青山学院大学理工学部情報テクノロジー学科
2016 (平成 28) 年度 卒業論文要旨

Ruby VM におけるスタック領域の拡張の提案と実装

小池 翔 (15813031)
Dürst 研究室

1 研究背景

オブジェクト指向スクリプト言語 Ruby は, 直感的な文法と豊かな表現力を備えた動的言語である. Web アプリケーションフレームワーク Ruby on Rails の普及で, Web アプリケーション開発言語としても様々な環境下で利用されることが増えている. しかし, 普及に伴いマルチスレッドなど, より多くのプログラムを処理する必要が出てくる.

Ruby の VM である YARV では, スレッド一つにつきスタックを用意する. 再帰の深いプログラムでは, スタックを使い切ってしまいスタックオーバーフローを起こす. 省メモリの目的でスタック一つあたりは小さいことが望ましいが, 十分な再帰もできるようにするべきである.

よって本研究では, スタック一つあたりを小さくし, 必要なときのみスタック領域を大きくすることで, 従来よりも省メモリに, かつ十分な再帰の深さを処理する拡張を提案し実装する.

2 YARV

YARV : Yet Another Ruby VM (以降 YARV) は笹田耕一が Ruby インタプリタの高速化を目指し, 開発が進められたスタックマシンモデルの仮想マシンである. Ruby 1.9.0 より Ruby の公式の処理系となった. YARV は Ruby プログラムを YARV バイトコードへ変換し, そのバイトコードを仮想マシンで実行することで高速化を実現した [1].

Ruby では, YARV で表現されるメソッド一つに対して制御フレーム一つをスタック領域に積み, `return` とともに一つ降ろす. メソッドの呼出しを把握するコールスタックは, コントロールフレームポインタ (以降 `cfp`) によって指されている [2]. `cfp` は, スタック領域の上端から下端に向かって伸びる.

また, スタックポインタ (以降 `sp`) は, 変数などを把握している内部スタックの上端を指しており, スタック領域の下端から上端に向かって伸びる. `cfp` と `sp` の様子を図 1 に示す.

`cfp` と `sp` が衝突するときに YARV はスタックオーバーフローと判定する. Ruby プログラムにおいては, ソースコード 1 のように C で実装された `times` メソッドのようなネイティブメソッドと, `foo1` のような YARV で表現されるメソッドやブロックが

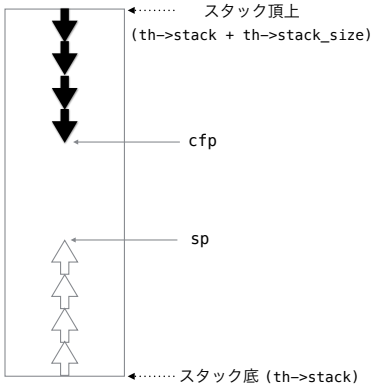

図 1: スタック領域内の `sp` と `cfp`

お互いを呼び合うと, 制御フレームが `return` によって降ろされることなく, スタック領域を使い切ってしまいスタックオーバーフローが生じる.

3 スタック領域の拡張の実装

3.1 概要

スタックオーバーフローが生じた際に, 元のスタック領域よりも大きなスタック領域を確保することでスタックオーバーフローを避ける手法を提案する. その際に, 新しいスタック領域内で処理を進めるために, `cfp`, `sp` の正しい設定を行う必要がある. また, 変数アクセスに用いられる環境ポインタ (以降 `ep`) についても, 新しいスタックに設定する必要がある. スタックの移動について図 2 に示す.

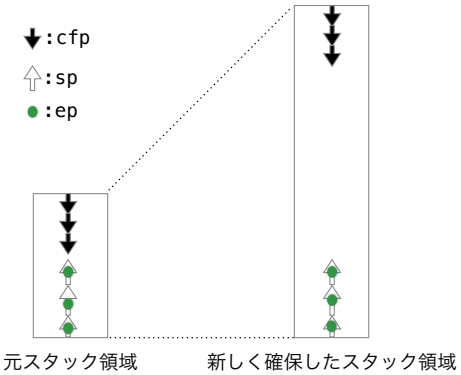

図 2: スタック領域の拡張

3.2 スタック領域

本研究は，開発者による利用率が高い Linux OS 上で開発を行った．Linux OS には，メモリを扱うシステムコールがいくつも備わっている．その中で mmap システムコールを用いることで新しいスタック領域を仮想アドレス空間に確保することが可能となる．また，他の領域にスタックを移動したことで移動処理が済んだ後は，元のスタック領域を解放する仕様とし，より省メモリ化を計った．

