Scheme on TRV

大浦裕太
Yuta Oura
熊本大学工学部
c3412@st.cs.kumamoto-u.ac.jp

黒田修平
Shuhei Kuroda
熊本大学大学院自然科学研究科
kuroda@st.cs.kumamoto-u.ac.jp

木山真人
Masato Kiyama
熊本大学大学院自然科学研究科
masato@cs.kumamoto-u.ac.jp

芦原評
Hyo Ashihara
熊本大学大学院自然科学研究科
ashihara@cs.kumamoto-u.ac.jp

本論文では，Ruby で開発されるアプリケーションに組込んで使用することを主要目的として設計・実装した Scheme コンパイラについて述べる．本処理系は，Scheme プログラムから Ruby の仮想マシンである TRV(The Ruby Virtual Machine) 用の命令列を生成する．以前，我々は Ruby で開発されるアプリケーションに組込んで使用することを目的とした Scheme ドライバ Cherubim を開発した．Cherubim は，(1)Scheme 処理系の実装ノウハウを持たない Ruby プログラマにも機能の追加・削除・変更等が容易，(2) コンパクトな実装，(3) デバックのために最低限必要な機能は備える，などを目的として設計している．しかし，Cherubim には実行速度が遅いという問題がある．一方 Ruby の開発者は，TRV の開発により実行速度の向上を目指している．そこで我々は，TRV 用の命令列を生成するコンパイラを開発することで，Ruby アプリケーション組込み用の Scheme ドライバの実行速度の向上を目指す．実行速度向上を目的とした TRV 用の命令列を生成するコンパイラの実装・評価を行った．評価の結果，Cherubim の設計目標の「Scheme 処理系の実装ノウハウを持たない Ruby プログラマにも機能の追加・削除・変更等が容易」を実現したまま実行速度の高速化を行えた．

In this paper, we preaent a Scheme Compiler which is desigend to be used primarily as an embedded system in Ruby apprication. This processing system generates a sequence of instruction for TRV (The Ruby Virtual Machine) which is a virtual machine of Ruby from a Scheme program.Before, we devleloped Scheme driver Cherubim which is desigend to be used primarily as an embedded system in Ruby apprication. The key design issues include (1)it should be easy to extend, modify and delete the functionality even for a Ruby programmer who is not familiar with Scheme implementation, (2)the driver should be compact enough, and (3)the driver should have feature for debug. However,there is the problem that an execution speed is slow in Cherubim. On the other hand, the developer of Ruby aims at the improvement of the execution speed by development of TRV. Therefore we evaluated that we implemented the compiler which generated a sequence of instruction for TRV aimed for execution speed improvement. As result of evaluation, we speeded up the execution speed with having realized “it should be easy to extend, modify and delete the functionality even for a Ruby programmer who is not familiar with Scheme implementation” which was design target of Cherubim.

1 はじめに

以前我々は，Ruby で開発されるアプリケーションに組み込んで使用することを主要目的として Scheme ドライバ Cherubim[10] を開発した．Cherubim は以下の項目を特に重視して設計してある．

- Scheme 処理系の実装ノウハウをもたない Ruby プログラマにも機能の追加・削除・変更が容易
- Ruby で開発したソフトウェア部品を扱うための機能を容易に組み込める
- コンパクトな実装
- 高度な Scheme プログラム開発支援ツールを備える必要はないが，デバッグのために最低限必要な機能は備える

しかし，Cherubim は以下の 2 点が原因で実行速度が遅くなってしまう．

- 可読性・拡張性を優先した設計
- Ruby1.8(Ruby の安定版処理系) の実行速度

Cherubim は可読性・拡張性を重視して設計したインタプリタである．可読性・拡張性と実行速度がトレードオフの関係になったときには，実行速度は犠牲にし，可読性・拡張性を優先して実装してある．また，Ruby1.8 は Ruby プログラムをパースした結果生成される構文木を辿りながら実行するので，実行速度が遅い．そのため，Cherubim を Ruby1.8 上で動作させると，実行速度がどうしても遅くなってしまう．

