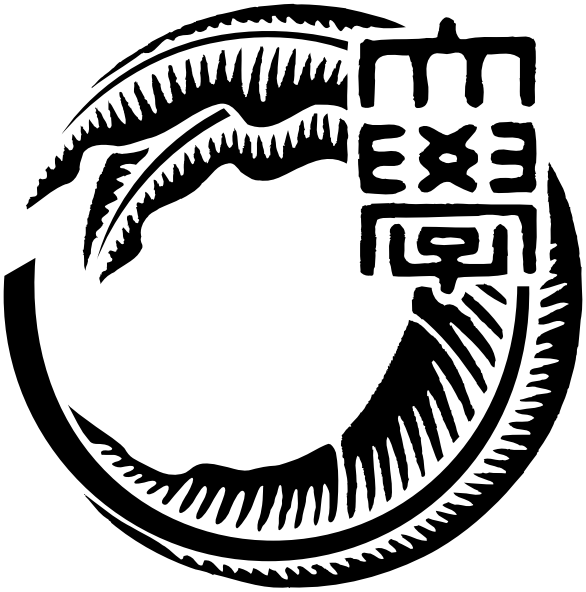
平成30年度 卒業論文

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

図 目 次

表 目 次

ソースコード目次

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　Continuation Based C

2.1　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。

2.2　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は通常のC言語の関数とは異なり、 返り値を持たない。そのためreturn文や返り値の型などの情報が存在しない。

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。軽量継続を利用する為、 ループ文を軽量継続の再帰呼び出しなどで実現する事が可能となる。

CodeGear 間の入出力の受け渡しは引数を利用して行う。

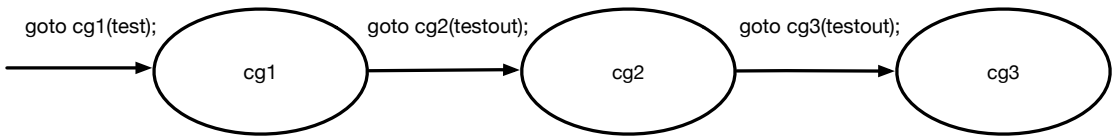
これは軽量継続時に、 CodeGear の入出力のインターフェイスを揃えることで、 引数で与えられたレジスタを変更せずに繊維する事が可能であるためである。

実際に CbC で書いたコード例をソースコード 2.1 に示し、 コード中での状態遷移を図 2.1 に示す。

ソースコード 2.1: 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 }
```

図 2.1: ソースコード 2.1 における CodeGear の状態遷移



この例では、cg1, cg2, cg3 という CodeGear を用意し、これらを図 2.1 の通り、cg1,cg2,cg3 の順で軽量継続していく。それぞれの CodeGear へは、 goto 文を利用する。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。

CbC では関数呼び出しの他に、 for 文や while 文などのループ制御を廃している。CbC でループ相当の物を記述する際は、 再帰呼び出しを利用する。C 言語で、ループを再帰呼び出しで表現する場合、 再帰呼び出しの度にスタックに値が積まれていく為に、 スタック領域を埋め尽くしてしまいスタックオーバーフローが発生する。これを回避するには、末尾再帰と呼ばれる形でのプログラミングが要求される。CbC の場合、 CodeGear 同士の軽量継続は、 強制的に末尾再帰の形になる。また、 CodeGear から CodeGear への遷移は、 スタックを利用しない。その為、 CodeGear の再帰呼び出しを利用しても、 スタックオーバーフローを発生させることがない。この処理を末尾呼び出し除去 (tail call elimination) と呼び、 CbC コンパイラは、 各 CodeGear の遷移を末尾再帰に変換する。

実際にある数の階乗を計算するプログラムを CbC 書いた場合のコードをソースコード 2.2 に示す。

ソースコード 2.2: 階乗を求める CbC のサンプルコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  __code initialize(char* input){
5      int start_number = atoi(input);
6      goto fact(start_number,1);
7  }
8
9  __code fact(int cur,int result){
10     if ( cur > 0 ){
11         result *= cur;
12         cur--;
13         goto fact(cur,result);
14     }
15     goto print_fact(result);
16 }
17
18 __code print_fact(int result){
19     printf("result␣=␣%d\n",result);
20 }
21
22 int main(int argc, char** argv){
23     if ( argc == 1){
24         printf("require␣arg\n");
25         exit(1);
26     }
27     goto initialize(argv[1]);
28 }
```

