて C の `while` ループを実現している．

__Ccont__を利用することで，繰り返しなどの複雑な C プログラム片からの Ruby 処理の実行を，VM の起動オーバヘッドなく高速に実行することができる．

4.2 C レベル Ricsin 記法

__C__や__Cb__，__Ccont__中に記述する C プログラム片からは，それが埋め込まれているローカル変数などのコンテキスト情報へ容易にアクセスすることができるように C レベル Ricsin 記法を設けた．

例えば，Ruby スクリプトでローカル変数 v が定義されていたとき，C プログラム片で変数 v への参照，代入を行うと，Ruby の変数 v への参照，代入となる（**図 8**）．C プログラム片では，そのための宣言は不要である．

ローカル変数以外の，Ruby スクリプトで参照できるグローバル変数，インスタンス変数，クラス変数（それぞれ，Ruby の文法で`$gv`，`@iv`，`@@cv` と表記）への参照と代入が可能である．

C プログラム片に，Ruby での表記（`$gv`，`@iv`，`@@cv`）を直接記述することで，その変数への参照を行うことができる（**図 9**）．C プログラム中で，文字列やコメント以外で`@` や`$`が現れることはないため，このような C レベル Ricsin 記法を定義することができる．

また，それぞれ`RGV_SET()`，`RIV_SET()`，`RCV_SET()`

```
v = 42
__C__ %q{
  printf("%d\n", FIX2INT(v));
                      #=> 42 を表示
  v = INT2FIX(43);    #   43 を代入
}
print v               #=> 43 を表示
```

図 8 変数アクセスの例
Fig. 8 Example of Variable Access

```
$gv = 42; @iv = 43; @@cv = 44
__C__ %q{
  printf("$gv: %d, @iv: %d, @@cv: %d",
    FIX2INT($gv), FIX2INT(@iv),
    FIX2INT(@@cv)); /* 42, 43, 44 と表示 */
}
```

図 9 各変数へのアクセスの例
Fig. 9 Example of Various types of Variable Access

というマクロを利用することで，変数への代入を実行することができる．例えば，Ruby のグローバル変数 `$gv` へ値 v を代入するときには `RGV_SET(gv, v)` と記述する．参照と比べ，若干面倒であるが，既存の Ruby C API にはこれらの変数を格納する箇所へのポインタを返す手段がないため，このような実装とした．ただし，今後利用していく上でこの点が問題となれば，Ruby C API を追加することで対処が可能である．

Ruby スクリプトで定義した定数は，`RConst(Name)` というマクロを利用して参照を行うことができる．この場合，`Name` という Ruby の定数にアクセスすることができる．

Ruby プログラムでは，`self` 特殊変数によって実行中のメソッドのレシーバを得ることができる．そこで，この `self` 特殊変数の内容を C プログラム片では `self` という変数へ参照することで得ることができるようにした．

このような変数アクセスを直接記述可能とする工夫によって，rcb ファイルに記述する Ruby スクリプトと C プログラムは値を柔軟にやりとりすることができる．

変数や定数への参照で得られる値，および代入可能な値は，すべて `VALUE` 型の値に限られる．そのため，例えば Fixnum（数値）オブジェクトであれば，C プログラムで利用可能な整数を取り出すために `FIX2INT()` 等の Ruby C API を利用する必要がある．

5. Ricsin の実装

本章では Ricsin 変換器が rcb ファイルを受け取った後，どのような変換を行うか，その具体的な内容と手法を述べる．

5.1 Ricsin での変換と実行の詳細

Ricsin 変換器は Ruby で実装した．また，rcb ファイルに記述されたプログラムの解析は CRuby のパーサ，コンパイラを利用するようにした．これにより，Ruby パーサの再実装を不要とした．

