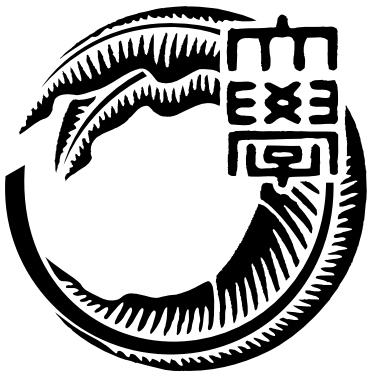
令和元年度 卒業論文

Rakuのサーバーを使った実行



琉球大学工学部情報工学科

165727F 福田光希

指導教員 河野真治

目 次

図 目 次

表 目 次

ソースコード目次

第1章　序論

1.1　背景と目的

現在開発の進んでいる言語に Raku がある. スクリプト言語 Raku は任意の VM が選択できるようになっており, 主に利用されている VM に C で書かれた MoarVM が存在する. MoarVM は JIT コンパイルなどをサポートしているが, 全体的な起動時間及び処理速度が Perl5 や Python , Ruby などの他のスクリプト言語と比較し非常に低速である. その為, 現在日本国内では Raku は実務としてあまり使われていない.

Raku の持つ言語機能や型システムは非常に柔軟かつ強力であるため, 実用的な処理速度に達すれば, 言語の利用件数が向上することが期待される. Raku は MoarVM に基づく JIT コンパイラを持っており, コンパイルされた結果はプロセッサが実行可能な機械語に相当する.

Raku のような言語に JIT コンパイラを持ち, 起動時間が遅い PyPy などの言語がある本研究では, このようなスクリプト言語の起動時間及び, 処理速度の改善を図る

また, その手法として同一ホスト内で終了せずに実行を続けるサーバープロセスを立ち上げ, このサーバープロセス上で立ち上げておいたコンパイラに実行するファイル名を転送し, サーバー上でコンパイルを行う手法を提案する. 著者らは, この提案手法に沿って『Abyss サーバー』を実装している.

またサーバーでは, サーバーに投げられた Raku スクリプトをコンパイラで実行する際に, そのスクリプトが次に実行するスクリプトに影響を与えないことを保証する必要がある.

研究をするにあたり得られた, サーバー上で script 言語を実行する場合の利点と欠点について述べ, 今後の展望について記載する.

1.2　論文の構成

本論文は全 5 章で構成される。2 章では Raku の概要について紹介する。 3 章では 提案手法で述べた「Abyss Server」の具体的な実装について解説する。4 章では Abyss Server の性能評価について解説する。5 章はまとめとなっている。

第2章　Raku の概要

2.1　Raku

Raku は 2002 年に LarryWall が Perl を置き換える言語として設計を開始した. Perl5 の言語的な問題点であるオブジェクト指向機能の強力なサポートなどを取り入れた言語として設計された. Perl5 は設計と実装が同一であり, Larry らによって書かれた C 実装のみだった. Raku は設計と実装が分離している. 言語的な特徴としては, 独自に Raku の文法の拡張が可能な Grammar, Perl5 と比較した場合のオブジェクト指向言語としての進化も見られる. また Perl6 は漸進的型付け言語である. 従来の Perl の様に変数に代入する対象の型や, 文脈に応じて型を変更する動的型言語としての側面を持ちつつ, 独自に定義した型を始めとする様々な型に, 静的に変数の型を設定する事が可能である. Raku は元々は Perl6 という名称であったが, 言語仕様及び処理実装が Perl5 と大幅に異なっており, 言語的な互換性が存在しない. 従って現在では Perl6 と Perl5 は別言語としての開発方針になっている. Raku は現在有力な処理系である Rakudo から名前を取り Raku という別名がつけられている.

Raku の現在の主流な実装は Rakudo である. Rakudo は MoarVM, と NQP と呼ばれる Raku のサブセット, NQP と Raku 自身で記述された Raku という構成である. MoarVM は NQP と Byte Code を解釈する.

NQP とは Not Quite Perl の略で Raku のサブセットである. その為基本的な文法などは Raku に準拠しているが, 変数を束縛で宣言するなどの違いが見られる.

この NQP で記述された Raku の事を Rakudo と呼ぶ. Rakudo は MoarVM の他に JVM , Javascript を動作環境として選択可能である.

Raku の起動は, MoarVM を起動, NQP をロード, Rakudo をロードもしくはコンパイルし, その後 JIT しながら実行する.

