Perl6 のサーバを使った実行

福田 光希[1,a]　河野 真治[1,b]

概要：
スクリプト言語である Perl5 の後継言語として Perl6 が現在開発されている．Perl6 は設計と実装が区分されており様々な処理系が開発されている．現在主流な Perl6 は Rakudo と言われるプロジェクトである．Rakudo では Perl6 自体を NQP(NotQuitPerl) と言われる Perl6 のサブセットで記述し, NQP を VM が解釈するという処理の流れになっている．この VM 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 と比較し非常に低速である．この問題を解決するために Continuation based C (CbC) という言語を一部用いて MoarVM の書き換えを行う．CbC は C よりも細かな単位で記述が可能である為, 言語処理系の実装に適していると考えられる．CbC に関する今までの研究においては, 言語処理系に CbC を利用した事例が少ない．その為, 本稿では CbC を言語処理系に用いた場合の利点やデバッグ手法などについても述べる．

キーワード：プログラミング言語, コンパイラ, CbC, Perl6, MoarVM, Raku

1. スクリプト言語の高速実行

スクリプト言語 Perl6 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 などの主要なスクリプト言語と比較し非常に低速である．この問題を解決するために, Perl6 プログラムのサーバーを用いた実行手法の提案を行う．ここでいうサーバーとは転送したスクリプトを実行する環境のことである．

またサーバーでは, サーバーに投げられた Raku 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．この問題を解決するために, サーバーのコンテナ化を行う．

[1] 琉球大学工学部情報工学科
[a] k.fukuda@cr.ie.u-ryukyu.ac.jp
[b] kono@ie.u-ryukyu.ac.jp

2. Raku

この章では現在までの Raku の遍歴及び Raku の言語的な特徴について記載する．

2.1 Raku の構想

Perl6 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.Perl6 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 Perl6

の文法を拡張可能な 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. Perl6 は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.Perl6 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 は元は Perl5 の後継言語の Perl6 として開発されていたが, 現在は名称が変更され Raku となっている. 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.

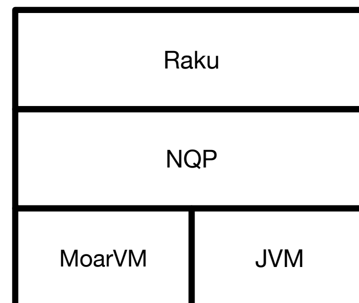

図 1: Rakudo の構成

2.2 MoarVM

2.3 NQP

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. この MoarVM のバイトコードの状態を Stage0 と言う.Perl6 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.

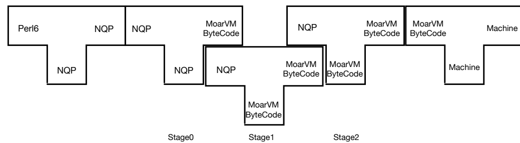

図 2: NQP のビルドフロー

NQP のビルドフローを図 2 に示す．Rakudo による Perl6 に処理系は 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 rubyVM 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 で書かれているため,MoarVM を起動し,Rakudo と NQP をコンパイルし, その後スクリプトの Byte code 変換というような手順で進むためである。また Raku 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.

3. Abyss サーバー

ここでは Abyss サーバーについて説明する.Abyss サーバーは Raku で書かれている．クライアント側から投げられた Raku を実行するためのサーバーである．図 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 を実行する際, クライアント側から転送されてきたファイルを事前に起動してあるサーバー側が処理し, その実行結果を返す構造となっている.

この手法を用いることで, サーバー上で事前に Rakudo を起動した 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.

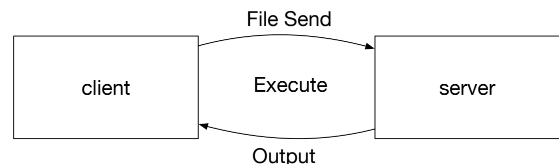

図 3: Abyss サーバーを用いたスクリプト言語実行手順

Code1 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を転送するためのソケットを生成し, その後ファイルを受け取るための待機ループに入る.

ファイルパスを受け取るとファイルパスをバッファーに変換し読み込む, その後読み込んだバッファーを文字列にデコードし, ファイルパスの文字列を読み込み, ファイルの中身を式として評価する EVALFILE を用いて, プログラムを実行する. Code1 の 2 行目にある MONKEY − SEE − NO − EVAL は Raku 上で EVALFILE を使用可能にする pragma である．現状の Raku の EVALFILE では, 出力がサーバー側に返っているので, クライアント側から出力を見るためにクライアント側に返す必要がある.

```
unit class Abyss::Server:ver<0.0.1>;
```

```
use MONKEY-SEE-NO-EVAL;

method readeval {
  my $listen = IO::Socket::INET.new( :listen,
                                     :localhost<
                                          localhost
                                          >,
                                     :localport
                                          (3333)
                                          );
  loop {
      my $conn = $listen.accept;
          while my $buf = $conn.read(1024) {
              EVALFILE $buf.decode;
          }
      $conn.close;
  }
}
```

Code 1: Abyss サーバーの source code

```
my $conn = IO::Socket::INET.new( :host<
     localhost>,
                                 :port(3333) );

$conn.print: 'FILEPASS';
```

Code 2: クライアント側の source code

4. まとめ

中間予稿までに Perl6 スクリプトを投げて実行するサーバーの実装、および「自分でプロセス立ち上げて Perl6 実行する手法」と「既にあるサーバーに投げて Perl6 スクリプトを実行する手法」の差を測るために時間の計測を行った。

今回実装したサーバーでは, 別のスクリプトを実行する前にサーバーのコンテナ化をできていないので次回以降の課題とする．今回の実装では TCP ソケットを用いたが TCP ソケットを用いるとサーバーを立ち上げた際に外部からファイルを転送される可能性があるので,Unix domain socket の実装を行い, それを用いたクライアント・サーバーを作成することで安全性が高まると考えた Raku には現状 Unix domain ソケットの実装がないので、Unix domain ソケットを実装し、自分以外が実行できないようにすることが今後の課題に挙げられる．また今回例題として用いたものはスクリプト言語 Raku であったが, その他のスクリプト言語にも応用が利くかどうか検討する必要はある

今回用いた Raku の EVALFILE 自体にクライアント側に出力を返す実装追加することも今後の課題に挙げられる.

謝辞　謝辞が必要であれば，ここに書く.

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. 琉球大学工学部情報工学科平成 30 年度学位論文 (学士), 2018.
[3] Perl6 Documentation https://docs.perl6.org (2019/10/22 アクセス)
[4] 奥村晴彦, 黒木裕介: **LaTeX2e 美文書作成入門**. 技術評論社, 2013.