3.3 ポインタの設定

cfp は，一つの制御フレームに対して，進む距離は一定である．よって，新しいスタック領域の上端から制御フレームの数と同じだけ進ませることで設定が可能である．しかし，sp および ep については，制御フレーム一つあたりに進む距離は必ずしも一定ではなく，また位置を決定付ける定点もない．よって，これらのポインタの設定は，元スタック領域の開始アドレスと新しいスタック領域の開始アドレスのオフセットをとり相対的に一つずつ設定することとした．

3.4 評価

ソースコード 1 は 2 行目の if 文の条件によって再帰の深さが変わる．スタック領域の大きさを 6 KB と設定した場合，終了条件が a <= 13 のように右辺が 13 以下であれば正常に終了する．しかし，右辺を 14 以上とした場合 2 節で詳述のとおり，スタックを使い切ってしまいスタックオーバーフローを起こす．よって，本研究ではソースコード 1 を評価に用いる．

結果として，従来の実装ではスタックオーバーフローを起こしていたソースコード 1 を本実装では正常に処理することが可能となった．また，スタックオーバーフローを起こさずに処理したい場合，スタックの大きさのチューニングを行う必要がある．このとき，最も再帰の深いスレッドに合わせる必要もある．よって，従来の実装では，40 KB が必要となる．しかし，本研究の実装を用いると，合計 36 KB のみが必要となる．よって，従来の Ruby の実装に比べてスタックオーバーフローを回避するだけでなく省メモリ化に成功したと言える．

また，スレッドの本数が多くなるほど省メモリ化の効果が大きくなる．

ソースコード 1: スタックオーバーフローを起こすプログラム

```
def foo3(a)
  if a <= 14 then foo1(a) end
end

def foo2(n)
  foo3(n+1)
end

def foo1(num)
  1.times { foo2(num) }
end

thr1 = Thread.new { foo1(0) }
thr2 = Thread.new { foo1(3) }
thr3 = Thread.new { foo1(5) }
thr4 = Thread.new { foo1(10) }
thr5 = Thread.new { foo1(12) }

thr1.join
thr2.join
thr3.join
thr4.join
thr5.join
```

4 まとめと今後の展望

従来の Ruby では，マルチスレッド時に一つでも再帰の深さが大きくなるものを含んでいる場合に，それにあわせてスタックサイズをチューニングしなければならなかった．しかし，本研究ではスタックを小さく用意しておくのが基本であり，再帰の深さが大きくなるスレッドのみを特別に処理することで，特にマルチスレッド時に，スレッド一つあたりのスタック領域を小さくすることが可能であるため，省メモリが期待できる．また，スタックオーバーフローを避けるためのスタック領域の拡張は，一度大きくするとその大きさを保持して処理を進めるため繰り返し拡張処理を行う必要がない．よって，プログラム全体の実行時間に比べて時間的負担が少なくボトルネックにはならないと考える．

しかし，一部のプログラムでは正常に処理ができなかった．これは，スタック移動時に余分なものを含んだり，誤った位置に移動していることと，スタック領域の拡張数が複数の場合に対応できていないことが原因と考えられる．これらの対処をすることで，より省メモリな Ruby の実装が可能となる．また，スタック領域の拡張を複数回可能にする事で，スタックの最初の大きさを最小限にする事が可能となり，さらなる省メモリが期待できる．

謝辞 本研究を進めるにあたり，笹田耕一氏にご教示頂きました．ここに感謝の意を表します．

参考文献

[1] 笹田耕一, 松本行弘, 前田敦司, 並木美太郎ほか. 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 (PRO), Vol. 47, No. SIG2 (PRO28), pp. 57–73, 2006.

[2] Pat Shaughnessy 著, 島田浩二・角谷信太郎共訳. Ruby のしくみ (Ruby Under a Microscope). オーム社, 2014.