Ruby1.8 の実行速度の問題を解決するために，近年 Ruby1.9 の開発が進んでいる．Ruby1.9 では，構文木を辿りながら仮想マシン TRV(The Ruby Virtual Machine) の命令列を生成し，その命令列を TRV によって解釈実行する．これにより Ruby1.9 では，Ruby プログラムを高速実行を目指している．

そこで本研究では，Ruby アプリケーション組み込み用の Scheme ドライバの実行速度向上を得るために，Scheme プログラムから TRV の命令列を生成するコンパイラを開発する．このコンパイラは，インタプリタである Cherubim と比べて多少の可読性・拡張性の低下はあるものの，大幅な速度向上が見込まれる．

以下，2 章では TRV について述べ，3 章では本処理系の設計について，4 章では本処理系の実装についてそれぞれ述べる．さらに 5 章でベンチマークによる評価について述べ，6 章でまとめる．

2 TRV

プログラミング言語 Ruby の処理系である Ruby には，現在安定版の Ruby1.8 と開発版の Ruby1.9 が存在する．前述したように，Ruby1.9 では処理系内部の仮想マシン TRV によって Ruby プログラムの高速実行を目指している．本章では，Ruby1.9 の仮想マシン TRV について述べる．

TRV[6] は，Ruby プログラムを高速に実行することを目的とした仮想マシンであり，Ruby プログラムを命令列へコンパイルし，命令列を実行する．TRV は C 言語で実装してあり，実行部分はシンプルなスタックマシンとなっている．これにより Ruby1.9 では，Ruby プログラムを高速に実行する．Ruby1.9 には，VM::InstructionSequence モジュールがある．このモジュールは，Ruby プログラムから TRV 用の命令列を生成したり，TRV 用の命令列を読み込んで実行を行う．本処理系ではこのモジュールを用いて実装を行う．

2.1 命令セット

TRV では，Ruby プログラムを正しく表現するための基本命令セットが定義してある．表 1 に基本命令セットのカテゴリ一覧を示す．

表 1. 基本命令セットのカテゴリ一覧

変数系	ローカル変数などの値を取得・設定
値系	self の値や文字列・配列などを生成
スタック操作	スタック上の値操作
メソッド定義	メソッドを定義
クラス定義系	クラス・モジュールを定義
メソッド呼び出し系	メソッド呼び出しや yield など
例外系	例外を実装するために利用
ジャンプ系	ジャンプ・条件分岐
最適化系	最適化のための命令

Ruby にはプリミティブ型がないので，数値の演算命令などは用意されていない．また，メソッド定義や，ク

ラス定義は動的に行う必要があるので命令として用意されている．

2.2 最適化

TRV では，Ruby プログラムを高速に実行するためにいくつかの最適化が行われている．以下に TRV で行われている最適化を示す．

- 命令ディスパッチ
- インラインキャッシュ
- 特化命令
- オペランドの融合と命令融合
- スタックキャッシング

3 設計

本章では，本処理系の設計について述べる．

本処理系では，以下の 3 つを設計目標として実装を行う．

(1) Scheme 処理系の実装ノウハウを持たない Ruby プログラマにも機能の追加・削除・変更が容易

(2) Ruby で開発したソフトウェア部品を扱うための機能を容易に組み込める

(3) 高性能である必要はないが，性能が極端に悪くない

(1) を達成するために，処理系記述言語は Ruby プログラマが使える Ruby にする．さらに，Scheme の組み込み手続きを Ruby で容易に定義できれば，Ruby の部品を利用するためのインターフェースは容易に定義できるので (2) を満たせる．また，Ruby の豊富なクラスライブラリを利用すれば，開発コストの削減ができる．

しかし，処理系のさまざまな実装上の知識を必要とするために，Ruby で記述すれば (1) を達成できるとは限らない．そこで本処理系では，実装上の知識をほとんど必要としないための工夫を行っている．この点については，4.4 節で詳しく述べる．

