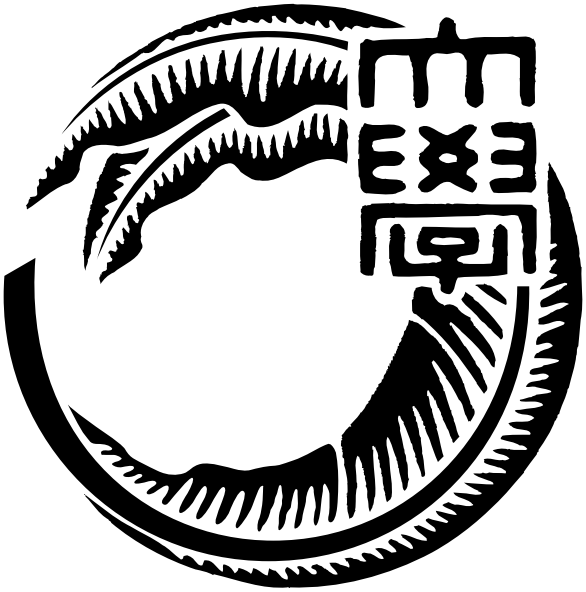
平成30年度 卒業論文

CbC による Perl6処理系



琉球大学工学部情報工学科

155730B 清水 隆博

指導教員 河野 真治

目 次

第 1 章　現在の Perl6 処理系 **1**

第 2 章　Perl6 **2**

2.1 Perl6 の概要 2
2.2 Rakudo 2
2.3 MoarVM 3
2.4 NQP 3

第 3 章　Continuation Based C **5**

3.1 CbC の概要 5
3.2 CodeGear 5

第 4 章　MoarVM のバイトコード実行 **7**

4.1 オリジナルの MoarVM の処理 7

図 目 次

コード目次

2.1 Rakudo の実装の一部 2
2.2 フィボナッチ数列を求める NQP のソースコード 3
2.3 NQP が加算命令を生成する箇所 4
3.1 加算と文字列を設定する CbC コードの例 6

第1章　現在のPerl6処理系

現在開発が進んでいるプログラミング言語に Perl6 がある。Perl6 は設計と実装が分離しており、 現在の主要な実装は Rakudo と呼ばれている。Rakudo は Perl6 のサブセットである、NQP と呼ばれる言語を中心に、 記述されている。NQP 自体はプロセス仮想機械と呼ばれる、 言語処理系の仮想機械で実行される。Rakudo の場合実行する仮想機械は、Perl6 専用の処理系である MoarVM、 Java 環境の JVM が選択可能である。

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。また、 実行環境である MoarVM の実装事態も複雑であり、巨大な case 文が利用されているなど、 見通しが悪くなっている。

当研究室で開発しているプログラミング言語に、 Continuation Based C (CbC) がある。CbC は C と互換性のある言語であり、 関数より細かな単位である、 CodeGear を用いて記述することが可能となる。CbC では各 CodeGear 間の移動に、 環境などを保存せず次の状態に移動する軽量継続を用いている。軽量継続を用いる事が可能である為、 C 言語におけるループや関数呼び出しを排除する事が可能となる。

現在までの CbC を用いた研究においては、 CbC の言語処理系への応用例が少ない。スクリプト言語処理系では、 バイトコードから実行するべき命令のディスパッチの際に switch 分や gcc 拡張のラベル goto などを利用している。これらは通常巨大な switch-case 文となり、特定の C ファイルに記述せざるを得なくなる。CbC の場合、 この case 文相当の CondeGear を生成する事が可能である為、 スクリプト言語処理系の記述に適していると考えられる。またこの命令ディスパッチ部分は、 スクリプト言語の中心的な処理である為、 スクリプト言語の改修にはまず中心部分の実装から変更したい為、この箇所を修正する。

MoarVM は C 言語で記述されており、 C と互換性のある言語であれば拡張する事が可能となる。CbC は C と互換性のある言語である為、 MoarVM の一部記述を CbC で書き換える事が可能となる。CbC における CodeGear は、 関数より細かな単位として利用出来る為、 MoarVM の命令ディスパッチの巨大な case 文の書き換えが CodeGear を用いることで可能であると考える。