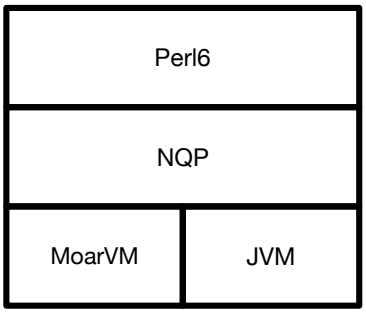


図 2.1: Rakudo の構成

2.2 MoarVM

MoarVM は Rakudo, NQP のために構築された VM である. C 言語で実装されている. JIT コンパイルなどが現在導入されているが, 起動時間が低速であるなどの問題がある. MoarVM 独自の ByteCode があり, NQP からこれを出力する機能などが存在している.

2.3 NQP

NQP とは Raku のサブセットである．その為基本的な文法などは Raku に準拠しているが，変数を束縛で宣言するなどの違いが見られる．

Rakudo における NQP は現在 MoarVM, JVM 上で動作する．NQP は Perl6 のサブセットであるため, 主な文法などは Perl6 に準拠しているが幾つか異なる点が存在する. NQP は最終的には NQP 自身でブートストラップする言語であるが, ビルドの最初にはすでに書かれた MoarVM のバイトコードを必要とする．Raku の一部は NQP を拡張したもので書かれている為, Rakudo を動作させる為には MoarVM などの VM, VM に対応させる様にビルドした NQP がそれぞれ必要となる．現在の NQP では MoarVM, JVM に対応する Stage0 はそれぞれ MoarVM のバイトコード, jar ファイルが用意されている. MoarVM の ModuleLoader は Stage0 にある MoarVM のバイトコードで書かれた一連のファイルが該当する.

Stage0 にあるファイルを MoarVM に与えることで, NQP のインタプリタが実行される様になっている．これは Stage0 の一連のファイルは, MoarVM のバイトコードなどで記述された NQP コンパイラのモジュールである為である．NQP のインタプリタはセルフビルドが完了すると, nqp というシェルスクリプトとして提供される．このシェルスクリプトは, ライブラリパスなどを設定して MoarVM の実行バイナリである moar を起動するものである.

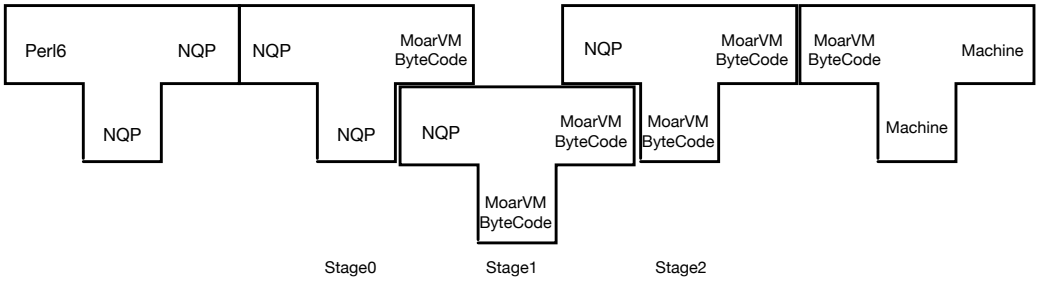


図 2.2: NQP のビルドフロー

NQP のビルドフローを図 2.2 に示す．Rakudo による Perl6 処理系は NQP における nqp と同様に, moar にライブラリパスなどを設定した perl6 というシェルスクリプトである．この perl6 を動かすためには self build した NQP コンパイラが必要となる．その為に Stage0 を利用して Stage1 をビルドし NQP コンパイラを作成する．Stage1 は中間的な出力であり, 生成された NQP ファイルは Stage2 と同一であるが, MoarVM のバイトコードが異なる．Perl6 では完全なセルフコンパイルを実行した NQP が要求される為, Stage1 を利用してもう一度ビルドを行い Stage2 を作成する．
Perl6 のテストスイートである Roast やドキュメントなどによって設計が定まっている Perl6 とは異なり NQP 自身の設計は今後も変更になる可能性が開発者から公表されている．現在の公表されている NQP のオペコードは NQP のリポジトリに記述されているものである．

2.4 なぜ Raku は遅いのか

通常 Ruby のようなスクリプト言語ではまず YARV などのプロセス VM が起動し, その後スクリプトを Byte code に変換して実行という手順を踏む. Rakudo はインタプリタの起動時間及び, 全体的な処理時間が他のスクリプト言語と比較して非常に低速である. これは Rakudo 自体が Raku と NQP で書かれているため, MoarVM を起動し, Rakudo と NQP の Byte code を読み取り, Rakudo を起動し, その後スクリプトを読み取り, スクリプトの Byte code 変換というような手順で進むためである. また Perl6 は実行時の情報が必要であり, メソッドを実行する際に invoke が走ることも遅い原因である. invoke は MoarVM の method 呼び出しの Byte code である.