まず，Ricsin 変換器によって rcb ファイル☆を CRuby のバイトコード列にコンパイルする．このとき，*Ricsin Compile Mode* というコンパイルオプションを指定しておく．バイトコードコンパイラは，Ricsin Compile Mode のときは表 1 で示した Ricsin セレクタによるメソッド呼び出しを検知すると，通常のメソッド呼び出しバイトコード生成を行わずにプログラム片を集める．このとき，そのメソッド呼び出しの行番号と，引数として渡された文字列（C プログラム片）の組を保存する．Ricsin セレクタが `__Ccont__` のときは，そのバイトコードが位置するプログラムカウンタ（PC）も組にして保存する．

rcb ファイルの内容をバイトコードコンパイルが終了したとき，埋め込まれた C プログラム片の情報が Ricsin セレクタごとに得られる．Ricsin 変換器はこの情報をもとに C ソースファイルを生成する．C ソースファイルには，次節で述べる方式に従ってプログラム片ごとに C の関数（Ricsin 関数）を作り，その関数ポインタ一覧を収めた配列を作る．ただし，`__Cinit__` および `__Cdecl__` に示した C プログラム片はこの配列には含まれず，生成する C ソースファイルの該当箇所に直接埋め込まれる．なお，ここで生成したバイトコード列自体は現在は利用しない．

C ソースファイルを生成後，mkmf.rb ライブラリを利用して Makefile を生成する．これらをビルドすることで C 拡張ライブラリを生成する．また，Ricsin 変換器は C 拡張ライブラリの該当機能を呼ぶための Ruby スクリプトも生成する．

実行時は，コンパイルオプションに *Ricsin Exec mode* を指定して，再度 rcb ファイルの内容をバイトコードコンパイルする．このとき，Ricsin 関数ポインタの配列をあわせてコンパイルオプションとして渡しておく．バイトコードコンパイラは Ricsin セレクタによる呼び出しを，該当する関数ポインタを用いた関数呼び出しを行う特別な命令（`opt_ricsin_call` 命令）に置き換える．生成されたバイトコード列を実行することで，rcb ファイルの内容を正しく実行することができる．

☆ 正確には，この前に `#C` 行を `__Ccont__` に変換するプリプロセスを行う．

なお，rcb ファイルの内容を実行時に参照する必要があるため，C 拡張ライブラリ中にテキストデータとして rcb ファイルの内容を保持しておく．

5.2 C ソースファイルの生成

Ricsin Compile Mode でのバイトコードコンパイルで収集したプログラム片の情報を利用して，C 拡張ライブラリとしてコンパイルするための C ソースファイルを作成する．

C ソースファイルは次の要素を含む．

(1) Ruby C API を利用するためのヘッダファイル挿入指示
(2) Ricsin で利用するマクロ定義等
(3) `__Cdecl__`で指定された定義
(4) `__C__`等で指定されたプログラム片を変換した関数（Ricsin 関数）の定義
(5) Ricsin C 関数一覧を示す配列定義
(6) rcb ファイルの内容を示すテキストデータ
(7) C 拡張ライブラリロード時に行われる初期化処理

(1)〜(3) は，ただ挿入するだけである．(5) は生成した Ricsin 関数の関数ポインタを要素とする配列を用意する．(7) は，C 拡張ライブラリロード時に (6) の rcb ファイルの内容を Ricsin Exec Mode でバイトコードコンパイルする処理を記述する．また，`__Cinit__`で指定された初期化プログラム片を挿入する．

5.2.1 Ricsin C 関数への変換

図 10 に，`__C__`を用いた C 埋め込みプログラムをどのような Ricsin 関数に変換するのか述べる．

Ricsin 関数は `rb_control_frame_t` 型の引数 `cfp` をもつ，VALUE 型の値を返す関数として定義される．ボディ部にはプログラム片がそのまま埋め込まれる．`return` を指定しなければ `nil` を返すように，`Qnil` を返すプログラムを埋め込む．

`cfp` は，現在の Ruby のメソッドフレームへのポインタが格納されている．`cfp` を利用することで，ローカル変数へのアクセスや `self` の取得が可能となる．