2.3 Cとの互換性

CbC コンパイラはコンパイル対象のソースコードが、 CbC であるか C 言語であるかを判断する。この際に CodeGear を利用していない場合は、 通常の C プログラムとしてコンパイルを行う。本研究で検証する MoarVM のビルドにおいても、 CbC で書き換えたソースコードがある MoarVM と、 手を加えていないオリジナルの C 言語で実装された MoarVM を同一の CbC コンパイラでビルドする事が可能である。

また、C 言語の関数から CodeGear へ繊維することは goto 文で可能である。CodeGear 中で C の関数を呼び出し、 その結果を受取り、 次の CodeGear に遷移する事も通常の C のプログラミング同様可能である。

しかし CodeGear から C の関数に再び戻り、 CodeGear 同士の遷移から外れるように実装したい場合がある。この際は環境付き goto と呼ばれる手法を取る。これは_CbC_return 及び、 _CbC_environment という変数を使用する。この変数は _CbC_return が元の環境に戻る際に利用する CodeGear を指し、 _CbC_environment は復帰時に戻す元の環境である。復帰する場合、 呼び出した位置には帰らず、 呼び出した関数の終了する位置に帰る。実際に環境付き継続を利用した場合のサンプルコードをソースコード 2.3 に示す。

ソースコード 2.3: 環境付き継続の例

```
1  #include <stdio.h>
2
3  __code cg(__code (*ret)(int,void *),void *env){
4    goto ret(1,env);
5  }
6
7  int c_func(){
8    goto cg(_CbC_return,_CbC_environment);
9    return -1;
10 }
11
12 int main(){
13   int test;
14   test = c_func();
15   printf("%d\n",test);
16   return 0;
17 }
```

この例では、 通常 c_func の返り値が-1 である為、 変数 test には-1 が設定されるかの様に見える。しかし関数 c_func 内で CodeGear である cg に軽量継続しており、 cg では環境付き goto を利用して、 1 を返り値として C の関数に戻る。この場合、 呼び出し元 c_func の返り値である -1 の代わりに、 環境付き goto で渡される 1 が優先され、 変数 test には 1 が代入される。

第3章　Perl6

3.1　Perl6の概要

Perl6は2002年にLarryWallが、 Perl5を置き換える言語として設計を開始したプログラミング言語である。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や、 正規表現の表現力の拡大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、 Unixベースの環境で主に利用されている perl はLarryらによって開発されているC言語による実装のみである。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いる。また、 Perl5の様に型が無い様にも振る舞えるが、 静的型付け言語の様に型を付けることや、 型を定義する事が可能である。この型システムの特徴を、漸進的型付けと呼び、 Perl6は漸進的型付け言語である。漸進的型付き言語としては、 他にJavaScriptへトランスコンパイルを行うTypeScriptなどが存在する。

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。Perl6の実際のサンプルコードをソースコード3.1とソースコード3.2に示す。

ソースコード 3.1: Perl6のGrammerを利用したサンプルコード

```
1  grammar Parser {
2      rule TOP { I <love> <lang> }
3      token love { '^^e2^^99^^a5' | love }
4      token lang { < Perl Rust Go Python Ruby > }
5  }
6
7  say Parser.parse: 'I␣^^e2^^99^^a5␣Perl';
8  # OUTPUT: ^^ef^^bd^^a2I ^^e2^^99^^a5 Perl^^ef^^bd^^a3 love => ^^ef^^bd^^a2^^e2
       ^^99^^a5^^ef^^bd^^a3 lang => ^^ef^^bd^^a2Perl^^ef^^bd^^a3
9
10 say Parser.parse: 'I␣love␣Rust';
11 # OUTPUT: ^^ef^^bd^^a2I love Rust^^ef^^bd^^a3 love => ^^ef^^bd^^a2love^^ef^^bd^^
       a3 lang => ^^ef^^bd^^a2Rust^^ef^^bd^^a3
```