前述したように，容易に手続きができるような枠組みがあれば，Ruby で記述された部品を呼び出すための組み込み手続きが容易に定義できる．また一般には，Scheme からの Ruby の呼び出しのみならず，逆に Ruby からの Scheme の呼び出しも望ましい．Ruby のクラスのサブクラスを Scheme で定義したいような場合である．本処理系では，この機能を実現するために，新たにインターフェース用にクラスを作り対応している．この点については，4.8 節で詳しく述べる．

本処理系では，インタプリタ方式ではなくコンパイラ方式を採用した．我々が以前開発した Cherubim は実行速度が遅いが，この問題の原因のひとつには Cherubim がインタプリタ方式を採用した点にある．そこで本処理系では，TRV の命令列を生成するコンパイラ方式の採用により (3) の実現をはかる．

4 実装

本章では，本処理系の実装ついて述べる．まず 4.1 節では本処理系で採用した言語仕様を述べ，4.2 節では Scheme データを表現するために，Ruby でどのように実装しているかついて述べる．4.3 節で構文，4.4 節で手続きのそれぞれの処理について述べ，4.5 節で継続，4.6 節では末尾再帰の最適化の実装についてそれぞれ述べる．4.7 節では，本処理系での手続き呼び出しについて述べ，4.8 節では本処理系の利用法について述べる．

4.1 言語仕様

本処理系は，言語仕様として Scheme[1] のほぼフルセットを採用している．これは，Scheme の仕様書である R5RS[1](Revised 5 Report onthe Alogrithmic Language Scheme) のほぼフルセットに準拠していることを意味する．R5RS に準拠していない点は，次の点である．

- 継続 (continuation) は，それを生成した call/cc 手続きがリターンした後には，呼び出せない．つまり，escape procedure として機能し，大域脱出には利用できるが，コルーチンは再現できない．

ただし本処理系は現在実装途中であるため，Scheme の処理系として持つべき機能を未だに実装していない部分がある．現段階では，全 24 構文・ライブラリ構文のうち 18，全 217 手続き・ライブラリ手続き・省略可能手続きのうち 145 を実装している．

4.2 データの種類

Scheme データを表現するために使用する Ruby のクラスを図 1 に示す.

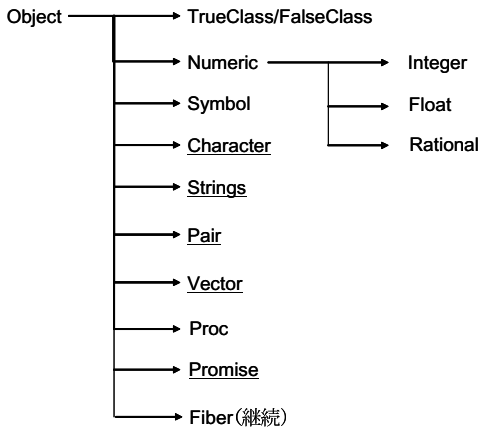

図 1. Scheme データを表現する Ruby クラス

矢印はクラス階層を表している．始点に位置するクラスが，終点に位置するクラスのスーパークラスであることを意味している．下線を引いたクラスは，処理系実装のために定義したクラスである．他のクラスは Ruby の標準的なクラスである．

本処理系では，Scheme データの種類をできる限り Ruby の持つ型に置き換えて実装している．数に関するデータは，Ruby の Integer，Float，Rational クラスで表現し，真 (#t)，偽 (#f) のブーリアン型も Ruby の TrueClass，FalseClasss をそのまま用いる．

Scheme の完全な継続は実装できないが，Fiber オブジェクトを継続オブジェクトとし実装することで継続の実装を行う．継続の実装については 4.6 節で詳しく説明する．

Ruby の持つ型に置き換えられない Scheme データに対しては新しくクラスを作って対応している．例えば，コンス・ペアは Pair クラスを作って対応している．また，空オブジェクトを表す Null クラス，未定義値を表す Undfined クラス，未規定を表す Unspecified クラスを定義している．