埋め込まれるコンテキストで参照可能なローカル変数一覧を，マクロ定義により C プログラム片から参照可能にする．ローカル変数は `cfp->lfp[index]` として参照可能である．ローカル変数ごとに，`index` にあたる数値が異なるが，これはバイトコードコンパイル時に決定される☆．`cfp->lfp[index]` へ参照もしくは代入を行うことで，Ruby レベルのローカル変数に直接アクセスすることが可能である．

```
# rcb ファイル
v = 42
r = __C__(%q{
  /* 埋め込み C ボディ部 */
  rb_p(self); /* main と表示 */
  return INT2FIX(FIX2INT(v) + 1);
})
p r #=> 43 と表示

/* 生成される C ソースファイルの一部 */
#define v (cfp->lfp[3])
#define r (cfp->lfp[2])
VALUE ricsin_func_1(rb_control_frame_t *cfp)
{
  const VALUE self = cfp->self;
  {
    /* 埋め込み C ボディ部 */
    rb_p(self);
    return INT2FIX(FIX2INT(v) + 1);
  }
  return Qnil;
}
#undef v
#undef r
```

図 10 Ricsin プログラムの生成例
Fig. 10 Example of Ricsin Transform to C

マクロで定義するため，例えば定義されるローカル変数と同名の構造体メンバ変数があった場合，不正な置換が行われる．これは，現状の Ricsin の制限である．

C プログラム片に `self` への参照がある場合，`cfp->self` の値を const 変数 `self` として定義し，参照可能としておく．参照があるかどうかは C プログラム片に対して `self` という変数参照を正規表現によるパターンマッチで判定する．

なお，実際は変数名衝突を避けるため，`cfp` などの自動生成される変数名にはプレフィックスを適宜挿入する．

また，rcb ファイル上のプログラム片の行番号情報を利用して`#line` プラグマを挿入する．そのため，C プログラム片のデバッグは，既存のデバッガで通常の C ファイルに対して行うものとほぼ同様に行うことができる．

5.2.2 変数アクセスの変換

Ruby のローカル変数へのアクセスは前項で説明した．ここでは，それ以外のグローバル変数，インスタンス変数，クラス変数の変換について述べる．

C プログラム片の`$gv`，`@iv`，`@@cv` といった表記を，文字列のパターンマッチを利用してそれぞれ `RGV(gv)`，

☆ 厳密には，Ruby にはローカル変数にはメソッドローカル変数とブロックローカル変数があり，ここでは前者について説明している．後者へのアクセスにはさらに「どのブロックローカル変数であるか」，という情報が必要になるが，これもコンパイル時に決定可能であり，その情報を添えてマクロ定義を行う．

RIV(iv), RCV(cv) というマクロに変換する．マクロでは，それぞれ対応する Ruby C API に展開される．

これらの変数アクセスのためには，Ruby C API に ID をキーとして参照を行う必要がある．ID とは CRuby が管理する名前表のエントリであり，rb_intern() という Ruby C API に C 文字列を渡すことで，その C 文字列に対応した ID が得られる．

変数アクセスごとに，この rb_intern() による ID 問い合わせを行うのは無駄なので，出現するすべての変数名について C 拡張ライブラリロード時に ID 問い合わせを終了しておき，変数アクセス時には予め用意した ID を用いる．

5.3 専用バイトコードの挿入

Ricsin Exec Mode でバイトコードコンパイルすると，Ricsin セレクタでのメソッド呼び出しは，通常のメソッド呼び出し命令ではなく Ricsin 関数を呼び出すだけの opt_ricsin_call 命令を利用してコンパイルされる．

opt_ricsin_call 命令は，命令オペランドで呼び出す Ricsin 関数のポインタを指定する．関数ポインタを呼び出し，現在のメソッドのフレーム情報を引数に呼び出す．呼び出し後，関数からの返値をスタックにプッシュし，次の命令を実行する．

5.4 __Ccont__による Ricsin 関数の変換

__Ccont__では，スコープ中のすべての__Ccont__で指定したプログラム片を連結し，一つの Ricsin 関数として生成する．挟まれた Ruby スクリプト片を実行するときには，いったん Ricsin 関数から抜け，VM で Ruby スクリプト片を実行する．そして，Ricsin 関数を再開する．