ソースコード 3.2: Perl6の型システムを利用したfizzbuzz

```
1  my subset Fizz of Int where * %% 3;
```

```
2  my subset Buzz of Int where * %% 5;
3  my subset FizzBuzz of Int where Fizz&Buzz;
4  my subset OtherNumber of Int where none Fizz|Buzz;
5
6  proto sub fizzbuzz ($) { * }
7  multi sub fizzbuzz (FizzBuzz) { "FuzzBuzz" }
8  multi sub fizzbuzz (Fizz) { "Fizz" }
9  multi sub fizzbuzz (Buzz) { "Buzz" }
10 multi sub fizzbuzz (OtherNumber $number) { $number }
11
12 fizzbuzz($_).say for 1..15;
```

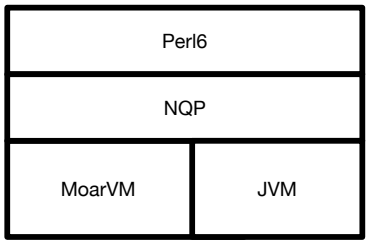
Perl6 は言語的な仕様や、 実装が Perl5 と大幅に異なっており、 言語的な互換性が存在しない。その為、 現在では Perl5 と Perl6 は別言語として開発されており、 Perl6 は主要な処理系である Rakudo から名前を取り、 Raku という別名がついている。

3.2 Rakudo

Rakudo とは NQP によって記述され、 MoarVM、 JVM 上で動作する Perl6 の実装である。NQP とは NotQuitPerl の略であり、 Perl6 のサブセットである。

Rakudo が Perl6 のコンパイラかつインタプリタとして機能する。Rakudo の構成を図 3.1 に示す。

図 3.1: Rakudo の構成図



Perl6 そのものは NQP で大本が記述されており、 その上に Perl6 自身で記述された箇所が存在する。図 3.1 に示すとおり、 MoarVM が解釈するのは NQP が発行した MoarVM バイトコードである。Perl6 のプログラムは Perl6 及び NQP コンパイラによって MoarVM バイトコードに変換され、 MoarVM が評価する。現在は MoarVM の他に JVM も動作環境として選択可能であるが、 JVM 側には MoarVM と比較して実装された機能が少ないなどの特徴がある。

MoarMV そのものは Perl6 や NQP プログラムを直接は評価する事が出来ない。従って、NQP 及び Perl6 で書かれている 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。実際に NQP で記述された Rakudo の実装の一部をソースコード 3.3 に示す。

ソースコード 3.3: Rakudo の実装の一部

```
use Perl6::Grammar;
use Perl6::Actions;
use Perl6::Compiler;

# Initialize Rakudo runtime support.
nqp::p6init();

# Create and configure compiler object.
my $comp := Perl6::Compiler.new();
$comp.language('perl6');
$comp.parsegrammar(Perl6::Grammar);
$comp.parseactions(Perl6::Actions);
$comp.addstage('syntaxcheck', :before<ast>);
$comp.addstage('optimize', :after<ast>);
hll-config($comp.config);
nqp::bindhllsym('perl6', '$COMPILER_CONFIG', $comp.config);
```

3.3 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 独自に作成されたプロセス仮想機械であり、 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、 レジスタマシンである。MoarVM は NQP や Perl6 から与えられた MoarVM バイトコードを評価する。

MoarVM 自体の改良は現在も行われているが、 開発者の多くは新機能の実装などを中心に行っている。速度上昇を目指したプロジェクトも存在はするが、 介入する余地があると考えられる。また、 内部では LuaJit という JIT コンパイル用のライブラリを利用しているが、 JIT に対して開発者チームの力が注がれていない。その為、 本研究では JIT や速度上昇を最終的な目標として考え、 速度上昇までに必要なモジュール化などの実装を行う。