4.3 構文

本処理系では，まず Scheme プログラムをパーサによって構文解析し構文木を生成する．次に生成した構文木を visitor パターンを使って辿りながらコンパイルを行う．

以下に，条件分岐を行う構文 if 式のコンパイラの実装例を示す.

```
def visit_IfNode (node, tailflag = false)
  l_else, l_end = @env.getlabel, @env.getlabel
  node.cond.accept(self)
  @body << [:branchunless, l_else]
  node.then.accept(self, tailflag)
  @body << [:jump, l_end]
  @body << l_else
  if node.else != nil
   node.else.accept(self,tailflag)
  else
   @body<<[:putnil]<<[:getconstant,:Unspecified]
  end
  @body << l_end
end
```

このメソッド定義は，if 式

```
(if <cond> <then> <else>)
```

のコンパイル手続きを，その仕様に従って記述したものである．

visit_IfNode メソッドの引数は，パースによって生成された構文木のノードを表す node と，末尾再帰最適化に使用する tailflag から成る．末尾再帰最適化の内容については 4.5 節で説明する．まず，条件式 cond 部をコンパイルするために再帰的に accept メソッドを呼び出し，branchunless 命令を命令列へ挿入する．branchunless 命令は，cond 部の実行結果が真であれば l_else ラベルまでジャンプする．次に，cond 部の実行結果が真の場合に実行される命令列を then 部のコンパイルによって生成し，jump 命令とラベルを命令列へ挿入する．最後に，cond 部の実行結果が偽の場合に実行される else 部のコンパイルを行い，ラベルを命令列へ挿入し if 式のコンパイルを終了する．else 部は省略可能であり，省略した場合は未規定 (Unspecified) を返す.

4.4 手続き

手続きは，実行時に手続きの内容を Ruby で記述してある procedure.rb を読み込んで実行する．

手続きを Ruby で記述し実行時に呼び出すことにより，本処理系の設計目標である「Scheme 処理系の実装のノウハウをもたない Ruby プログラマにも機能の追加・削除・変更が容易」を達成する．

手続きの記述例をあげると，Pair クラスで定義されているリストを受け取り，その先頭の値を返すという簡単な手続きである car は，本処理系では次のように実装されている．

```
$top[:car] = proc {|args|
  args.first.car
}
```

実装の詳細とは無関係に car という関数の機能を，忠実に Ruby プログラムに置き換えたプログラムである．グローバル変数$top は，Scheme プログラムを評価するためのトップレベルの環境を格納する．上記のプログラムのように手続きは全て Ruby の Proc オブジェクトとして生成する．

本処理系では，構文の部分だけコンパイルを行い，手続きは実行時に Ruby で記述されたプログラムを呼び出す方法をとる．

4.5 継続

継続は，手続き call/cc(call-with-current-continuation) によって行われる．call/cc は 1 引数の関数を受け取り，生成した継続を引数として呼び出す．

本処理系では，Scheme の継続を完全に実現することは断念し，escape procedure としての機能を実現する．これは，TRV がスタック上に特殊なフレームをワインドする dynamic-wind をサポートしていないからである．

```
(call/cc f)
```

によって生成された継続は，関数 f の実行中に限り呼び出すことができる．継続が呼び出されると，f の実行は直ちに終了し，継続への引数が call/cc の値として返される．f の実行中に継続が呼び出されないで，f が正常にリターンしたときは，f の返す値が call/cc の値となる．いずれの場合も，call/cc がリターンした後では，継続を呼び出せない．

以上の条件のもと本処理系では，Ruby1.9 から導入された Fiber を用いて実装を行う．Fiber はセミコルーチンとして動作する．Fiber オブジェクトは resume で処理を開始し，yield で処理を中断する．

本処理系では，f の引数の継続オブジェクトを

```
proc {|x| Fiber.yield(x.first).call}
```

としている．

call/cc 手続き全体の実装は以下のようになる．

```
$top[:'call/cc'] = proc {|args|
  f = args.first
  Fiber.new{
    f.call(proc{|x| Fiber.yield(x.first).call})
  }.resume
}
```