この処理を実現するため，__Ccont__によって生成される Ricsin 関数に，VM のプログラムカウンタ（PC）をみて復帰処理を行うコードを先頭に挿入している．

図 6 をバイトコードコンパイルした結果が**図 11** である．このとき，8，18 番地に Ricsin 関数を呼ぶ opt_call_ricsin 命令が挿入される．

図 12 に生成された Ricsin 関数のソースコードを示す．最初に GET_PC() によって現在のプログラムカウンタを得て，その値に応じて復帰する場所へ goto でジャンプする．なお，opt_call_ricsin 命令は 8, 18 番地であるのに，分岐条件には 10, 20 となっているのは，命令開始前にプログラムカウンタの値を次の命令の位置に設定するためである．

Ruby プログラム片を実行するために Ricsin 関数を中断するときには，正しい Ruby プログラム片を実行するために，PC を SET_PC() によって次の命令の位置に設定し，return で処理を VM に戻す．

```
[ADDR] [INSN]           [OPERAND]
 0002  putobject        true
 0004  setlocal         v
 0008  opt_call_ricsin  <funcptr>
 0010  pop
 0013  putnil
 0014  setlocal         v
 0018  opt_call_ricsin  <funcptr>
 0020  leave
```

図 11 図 6 のバイトコードコンパイル結果
Fig. 11 Bytecode Sequence of Fig.6

```
#define v (cfp->lfp[2])
VALUE ricsin_func_1(rb_control_frame_t *cfp)
{
  switch (GET_PC()) {
    case 10: goto label_10;
    case 20: goto label_20;
  }
  {
  label_10:;
    while (v != Qnil) {;  /* (a) */
      SET_PC(10); return Qnil;
    label_20:;
      rb_p(v);            /* (c) */
    };                    /* (d) */
    SET_PC(25); return Qnil;
  }
}
  return Qnil;
}
#undef v
```

図 12 図 6 の変換結果
Fig. 12 Translation Result of Fig.6

このプログラムの実行の経過を示す．まず 8 番地で Ricsin 関数を呼び出され (a) が実行される．その後，中断し PC を 10 として Ruby スクリプト片の実行を行う．10 番地から 14 番地の Ruby スクリプト片が実行されると，18 番地にて Ricsin 関数が呼び出される．このとき，PC は 20 になっているので制御を (c) へ移す．(d) では繰り返しの終端なので，(a) に戻る．(a) の条件が偽となるため，繰り返しを抜け，Ricsin 関数から戻る．このとき，PC は 20 番地を指しており，leave 命令によりこの Ruby プログラムを終了する．

GET_PC()，SET_PC() は cfp に格納されている PC の情報へアクセスすることで実現している．

なお，#C 記法については，rcb ファイルをプリプロセスして，文字列置き換えを行い実現している．

5.5 CRuby への変更点

Ricsin 導入のために CRuby を変更した箇所は次の 2 点である．

(1) バイトコードコンパイルオプションに Ricsin mode を設け，Ricsin セレクタでのメソッド呼

表 2 評価環境
Table 2 Evaluation Environment

	評価環境 1	評価環境 2
CPU	Xeon CPU E5335 (Quad Core CPU) x2	UltraSPARC T2 64 Threads
メモリ	2GB	16GB
OS	Linux 2.6.18 x86_64	SunOS 5.10
コンパイラ	gcc 4.1.2	gcc 3.4.3

表 3 C の機能呼び出しの実行時間
Table 3 Execution Time of Calling C Function

	C (sec)	Ricsin (sec)	C/Ricsin
評価環境 1	0.44	0.05	8.8
評価環境 2	4.56	0.44	10.4

び出しをモードによって特殊な処理を行うように変更した．

(2) VM にオペランドとして指定された C の関数を呼ぶ `opt_ricsin_call` 命令を追加した．

この 2 点の変更は，Ricsin を利用しないときには既存の CRuby の動作に一切影響を与えない．

`opt_ricsin_call` 命令は VM に任意の命令を与える命令ととらえると，Ricsin は C プログラム片を新しい VM 命令としてコンパイルし，それを埋め込み実行するためのシステムということもできる．