3.4 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 のコード例をソースコード 3.4 に示す。

ソースコード 3.4: フィボナッチ数列を求める NQP のソースコード

```
#! nqp

sub fib($n) {
    $n < 2 ?? $n !! fib($n-1) + fib($n - 2);
}

my $N := 29;

my $t0 := nqp::time_n();
my $z := fib($N);
my $t1 := nqp::time_n();

say("fib($N) = " ~ fib($N));
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ソースコード 3.5 に示す。

ソースコード 3.5: 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) )
        )));
});
```

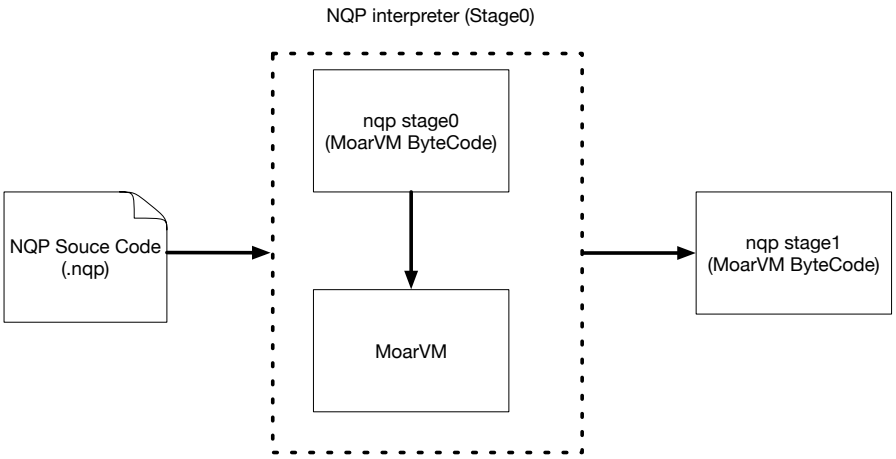
MoarVM を利用する場合、 MoarVM の実行バイナリである moar に対して、 ライブラリパスなどを予め用意した NQP インタプリタのバイトコードに設定する。moar の起動時の設定は、 コマンドライン引数のオプションで与える事が可能である。その為、 既に存在している MoarVM バイトコードで記述された NQP のインプリタファイルを、 適切にオプションで指定し、moar を実行することで NQP のインタプリタが起動する。

NQP のビルドフローの一部を図 3.2 に示す。

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。既に用意されている、 ターゲットの VM のバイトコード化している NQP インタプリタの状態を Stage0 と呼ぶ。Stage0 を利用し、NQP ソースコードからビルドした NQP インタプリタであるバイトコードを、 Stage1 と呼ぶ。

Stage1 を moar の起動時オプションにライブラリとして設定し、 起動した NQP インタプリタで再度ビルドした NQP インタプリタを、 Stage2 と呼ぶ。この 2 度目のビルドで、

図 3.2: NQP Stage1 のビルドフロー



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 Stage2 のバイトコードを利用する。2 度目のビルドの際に生成された Stage2 を利用して、moar を起動するスクリプトの事を小文字の nqp と呼び、これが NQP のインタプリタのコマンドとなる。

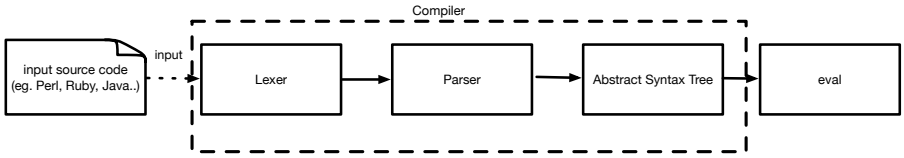
nqp は使用している VM のバイトコードを生成する機能があり、 Rakudo のビルド時にはこの機能を利用してバイトコードを生成する。

第4章　CbCによるMoarVM

4.1　スクリプト言語のバイトコード

プログラミング言語処理系は一般的に、 コンパイラ又はインタプリタに、 対象のソースコードを入力として与える。処理系はソースコード中の各文字列を、 トークンと呼ばれる形式に変換する。トークンは処理系によってはオブジェクトそのものなどに変換される。このトークンに変換するフェーズを字句解析と呼ぶ。変換されたトークンが、 対象のプログラミング言語の文法などに沿っているかどうかの確認を行う。文法に沿っていた場合、 文法に応じてトークンを木構造に変換する。これを構文解析と呼ぶ。構文解析の後は、 素朴なインタプリタ言語と呼ばれる種類のプログラミング言語の場合、 これらを木構造の根から順次実行する。この処理の流れを図 4.1 に示す。直接構文木を実行する場合、 実装そのものは単純になるが、 処理時間などが非常にかかるなどの特徴がある。