本研究では CbC を用いて Perl6 の実行環境である、 MoarVM の命令ディスパッチ部分の処理の書き換えを検討する。

第2章　Perl6

2.1　Perl6の概要

Perl6 は現在開発が勧められているプログラミング言語である。スクリプト言語 Perl5 の次期バージョンとして当初は開発されていたが、 現在では互換性の無さなどから別言語として開発されている。

Perl6 は仕様と実装が分離されており、 現在はテストスイートである Roast が仕様となっている。実装は歴史的に様々なものが開発されており、 Haskell で実装された Pugs、Python との共同実行環境を目指した Parrot などが存在する。Pugs や Parrot は現在は歴史的な実装となっており、 開発は行われていない。現在の主要な実装である Rakudo は、Parrot と入れ替わる形で実装が進んでいる。Perl6 そのものはスクリプト言語として実装されており、 漸進的型付け言語である。言語的な特徴としては、 独自に Perl6 の文法を拡張可能な Grammer、 Perl5 と比較してオブジェクト指向言語としての機能の強化などが見られる。

2.2　Rakudo

Rakudo とは NQP によって記述され、MoarVM、JVM、Javascript 上で動作する Perl6 の実装である。Rakudo 自体は NQP で実装されている箇所と、 Perl6 で実装されている箇所が混在する。これらは拡張子によって区別され、 NQP は.nqp、 Perl6 は .pm6 が拡張として設定されている。実際に NQP で実装されている箇所の一部を Code2.1 に示す。

Code 2.1: Rakudo の実装の一部

```
1  use Perl6::Grammar;
2  use Perl6::Actions;
3  use Perl6::Compiler;
4
5  # Initialize Rakudo runtime support.
6  nqp::p6init();
7
8  # Create and configure compiler object.
9  my $comp := Perl6::Compiler.new();
10 $comp.language('perl6');
11 $comp.parsegrammar(Perl6::Grammar);
12 $comp.parseactions(Perl6::Actions);
13 $comp.addstage('syntaxcheck', :before<ast>);
14 $comp.addstage('optimize', :after<ast>);
15 hll-config($comp.config);
16 nqp::bindhllsym('perl6', '$COMPILER_CONFIG', $comp.config);
```

Rakudo をソースコードからビルドする際は予め NQP インタプリタである nqp をビルドする必要が存在する。Rakudo のビルド時にはこの nqp と、 nqp が動作する VM を設定として与える必要がある。この両者を指定しない場合、ビルド時に動的に NQP、 MoarVM をソースコードをダウンロードし、 ビルドを行う。

2.3 MoarVM

MoarVM とは Rakudo 実装で主に使われる仮想機械である。Rakudo では Perl6 と NQP を実行する際に仮想機械上で実行する。この仮想機械は OS レベルの仮想化に使用する VirtualBox や qemu と異なり、プロセスレベルの仮想機械である。Rakudo ではこの仮想機械に MoarVM、 Java の仮想機械である JVM(JavaVirtualMachine) が選択可能である。MoarVM はこの中で Rakudo 独自に作成された仮想機械であり、 現在の Rakudo プロジェクトの主流な実装となっている。

MoarVM は C 言語で実装されており、 レジスタベースの仮想機械である。MoarVM は NQP や Perl6 から与えられた MoarVM バイトコードを評価する。

2.4 NQP

NQP とは Rakudo における Perl6 の実装に利用されているプログラミング言語である。NQP 自体は、 Perl6 のサブセットとして開発されている。歴史的には Perl6 の主力実装が Parrot であった際に開発され、 現在の Rakudo に引き継がれている。Rakudo における NQP は、 Parrot 依存であった実装が取り払われている。

基本文法などは Perl6 に準拠しているが、 変数を束縛で宣言する。インクリメント演算子が一部利用できない。Perl6 に存在する関数などが一部利用できないなどの制約が存在する。

NQP のコード例を Code2.2 に示す。

Code 2.2: フィボナッチ数列を求める NQP のソースコード

```
1  #! nqp
2
3  sub fib($n) {
4      $n < 2 ?? $n !! fib($n-1) + fib($n - 2);
5  }
6
7  my $N := 29;
8
9  my $t0 := nqp::time_n();
10 my $z := fib($N);
11 my $t1 := nqp::time_n();
12
13 say("fib($N)␣=␣" ~ fib($N));
14 say("time␣␣␣␣=␣" ~ ($t1-$t0));
```