6. 評　　価

本章では Ricsin の実行速度に着目した評価を行う．

評価には CRuby 1.9.0 revision 19508 をベースに，5.5 節で述べた変更を加えて Ricsin システムとしたものを利用した．評価環境は**表 2** に示す 2 種類を利用した．なお，今回の評価では並列計算を行わないため，コア数，スレッド数は無関係である．各評価は 10 回の試行のうち，もっとも速いものを計測結果とした．

6.1 C の機能呼び出し速度の比較

まず，従来の C メソッドと C の埋め込みの場合の比較評価を行う．C で実装した機能の呼び出しオーバヘッドを計測するため，空の C メソッド，空の C 埋め込みをそれぞれ 10^7 回繰り返し，その実行時間から繰り返しを行う処理時間を差し引いて呼び出しオーバヘッドを計測した．結果を**図 3** に示す．

結果を見ると，どちらの評価環境でも Ricsin の呼び出しが C のメソッド呼び出しよりも 10 倍程度高速であることがわかる．

6.2 イテレータ起動の高速化

Ricsin を利用してイテレータの起動を高速化することを試みる．Ruby では繰り返しを，イテレータメソッドにブロックを渡して行う．イテレータメソッド内で

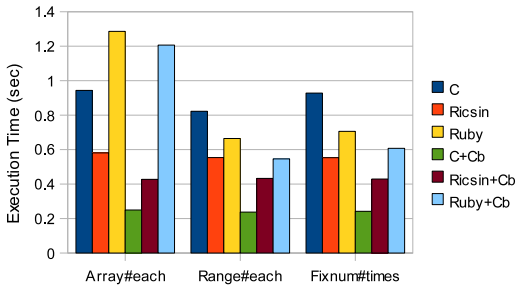

図 13 イテレータの高速化の評価（評価環境 1）
Fig. 13 Evaluation Result of Iterator (Eval. Env. 1)

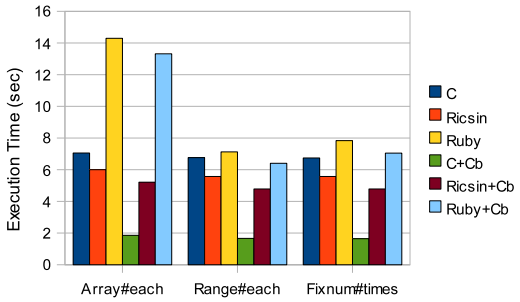

図 14 イテレータの高速化の評価（評価環境 2）
Fig. 14 Evaluation Result of Iterator (Eval. Env. 2)

必要回数だけブロックを起動することで，そのブロックを繰り返し実行する．`Array#each` などのイテレータは，C メソッドとして実装されているため，通常のブロックを渡すと，C から Ruby へ遷移するということになり，VM 起動オーバヘッドが必要になる．そこで，`__Ccont__`を利用して，この C から Ruby への遷移オーバヘッドをどれだけ削減可能かを評価した．また，`__Cb__` を利用して，C で実装したブロックを渡す記法がどれだけ性能に寄与するかを評価した．

評価は `Array#each`（配列の要素分だけ繰り返す），`Range#each`（範囲内の各要素について繰り返す），`Fixnum#times`（数値の回数だけ繰り返す）を従来の C メソッド，Ricsin で書き直したもの，Ruby だけで書いたものについて，それぞれ Ruby で空のブロックを渡した場合と，`__Cb__`を利用して C で記述した空のブロックを渡した場合について評価した．評価では 10^7 要素の配列，$1 \ldots 10^7$ の範囲オブジェクト，10^7 の数値を利用した．評価環境 1 での結果を**図 13**，評価環境 2 での結果を**図 14** に示す．

傾向としては，`Range#each` と `Fixnum#times` ではすでに C よりも Ruby で実装したバージョンのほうが高速である．これは，C から Ruby ブロックを呼び出すコストが高いことを示している．`Array#each` で