2.5 Raku と他言語との起動時間の比較

Raku と他言語の起動時間の比較行なった．題材として perl5, ruby, raku, python で helloworld を出力するプログラムを用いて行なった実行結果である.
実行環境

- macOS Mojave version 10.14.5
- メモリ 8GB
- プロセッサ 2.7GHz Intel Core i5

Language	version
Raku	2019.03.1
Perl5	v5.18.4
Python	2.7.10
Ruby	2.3.7p456

表 2.1: 比較言語のバージョン

Language	Time
Raku	0.249 sec
Perl5	0.004 sec
Python	0.013 sec
Ruby	0.083 sec

表 2.2: 起動時間の比較

第3章　Raku によるAbyss の実装

提案手法に沿い『Abyss Server』を実装した.

3.1　サーバーの構成

Abyss サーバーは クライアント側から投げられた Raku スクリプト を実行するためのサーバーである. 図3.1 は, Abyss サーバーを用いたスクリプト言語実行手順である. Abyss サーバーはユーザーが Raku を直接立ち上げるのではなく, まず 図3.1 右側の Abyss サーバーを起動し, ユーザーは Abyss サーバーにファイルパスをソケット通信で送り, Abyss サーバーがファイルを開き実行し, その実行結果をユーザーに返す.

この手法を用いることで, サーバー上で事前に起動した Rakudo を再利用し, 投げられた Raku スクリプトの実行を行うため, Rakudo の起動時間を短縮できると推測できる.

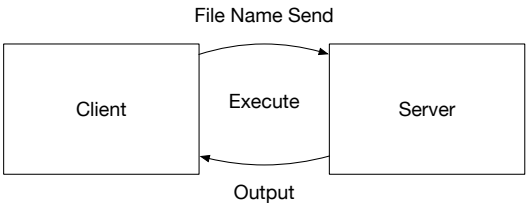


図 3.1: Abyss サーバーを用いたスクリプト言語実行手順

3.2　Abyss Server 側の実装

3.1 は Abyss サーバーのソースコードである. Abyss サーバーは起動すると, まず自身にファイルパスを転送するためのソケットを生成し, その後ファイルパスを受け取り実行して . 出力結果をソケットに書き込む

Listing 3.1: Abyss Server 側の source code

```
1 use v6.c;
2 unit class Abyss::Server:ver<0.0.1>:auth<cpan:ANATOFUZ>;
3 use MONKEY-SEE-NO-EVAL;
4 use IO::Socket::Unix;
```

```
5  use NativeCall;
6
7  sub close(int32) returns int32 is native { ... }
8  sub dup(int32 $old) returns int32 is native { ... }
9  sub dup2(int32 $new, int32 $old) returns int32 is native { ... }
10
11 method readeval
12 {
13     my $listen = IO::Socket::Unix.new( :listen,
14                                        :localhost<localhost>,
15                                        :localport(3333) );
16     my $backup = dup(1);
17     say DateTime.now;
18
19     loop
20     {
21         my $conn = $listen.accept;
22         my $sock_msg;
23         my $buf = $conn.recv();
24         $sock_msg = $buf;
25         close(1);
26         dup2($conn.native-descriptor(), 1);
27         EVALFILE $sock_msg;
28         dup2($backup, 1);
29         close($backup);
30         $conn.close;
31     }
32
33     $listen.close;
34 }
```

3.3 Abyss Client 側の実装

ユーザーは Abyss Server を起動後, Client 側からファイルパスをサーバーに送信し, **??** のように Socket に書き込まれた実行結果を読み取る.

Listing 3.2: Abyss Client 側の source code

```
1  use IO::Socket::Unix;
2  use NativeCall;
3
4  sub close(int32) returns int32 is native { ... }
5  sub dup(int32 $old) returns int32 is native { ... }
6  sub dup2(int32 $old, int32 $new) returns int32 is native { ... }
7
8  my $conn = IO::Socket::INET.new( :host<localhost>,
9                                   :port(3333) );
10
11 $conn.print: 'Absolute␣file␣path';
12
13 say $conn.lines;
14
15 $conn.close;
```

3.4 Raku の EVAL

Raku では EVAL 関数 [3.3] があり文字列を Raku のソースコード自身として評価できる

Raku では, EVAL は通常は使用できないようになっており, MONKEY-SEE-NO-EVAL という pragma を実行することで使うことができるようになる. EVALFILE はファイルパスを受け取るとファイル開き, バイト文字列に変換し読み込む, その後読み込んだバイト文字列にデコードし, ファイルパスの文字列を読み込み, ファイルの中身を EVAL と同様に解釈する.

?? の 2 行目にある MONKEY − SEE − NO − EVAL は Perl6 上で EVALFILE を使用可能にする pragma である.