Perl6 は NQP で実装されている為、 Perl6 における VM は NQP の実行を目標として開発されている。

NQP 自体も NQP で実装されており、 NQP のビルドには予め用意された MoarVM などの VM バイトコードによる NQP インタプリタが必要となる。実際に NQP 内部で入力として与えられた NQP から加算命令を生成する部分を Code2.3 に示す。

Code 2.3: NQP が加算命令を生成する箇所

```
$ops.add_hll_op('nqp', 'preinc', -> $qastcomp, $op {
    my $var := $op[0];
    unless nqp::istype($var, QAST::Var) {
        nqp::die("Pre-increment␣can␣only␣work␣on␣a␣variable");
    }
    $qastcomp.as_mast(QAST::Op.new(
        :op('bind'),
        $var,
        QAST::Op.new(
            :op('add_i'),
            $var,
            QAST::IVal.new( :value(1) )
        )));
});
```

MoarVM を利用する場合、 MoarVM の実行バイナリである moar に対して、 ライブラリパスなどを予め用意した NQP インタプリタのバイトコードに設定する。設定はオプションで与える事が可能であり、 moar を実行することで NQP のインタプリタが起動する。NQP のビルドには、 この NQP インタプリタをまず利用し、 NQP 自体のソースコードを入力して与え、 ターゲットとなる VM のバイトコードを生成する。このバイトコードは NQP ソースコードから生成された NQP インタプリタのバイトコードであり、 次にこのバイトコードをライブラリとして moar に与え、 再び NQP をビルドする。この 2 度目のビルドで、ソースコードからビルドされた VM バイトコードで NQP 自身をビルドした事になる。処理系自身をその処理系でビルドする事をセルフビルドと呼び、 NQP はセルフビルドしたバイナリを利用する。2 度目のビルドの際に生成された NQP インタプリタの事を小文字の nqp と呼び、これが NQP のインタプリタのコマンドとなる。

第3章　Continuation Based C

3.1　CbCの概要

Continuation Based C (CbC) は当研究室で開発を行っているプログラミング言語である。現在はC コンパイラである gcc 及び llvm をバックエンドとした clang、 MicroC コンパイラ上の 3 種類の実装がある。

C 言語を用いてプログラミングを行い場合、本来プログラマが行いたい処理の他に、malloc などの関数を利用したメモリのアロケートなどのメモリ管理が必要となる。他にもエラーハンドリングなどの雑多な処理が必要となる。

これらの処理を meta computation と呼ぶ。実装しているプログラムにおけるエラーの原因が、 通常の処理か meta computation なのか区別を行いたい。また、 プログラム自身の検証や証明も、 通常の関数などと meta computation は区別したい。通常 C 言語などを用いたプログラミングの場合、 meta computation と通常の処理を分割を行おうとすると、 それぞれ事細やかに関数やクラスを分割せねばならず容易ではない。

CbC では関数より meta computation を細かく記述する為に、 CodeGear と呼ばれる単位を導入する。CodeGear では、 データの入出力として DetaGear という単位を導入する。CbC では、 これら CodeGear と DetaGear を基本単位として実装していくプログラミングスタイルを取る。

3.2　CodeGear

CbC では C 言語の関数の代わりに CodeGear を導入する。CodeGear は C 言語の関数宣言の型名の代わりに __code と書く事で宣言出来る。__code は CbC コンパイラでの扱いは void と同じ型として扱うが、 CbC プログラミングでは CodeGear である事を示す、 識別子として利用する。

CodeGear 間の移動は goto 文で行い、 goto の後に対応する CodeGear 名を記述することで遷移する。通常 C 言語の関数呼び出しでは、 スタックポインタを操作し、 ローカル変数や、 レジスタ情報をスタックに保存する。これらは通常アセンブラの call 命令として処理される。

CbC の場合、 スタックフレームを操作せずに次の CodeGear に遷移する。この際 C の関数呼び出しとは異なり、 プロラムカウンタを操作するのみの jmp 命令として処理される。通常 Scheme の call/cc などの継続と呼ばれる処理は、 現在のプログラムまでの位置

や情報を、 環境として所持した状態で遷移する。CbC の場合これら環境を持たず遷移する為、 通常の継続と比較して軽量であるから、 軽量継続であると言える。CbC は軽量継続を利用するためにレジスタレベルでの実装が可能となり、 C よりも低級な言語と言える。