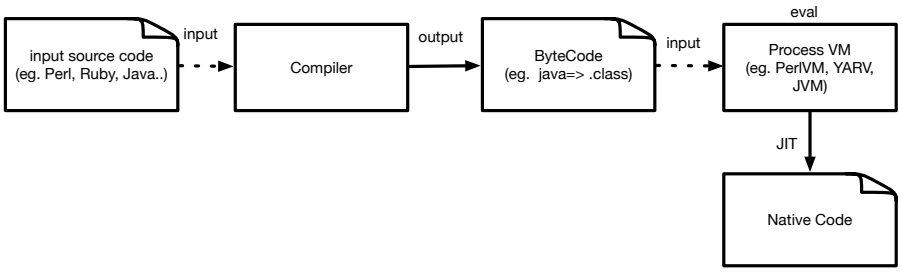
図 4.1: 構文木を直接実行するプログラミング言語の処理の流れ



現在の主流なスクリプト言語は、 一旦変換した構文木をバイトコードと呼ばれるバイナリ形式に変換する。バイトコードを利用する種類のプログラミング言語の、 処理の流れを図 4.2 に示す。この場合、 入力されたソースコードをバイトコードに変換する実装と、 変換されたバイトコードを評価する仮想機械に処理系が分けられる。仮想機械は OS のエミュレータではなく、 プロセス仮想マシンと呼ばれるものである。バイトコードを直接出力できる形式のプログラミング言語に Java、 Java の仮想機械に JVM が存在する。内部的に利用しており直接は出力されない言語に、C 言語で実装された MRI と呼ばれる ruby の実装などがあり、 この仮想機械に YARV が存在する。バイトコードを経由することで、 コンパイルを担当する実装と、 評価を担当する仮想機械の実装に分類する事が可能となり、 それぞれに適した最適化処理が実装可能となる。また実行する際の速度もバイトコードを経由することで上昇する。

Rakudo では Perl6、 NQP がそれぞれ対象の VM のバイトコードを生成し、 そのバイトコードを VM が実行する。バイトコード生成までの処理をフロントエンドと呼び、 バ

図 4.2: バイトコードを使用するプログラミング言語の処理の流れ



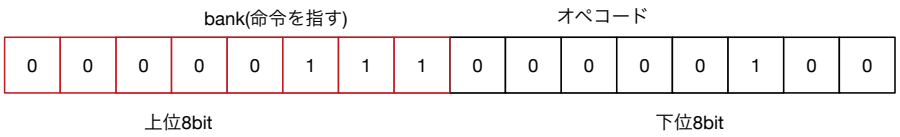
イトコードから評価を行う処理をバックエンドと呼ぶ。これらは Java の様にバイトコードを出力する事も可能であるが、 オプションで指定しない限りは、 ruby などの様に内部的にのみバイトコードを利用する。主に Rakudo で利用されている仮想機械に C 実装の MoarVM があり、 本研究では MoarVM のバイトコード評価部分について検討をする。

4.2 オリジナルの MoarVM の処理

CbC で MoarVM を書き換える際に、 いきなり全てを実装する事は難しい。スクリプト言語の処理系の中心は、 与えられたバイトコードから実際の処理を逐次実装する部分である。その部分をバイトコードインタプリタと呼ぶ。今回は MoarVM の書き換えを検討する為に、 まず MoarVM のバイトコードインタプリタ部分の書き換えを行う。書き換えを行うにあたり、 MoarVM のオリジナルの箇所の実装を確認する。今回対象とする MoarVM のバージョンは 2018.04.01 である。