表 4 行列計算の実行時間
Table 4 Execution Time of Matrix Calculation

	Ruby (sec)	Ricsin (sec)	Ruby/Ricsin
評価環境 1	10.57	0.52	20.33
評価環境 2	85.31	6.73	12.68

Ruby 版が遅いのは，Ruby での配列アクセスが遅いからである．

どのイテレータでも，Ricsin で実装したものが C 版，Ruby 版と比べて高速である．これは，C から Ruby ブロックを呼び出す負荷が不要となった点で C 版よりも高速になり，繰り返しに関する処理を C 文の埋め込みにより記述したため Ruby 版よりも高速になったと考えられる．

`__Cb__`によって C で記述したブロックを渡すと，C 版のイテレータが最も高速に実行されている．これは，C から C の呼び出しとなることで言語間を遷移するオーバヘッドが削減されたからである．Ricsin および Ruby 版でも C で記述したブロック呼び出しのほうが高速であるが，C 版ほどの性能改善はない．

このことから，Ruby のブロックを渡す一般的なイテレータは Ricsin で，さらに高速化が必要な場合は C で記述したイテレータに C で記述したブロックを渡すという実装が適していることがわかる．

6.3 プログラムの高速化

Ricsin を利用して添付ライブラリである matrix.rb で定義してある行列積を求める計算の高速化を行った．書き換えにおいては，Matrix 同士の乗算の処理のみに着目して `Matrix#*`の定義の一部を書き換えた．また，行列の要素が整数もしくは浮動小数点数だった場合にとくに高速になるように C プログラム片を記述した．300 × 300 の整数を要素とする行列についての評価結果を**表 4** に示す．

評価では整数を要素とする行列乗算を行ったため，12〜20 倍の高速化となり，C での最適化の効果が強く出た．このように，特定の場合だけ高速にするような処理を記述したい場合，従来はすべて C 言語で記述する必要があった．しかし，Ricsin を用いることで一部の書き換えが容易に行えることが確認できた．

置き換えにあたっては 12 行の Ruby スクリプトに対し 36 行の C プログラム片を埋め込んだ．処理が必要な箇所に直接埋め込むことができたので，プログラムの意図は取りやすいものとなった．また，C ソースファイルを分けるような煩雑な手間が不要となり，効率的な開発を行うことができた．

7. 議　　論

本章では，Ricsin について議論を行う．

まず，言語処理系の対象言語（この場合 Ruby）以外の言語（この場合 C）でプログラムを記述して高速化を目指す方式に関して議論する．一般的に，対象言語をその他の言語で拡張すると，性能向上の制限となることがある．対象言語の高機能な解析やそれを利用した高速化が進むと，その他の言語で記述した部分が制約やボトルネックになる可能性がある．これを解決するには言語間解析が必要になり，これは一般に困難である．そのため，従来の言語処理系開発では対象言語でプログラムを記述する，という方針がとられることが多い．

しかし，Ruby と C という関係のみに着目すると，Ruby のような解析・最適化が困難な言語に対して，C プログラム片の性能がボトルネックになるような最適化器が登場するにはまだしばらくかかるだろうと予測している☆．また，高性能な言語処理系は OS や CPU などの環境に依存した最適化が必要になるが，CRuby のように様々な環境に対応した言語処理系ではすべての環境に対応させるのは困難である．そこで，対応環境の豊富な C 言語を用いることで，多くの環境で高速化を行うことができる．以上から，Ricsin のような手軽な仕組みで C による高速化を図る手段は，まだ現実的で有効であると考えられる．

また，性能に関して，Ruby から C，C から Ruby のような言語間の遷移がオーバヘッドとなっていたが，Ricsin ではその負荷をなるべく排除するように設計している．ただし，GC や並列実行については，ライトバリアや同期に関して C での記述が問題となってくる可能性はある．この点については CRuby の問題点として，今後も検討していく必要がある．