Listing 3.3: eval のサンプルコード

```
1 use MONKEY-SEE-NO-EVAL;
2
3 EVAL "say␣{␣5␣+␣5␣}"; # OUTPUT: 10
```

3.5 NativeCall

第4章　Abyss Server の性能評価

4.1　比較

- Microsoft CLR
  .NET Framework には, 共通言語ランタイム (Common Language Runtime) と呼ばれるランタイム環境がある．.NET 対応のソフトウェアは, 様々なプログラミング言語で書かれたソースコードから, いったん共通中間言語 (Common Intermediate Language）による形式に変換されて利用者のもとに配布される. CIL 形式のプログラムを解釈し, コンピュータが直に実行可能な機械語によるプログラムに変換して実行するソフトウェアが CLR である. 現状の Abyss サーバーはプロセスとして立ち上げているが, CLR は OS に直接組み込む必要があるが, Abyss サーバーはプロセス上で実行しているため OS に手を加えず実装が容易である.

- PyPy
  PyPy は Python の 実装の一つであり, Cpython のサブセットである RPython で記述された 処理系である.
  PyPy は JIT コンパイル を採用しており, 実行時にコードを機械語にコンパイルして効率的に実行させることができる. PyPy は Cpython より実行速度が速いが起動速度は Cpython と比較して約 3 倍遅い. Perl6 と同様, PyPy は Cpython と比較して起動時間が遅いため今回提案した手法を応用できると予測できる.

4.2　Abyss Server の利点

- Abyss Server を用いて実行することで, サーバー上で事前に起動した Rakudo を再利用し, 投げられた Raku スクリプトの実行を行うため, Rakudo の起動時間を短縮できる.

- 一度投げられたスクリプトのバイトコード, もしくは計算結果をキャッシュで保存しておき, 再度実行する際に, そのキャッシュを用いてコンパイル時間を省くような仕組みを入れやすいと考えられる.

- 他の起動時間遅いスクリプト言語や, モジュールの読み込みが遅い言語などにも, 応用しやすいと考えられる.

- 普通のスクリプト言語だと実行するたびに fork して実行しインタプリタの立ち上げという処理になるが, プロセス毎回起動しなくて済む

4.3 Abyss Server の欠点

- 現在 Abyss Server には 一度スクリプトを実行した後にサーバー内の環境をリセットする機能が存在しないため, スクリプトがサーバー内の環境に影響を及ぼした場合, 通常実行と違う挙動をする危険性がある
- 同時に二つ以上のタスクを与えられると実行順のスケジューリングができない
- 異常に長いタスクが投げられた場合, 次のタスクが前のタスクが終わるまで実行ができない
- 起動時のオプションが選択出来ない

第5章　結論

5.1　まとめ

本稿では実行する Raku スクリプトのファイル名をサーバーに転送し, コンパイラサーバーでコンパイルを行い実行することで全体的に処理時間が早くなることを示した.
今後 Abyss サーバーでの開発をより深く行っていくにあたって以下のような改善点が見られた

- コンパイラの起動が遅い言語だけでなく, モジュールの読み込みが遅い言語などを, あらかじめサーバーを側でモジュールを読み込んでおき, それを利用してプログラムを実行する手法も応用できるように改良を行う.
- モジュールを送信する機能の追加
- プログラムの実行終了したらモジュールを削除する機能の追加

今後の開発を行っていくにあたって, 他の Python のような script 言語にも応用できるように開発を行っていく.

参考文献

[1] Andrew Shitov. Perl6 Deep Dive

[2] 清水隆博, 河野真治. CbC を用いた Perl6 処理系. プログラミングシンポジウム論文, 2019.

[3] Perl6 Documentation
(https://docs.perl6.org) (2019/10/22 access)

[4] rakudo
(https://github.com/rakudo/rakudo) (2020/2/14 access)

[5] The Official Raku Test Suite
(https://github.com/perl6/roast/)

[6] NQP - Not Quite Perl
(https://github.com/perl6/nqp)

[7] MoarVM
(https://github.com/MoarVM/MoarVM) (2020/2/14 access)

[8] ThePerlFoundation: Perl 6 Design Docu- ments, ThePerlFoundation (online), available from (https://design.raku.org)

[9] C Documentation
(https://devdocs.io/c/) (2020/2/14 access)

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に深く感謝致します。本研究の遂行及び本論文の作成にあたり、数々の貴重な御助言と細かな御配慮を戴いた、 東恩納琢偉さん、 外間政尊さん、 桃原優さん、 清水隆博さん、 並びに並列信頼研究室の皆様に深く感謝致します。

また、 毎週のゼミでお世話になっているハリーズの皆様に感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、 並びに物心両面で支えてくれた両親に深く感謝致します。

2020 年 2 月

福田光希