MoarVM のバイトコードは、 フォーマットが決まっており、 複数の意味のあるバイトコードの集合となっている。今回は、MoarVM の命令を実行する際に必要な、 命令コードに対応するバイトコードの部分に着目する。これは、 今回書き換えを検討する MoarVM バイトコードインタプリタが読み込み、 評価するバイトコードの部分が、 命令コードバイトコードに対応する為である。MoarVM のバイトコードの中の、 命令に対応するバイトコードの構成を図 4.3 に示す。

図 4.3: MoarVM の命令バイトコード



MoarVM の命令バイトコードは 16 ビットである。上位 8 ビットが、 バンクと呼ばれる命令セットの指定となる。MoarVM の命令はバンクの 0 から 127 番までが、 MoarVM の

コア機能となっており、 128 から 255 番までが、 拡張可能な命令セットとなっている。下位 8 ビットは、 バンクの内部の命令指定となっている。命令によっては、 この後のバイトコードで 0 個以上のオペランドを必要とする物がある。オペランドの中にはレジスタの型指定、 引数などが埋め込まれ、 命令バイトコードとの対応のために 16 ビットで表現される。

MoarVM のバイトコードインタプリタは src/core/interp.c 中の関数 MVM_interp_run で定義されている。この関数では MoarVM のバイトコードの中の、 命令に対応するバイトコードを解釈する。関数内では、 解釈するべきバイトコード列が格納されている変数 cur_op や、現在と次の命令を指し示す op、 命令に対して受け渡す現在の VM 情報である ThreadContex tc などが変数として利用されている。実際に命令ディスパッチを行っている箇所の一部をソースコード 4.1 に示す。

ソースコード 4.1: オリジナルの MoarVM の命令ディスパッチ部分

```
1        DISPATCH(NEXT_OP) {
2            OP(no_op):
3                goto NEXT;
4            OP(const_i8):
5            OP(const_i16):
6            OP(const_i32):
7                MVM_exception_throw_adhoc(tc, "const_iX␣NYI");
8            OP(const_i64):
9                GET_REG(cur_op, 0).i64 = MVM_BC_get_I64(cur_op, 2);
10               cur_op += 10;
11               goto NEXT;
12           OP(pushcompsc): {
13               MVMObject * const sc = GET_REG(cur_op, 0).o;
14               if (REPR(sc)->ID != MVM_REPR_ID_SCRef)
15                   MVM_exception_throw_adhoc(tc, "Can␣only␣push␣an␣SCRef␣with␣
                         pushcompsc");
16               if (MVM_is_null(tc, tc->compiling_scs)) {
17                   MVMROOT(tc, sc, {
18                       tc->compiling_scs = MVM_repr_alloc_init(tc, tc->instance->
                             boot_types.BOOTArray);
19                   });
20               }
21               MVM_repr_unshift_o(tc, tc->compiling_scs, sc);
22               cur_op += 2;
23               goto NEXT;
24           }
25       }
```

ソースコード 4.1 中の OP(.*) と書かれている部分が、 それぞれのバイトコードが示す命令名となっている。例えば no_op は、 何もしない命令であるため、 マクロ NEXT を利用しプログラムカウンタ相当の cur_op を進めるのみの処理を行う。また、 登場する DISPATCH や OP 、 NEXT などはそれぞれマクロとして定義されている。これら MoarVM_interp_run 中で、 利用されるマクロの定義を、 ソースコード 4.2 に示す。

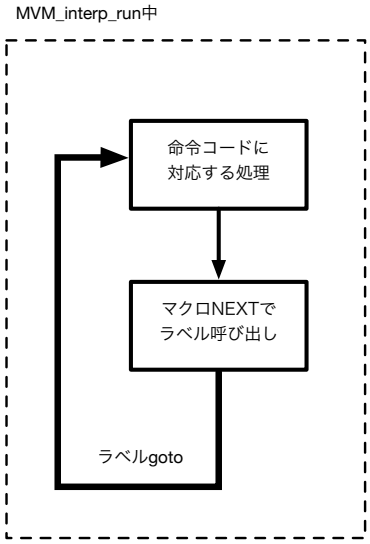
ソースコード 4.2: オリジナルの MoarVM_interp_run で使用されるマクロ

```
1 #define NEXT_OP (op = *(MVMuint16 *)(cur_op), cur_op += 2, op)
2
3 #if MVM_CGOTO
4 #define DISPATCH(op)
5 #define OP(name) OP_ ## name
6 #define NEXT *LABELS[NEXT_OP]
```

```
7  #else
8  #define DISPATCH(op) switch (op)
9  #define OP(name) case MVM_OP_ ## name
10 #define NEXT runloop
11 #endif
```