次に，開発環境に関する議論を行う．Ricsin の開発では，Ruby スクリプトに C プログラム片を埋め込むというスタイルから，それぞれの言語環境で得られる開発環境の支援が受けられないおそれがある．しかし，基本的に rcb ファイルは Ruby の文法となるように設計したため，Ruby 用開発環境をそのまま利用することができる．また，C プログラム片は必要な箇所に埋め込むという機能から，長い C プログラムを作成することはないと想定している．もし，長い C プログラムを埋め込む場合には，別途 C ソースファイルを

☆ Ruby は業ではなく，オープンソースソフトウェアでの有志による開発体制であることからも，高性能だが複雑な処理系がすぐにリリースされるとは考えづらい．

用意するのが自然である．Ricsin は mkmf.rb ライブラリを利用しているため，別途 C ソースファイルを用意しての分割コンパイルもサポートしている．

プログラムを自動生成する場合，デバッグが困難になる可能性があるが，生成された C ソースファイルには`#line`指示子によるファイル名，行番号指定を行うので，既存の C プログラム用デバッガを利用して，これまでの C 拡張ライブラリと同様の方式でデバッグなどを行うことができる．

ただし，開発環境においては，今後複数言語混在に対応するとより便利になると思われる．また，同様に Ruby の環境，C の環境のように，複数の言語環境へ同時にアクセスできるデバッガがあれば，デバッグ効率の向上が実現できると思われる．

8. 関 連 研 究

Ruby には他言語呼び出しインターフェースとして DL ライブラリや Win32OLE ライブラリが存在する．それぞれ，Ruby のメソッド呼び出しを C の関数呼び出しや OLE ライブラリ機能の呼び出しに変換する．Ricsin では，メソッド呼び出しを介さずに C の関数を直接呼び出すため，高速な機能の利用が可能である．

Ruby 以外の言語でも，プログラム中に他の言語を埋め込む提案や実際に利用されている例がある．

多くの C コンパイラは`asm`文，もしくは`__asm__`文といった機能でアセンブラを埋め込むことが可能である．GCC[4] では，この構文を拡張し，アセンブラに C 言語の値のやりとりが可能となっている．Ricsin の初期の目標はこの asm 文であったが，Ruby のコンテキスト情報へ直接アクセスするための C レベル Ricsin 記法により，より容易に言語間のやりとりを可能とした．同様の例として，Java に Java VM のバイトコードを埋め込む研究がある[8]．

Jeannie[7] という，Java プログラム中に C プログラム片を混在させるシステムが提案されている．Jeannie では，Java の文法を拡張し，特殊な記法を用いて C プログラムを Java プログラムに埋め込むことを可能としている．C プログラム片は JNI による拡張機能に変換される．しかし，Java の文法に特殊な記法を導入したことで，Java の開発環境による支援を受けることを困難にしている点，JNI のコードにするため Java-C 間の遷移のオーバヘッドが発生する問題がある．Ricsin では，基本的に専用文法を用意するのではなく，既存文法を別の意味で解釈するという拡張を行ったため，開発環境に関する問題は少ない．また，VM を拡張することで高速な C の呼び出し，もしくは C から Ruby への遷移を可能としている点で Ricsin が有利である．

Jeannie では，変換時に Java と C の型の解析を行い，C で Java の型を扱うためのサポートを行っている．Ricsin でも C プログラムの変数の型情報を利用して，Ruby と C の型情報を自動的に変換する手法もあるが，Ruby ではプリミティブ型が存在せず，C の型へ自明な変換をもつオブジェクトばかりではない点や，C 拡張ライブラリと同程度の表現力を持たせるという目的は達しているため，Ricsin では明示的な変換を必須としている．ただし，C の型情報を利用したより容易な記述方式も，今後検討していきたいと考えている．ただし，C の文法解析を行うためには C プリプロセッサや C 構文解析器を自前で用意，もしくは外部ツールの利用をする必要があり，Ricsin のツールとしての手軽さ，可搬性が損なわれるおそれがある．現在の Ricsin は Ruby で記述した単純な変換器と，C 拡張ライブラリをビルドする環境があればよいため，導入がきわめて手軽であり，保守も容易である．現実的なツールを提供するためには，このトレードオフは十分考慮する必要がある．