CodeGear 間の入出力の受け渡しは引数を利用して行う。これは軽量継続時に、CodeGear の入出力のインターフェイスを揃えることで、 引数で与えられたレジスタを変更せずに繊維する事が可能であるためである。

実際に CbC で書いたコード例を Code3.1 に示す。

Code 3.1: 加算と文字列を設定する CbC コードの例

```
1  extern int printf(const char*,...);
2
3  typedef struct test_struct {
4      int number;
5      char* string;
6  } TEST, *TESTP;
7
8
9  __code cg1(TEST);
10 __code cg2(TEST);
11 __code cg3(TEST);
12
13 __code cg1(TEST testin){
14     TEST testout;
15     testout.number = testin.number + 1;
16     testout.string = testin.string;
17     goto cg2(testout);
18 }
19
20 __code cg2(TEST testin){
21     TEST testout;
22     testout.number = testin.number;
23     testout.string = "Hello";
24     goto cg3(testout);
25 }
26
27 __code cg3(TEST testin){
28     printf("number␣=␣%d\t␣string=␣%s\n",testin.number,testin.string);
29 }
30
31 int main(){
32     TEST test = {0,0};
33     goto cg1(test);
34 }
```

この例では、 cg1, cg2, cg3 という CodeGear を用意し、これらを cg1,cg2,cg3 の順で軽量継続していく。入出力として main 関数で生成した TEST 構造体を受け渡し、 cg1 で数値の加算を、 cg2 で文字列の設定を行う。main 関数から cg1 への goto 文では、 C の関数から CodeGear への移動となる為、 call 命令ではなく jmp 命令で行われる。cg1 から cg2、また cg2 から cg3 へは、 CodeGear 間での移動となるため jmp 命令での軽量継続で処理される。この例では最終的に test.number には 1 が、 test.string には Hello が設定される。

第4章　MoarVMのバイトコード実行

4.1　オリジナルのMoarVMの処理

MoarVM では与えられたバイトコードを関数 MVM_interp_run 中で解釈、評価する。この関数中では C 言語の case/switch 文、 もしくは、 C 言語のラベルに対しての goto 文が使用する C コンパイラの拡張として含まれている場合は、 ラベル goto を利用して次の命令に遷移する。

MoarVM では実行するべき命令の並びである命令列を、 変数 cur_op が指しており、 この値を利用することで処理を進める。現在実行しているバイトコード命令は、 変数 op が指し示している。

参考文献

[1] 唐鳳. Pugs: A perl 6 implementation.

[2] ThePerlFoundation. Perl6 documentation.

[3] ThePerlFoundation. Perl 6 design documents.

[4] ThePerlFoundation. Roast – perl6 test suite.

[5] ParrotFoundation. Parrot.

[6] ThePerlFoundation. Nqp opcode list.

[7] ThePerlFoundation. Nqp - not quite perl (6).

[8] 大城信康, 河野真治. Continuation based c の gcc 4.6 上の実装について. 第 53 回プログラミング・シンポジウム, 1 2012.

[9] 徳森海斗, 河野真治. Llvm clang 上の continuation based c コンパイラの改良. 琉球大学工学部情報工学科平成 27 年度学位論文 (修士), 2015.

[10] 並列信頼研究室. Cbc_gcc.

[11] 並列信頼研究室. Cbc_llvm.

[12] Jonathan Worthington. Rakudo and nqp internals.

[13] Jonathan Worthington. Rakudo and nqp internals - day1.

[14] Anton Ertl. Threaded code.

[15] Kaito TOKUMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA*, 7 2015.

[16] 光希宮城, 優桃原, 真治河野. Gears os のモジュール化と並列 api. Technical Report 11, 琉球大学大学院理工学研究科情報工学専攻, 琉球大学大学院理工学研究科情報工学専攻, 琉球大学工学部情報工学科, may 2018.

[17] 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プログラミング（PRO）, 2 2006.

[18] James R. Bell. Threaded code. *Commun. ACM*, Vol. 16, No. 6, pp. 370–372, June 1973.

[19] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pp. 291–300, New York, NY, USA, 1998. ACM.

[20] Mike Pall. The luajit project.

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したします。また、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伊波立樹さん、比嘉健太さん、並びに並列信頼研究室の皆様に深く感謝致します。

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。

2019 年 2 月

清水隆博