上記の継続オブジェクトを，引数に f を呼び出して call/cc を escape procedure として実現する．

4.6 末尾再帰の最適化

関数型言語のコンパイルには継続渡し方式 (CPS) と呼ばれる中間コードを用いられる．その方式を用いた出力コードを実行するためには，関数を呼び出す際に引数とともに次に実行してほしい継続を渡す．関数から復帰する場合も実は呼び出す場合と同様であり，継続へジャンプすればよい．この方式の場合，関数呼び出しとは引数を持った goto でありフレームを消費しないので，深い再帰を行ってもスタックオーバーフローを起こさない．

```
(define (fact n a)
  (if (= n 1)
      a
      (fact (- n 1) (* n a)))
```

例えば，上記のような Scheme プログラムで末尾再帰の最適化を行わない場合は，fact 手続きが再帰的に呼び出されていき，スタックを消費する．しかし，末尾再帰の最適化を行うと fact 手続きが再帰的には呼び出されずに反復処理になりスタックを消費しないですむ．

本処理系では，コンパイル時に末尾式かどうかを判断して最適化を行う．末尾式に再帰呼び出しがある場合は，通常の手続き呼び出しを行うのではなく，jump 命令による反復処理に変換する．

以下に，本処理系での末尾再帰の最適化を if 式を例に示す．

```
(if <式> <末尾式> <末尾式>)
```

上記のように if 式の場合，then 部と else 部が末尾式となる．末尾式は，以下のように tailflag フラグを渡すしながら末尾式かどうかを判断しながらコンパイルを行う．

```
node.then.accept(self, tailflag)
```

末尾式が再帰呼び出しの場合，

```
label:
  命令列
  jump label
```

のように手続きの再帰呼び出しが，jump 命令による反復処理に変換される．

ただし，本処理系は現在実装中のため自己末尾再帰の場合にしか末尾再帰の最適化は行わない．

4.7 手続き呼び出し

本処理系では，すでに組み込まれている Scheme の関数を評価する method メソッドと Scheme プログラム自体を評価する eval メソッドの 2 つのメソッドを実装している．

以下に手続きを呼び出すメソッドである method メソッドの実装を示す．

```
def method (name, *args)
  $top[name].call(args)
end
```

このメソッドは引数に手続き名の name とその手続きの引数となる値の 2 つを受け取る．メソッドの動きとして，call メソッドによって環境に格納されている手続きを呼び出し，値を評価して返す．

Scheme プログラム自体を評価する eval メソッドは次のように実装されている．

```
def eval (str)
  @parser.parse(str)
  c = Compiler.toplevel(@parser.tree)
  c.compile
  VM::InstructionSequence.load(c.to_iseq).eval
end
```

このメソッドは引数に Scheme プログラムをとる．Parser クラスのインスタンス@parser によって Scheme プログラムをから構文木を作り，Compier クラスによってコンパイルを行い TRV 用の命令列を生成する．生成された命令列は VM::InstructionSequence モジュールによって実行される．

4.8 利用法

本処理系は，入力された Scheme プログラムを TRV の命令列にコンパイルし実行する．出力された命令列は Ruby1.9.0 の VM::InstructionSequence モジュールを利用して実行を行う．

Ruby アプリケーションに組み込んで利用する場合の利用法を例に挙げる．Ruby アプリケーション内で本処理系を関数として利用する場合，インターフェイス用クラスである Scheme クラスの 2 つのメソッド (method, eval) を利用して実行できる．以下に実行例を示す．

```
scheme = Scheme.new
scheme.eval('(define (fact n) (if (= n 1) 1
                (* n (fact (- n 1)))))')
scheme.method(:fact, 6)
```

new メソッドで新たに Scheme クラスのインスタンスを作り，変数 scheme に代入する．eval メソッドの引数に，階乗計算を行う関数を定義した Scheme プログラムを渡す．これによって，階乗計算を行う関数 fact が定義される．次の行で，method メソッドに関数名の fact と引数 6 を渡すことで，関数定義した関数 fact が呼び出され，6 の階乗を計算した値 720 が Ruby オブジェクトとして返る．