HTML などのコンテンツ記述言語に JavaScript[2] などのプログラミング言語を埋め込むような例もあるが，これは静的なコンテンツと動的な言語の混在という意味で本研究の対象とは異なる．また，一つの言語処理系を拡張し，複数の言語を処理するという方針もある[6),10)] が，本研究では既存の言語処理系を活用する方針をとっている．

9. ま と め

本稿では Ruby スクリプトに C プログラム片を混在させるためのシステムである Ricsin を提案した．Ricsin を用いることで，C で記述する必要がある箇所に直接 C の文の記述を可能とし，また Ruby のコンテキスト情報へアクセスする記法を用意することで，従来の C 拡張ライブラリ開発の効率やプログラムの見通しの悪さを改善した．また，VM に変更を加えることで，C の機能を呼び出す際のメソッド呼び出しのオーバヘッドを排除した．評価の結果，C メソッドの呼び出しに比べ 10 倍程度高速に C で記述した機能を呼び出すことができることを確認した．

今後の課題として，C プログラム片の解析により，型情報を利用した型変換の挿入や`__Ccont__`での C の変数の保存に関するの検討が必要である．また，Ruby プログラムを C 言語へ変換する AOT コンパイラ[16] と連携することで，さらなる高速化を目指したい．

謝　　辞

本稿執筆にあたり，Ruby 開発者の方々，筆者の所属する東京大学情報理工学系研究科創造情報学専攻の方々，筑波大学の水島宏太氏にご意見を頂きました．協力してくださった方々にこの場を借りて感謝いたします．

本研究の一部は，日本学術振興会科学研究費補助金若手研究（スタートアップ），課題番号 19800007 の助成を得て行いました．

参 考 文 献

1) Beazley, D. M.: SWIG: an easy to use tool for integrating scripting languages with C and C++, *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, Berkeley, CA, USA, USENIX Association, pp. 15–15 (1996).
2) Brent W. Benson, J.: JavaScript, *SIGPLAN Not.*, Vol. 34, No. 4, pp. 25–27 (1999).
3) Davis, R. and Hodel, E.: RubyForge: RubyInline. http://rubyforge.org/projects/rubyinline/.
4) Free Software Foundation (FSF): GCC Home Page. http://gcc.gnu.org/.
5) Gordon, R., 林秀幸 (訳): JNI:Java Native Interface プログラミング, ソフトバンククリエイティブ (1998).
6) Grimm, R.: Better extensibility through modular syntax, *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, ACM, pp. 38–51 (2006).
7) Hirzel, M. and Grimm, R.: Jeannie: granting java native interface developers their wishes, *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, New York, NY, USA, ACM, pp. 19–38 (2007).
8) Kats, L. C. L., Bravenboer, M. and Visser, E.: Mixing Source and Bytecode. A Case for Compilation by Normalization, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)* (Kiczales, G.(ed.)), Nashville, Tenessee, USA, ACM Press (2008).
9) Lam, J.: RubyForge: IronRuby: Project Info. http://rubyforge.org/projects/ironruby.
10) Meijer, E., Beckman, B. and Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework, *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, New York, NY, USA, ACM, pp. 706–706 (2006).
11) Nutter, C. O. and Enebo, T. E.: JRuby Java powered Ruby implementation. http://jruby.codehaus.org/.
12) Phoenix, E.: The Rubinius Project. http://rubini.us/.
13) Thomas, D., Fowler, C. and Hunt, A.: *Programming Ruby*, The Pragmatic Programmers (2004).
14) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, 株式会社アスキー (1999).
15) まつもとゆきひろ他: オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/ja/.
16) 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 第 70 回 プログラミング研究会 (2008).
17) 松本行弘: Ruby の真実, 情報処理, Vol. 44, No. 5, pp. 515–521 (2003).
18) 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 (PRO), Vol. 47, No. SIG 2(PRO28), pp. 57–73 (2006).
19) 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 における並列実行スレッドの実装, 情報処理学会論文誌 (PRO), Vol. 48, No. SIG 10(PRO33), pp. 1–16 (2007).