5 評価

本章では，ベンチマークを用いた評価について述べる．

今回は，Lisp 処理系のベンチマークとして有名な Gabriel のベンチマークを用いた．評価環境を表 2 に示す．

比較対象として，Ruby1.9.0 での本処理系，Ruby1.9.0 での Cherubim, Ruby1.8.6 での Cherubim，Java1.6.0 での JScheme，Java1.6.0 での JAKLD(Java Application Kumikomi Lisp Driver)[7] の 5 つを用いた．

表 2. 評価環境

CPU	:	Pentium4 3.2GHz
Memory	:	2GB
OS	:	Linux kernel 2.6.16
Ruby	:	1.9.0, 1.8.6
Java	:	1.6.0

表 3. ベンチマークの実行結果 (単位は秒)

テスト	本処理系	Cherubim(1.9)	Cherubim(1.8)	JScheme	JAKLD
tak	5.270	127.833	1750.67	0.924	3.831
takr	6.688	138.926	3508.588	0.821	4.288
takl	3.512	226.351	3157.051	0.775	3.954
deriv	2.592	77.676	3214.162	0.254	1.164
div-iter	1.679	43.754	108.351	0.255	0.944
div-rec	1.205	39.511	2464.103	0.226	1.029

JScheme[4] は，Java で実装されており Scheme プログラムを独自の内部コードに変換した後，内部コードのインタプリタによって実行する．JAKLD は，Java で実装された Scheme インタプリタである．評価の結果を表 3 に示す．

Cherubim と本処理系の実行速度の比較を行うと，本処理系は平均して 33 倍の速度向上が見られた．これは Cherubim のインタプリタ方式からコンパイラ方式への変更の相違と考えられる．

Cherubim の実行環境を，Ruby1.8 から Ruby1.9 へ変更しただけでも 27 倍の速度向上が見られた．

一方で，JScheme と比較すると本処理系は平均して約 6.8 倍の時間を要している．また JAKLD と比較すると，約 1.5 倍の時間を要している．これらは，本処理系の設計や Ruby と Java の実行速度の差だと考えられる．当初の目標である「高性能である必要はないが，極端に性能が悪くないこと」は達成できたと考えられる．

6 まとめ

本論文では，Ruby アプリケーションに組み込んで利用可能な Scheme ドライバの実行速度の高速化について述べてきた．実行速度の高速化の実現方法として，以前当研究室で開発された Cherubim のインタプリタ方式から，TRV の命令列を生成するコンパイラ方式へ変更することにより実行速度高速化を行った．評価の結果，設計目標を達成し実行速度の向上を行えた．

今後の課題として，処理系が言語仕様としてる Scheme のほぼフルセットのサポートがあげられる．

参考文献

[1] Richard Kelsey, William Clinger, and Jonathan Rees, “Revised 5 Report on the Algorithmic Language Scheme”(1998)

[2] IEEE: “IEEE Standard for the Scheme Programming Languaage”(IEEE p1178)(1991)

[3] Richard P.Gabriel, “Performance and Evaluation of Lisp Systems”(1985)

[4] JScheme http://jscheme.sourceforge.net/jscheme

[5] オブジェクト指向スクリプト言語 Ruby http://www.ruby-lang.org/jp

[6] YARV アーキテクチャ http://www.atdot.net/yarv

[7] 湯浅太一，“Java アプリケーション組み込み用の Lisp ドライバ”(2003)

[8] 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”

[9] 鵜川始陽，湯浅太一，小宮常康，八杉昌宏 “Javaと相互呼び出し可能な Scheme 処理系「ぶぶ」における継続機能と例外処理機能の実装” (2001)

[10] 黒田修平，木山真人， “Ruby アプリケーション組み込み用 Scheme ドライバの設計と実装”(2006)