このマクロの中では、 利用しているCコンパイラがラベルに対してのgotoが利用できる、 コンパイラ拡張を実装している場合はMVM_CGOTOが真となり、 6行目までが実行される。それ以外の場合は8行目以降のマクロ定義となる。ラベルgotoが利用できる場合、 マクロDISPATCHは空白として設定され、 マクロOPは、 それぞれの命令に対応したラベルとなる。この場合の処理の流れを図4.4に示す。

図 4.4: ラベルgotoが利用できる場合のオリジナルのMVM_interp_runの処理の流れ



次の命令に移動する際は、 マクロNEXT_OPを用いてcur_opを次の命令に移動させ、opの値を再設定する。このopが実行すべき命令の番号が格納されている。opを用いて、ソースコード4.3に示す配列LABELSから、 命令に対応するラベルを取得する。LABELSはマクロOPが変換したラベルのリストである。ソースコード4.1の場合、no_opはopが0が代入され、 const_i8は1が設定されている。

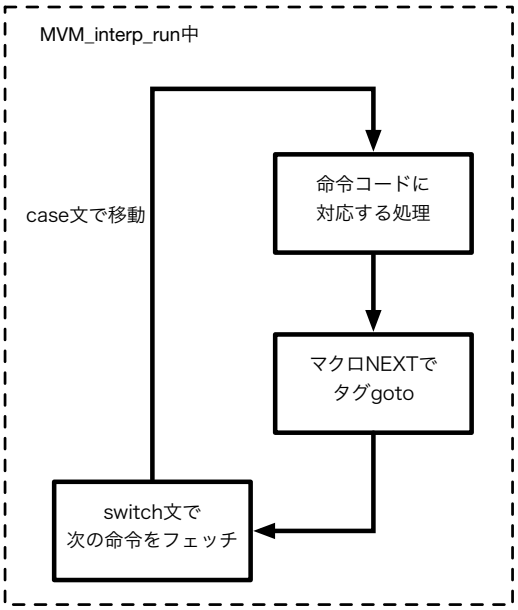
ソースコード 4.3: MoarVMの命令ラベルが設定されている配列

```
1  static const void * const LABELS[] = {
2      &&OP_no_op,
3      &&OP_const_i8,
4      &&OP_const_i16,
5      &&OP_const_i32,
6      &&OP_const_i64,
7      &&OP_const_n32,
8      &&OP_const_n64,
9      &&OP_const_s,
10     &&OP_set,
11     &&OP_extend_u8,
12     &&OP_extend_u16,
```

```
13      &&OP_extend_u32,
14      &&OP_extend_i8,
15      &&OP_extend_i16,
```

ラベル goto が利用できない場合、 マクロ DISPATCH は switch 文に、 OP は case 文にそれぞれ変換される。cur_op は数値そのものである為、 この場合はラベル配列へのアクセスは行われない。case 文に変換された場合の処理の流れを 4.5 に示す。

図 4.5: case 文に展開された場合のオリジナルの MVM_interp_run の処理の流れ



またソースコード 4.1 の中に含まれているマクロ GET_REG は、 ソースコード 4.4 に示す定義がされている。

ソースコード 4.4: レジスタ情報を取得するマクロ GET_REG

```
1 #define GET_REG(pc, idx) reg_base[*((MVMuint16 *)(pc + idx))]
```

配列 reg_base は MoarVM 上で利用される、 MoarVM のレジスタのリストである。このマクロ中の pc は、 MVM_interp_run 上では cur_op となっている。idx は命令ごと個別に設定しており、 例えば const_i64 内で利用されている GET_REG は、 idx の値が 0 に設定されている。これは MoarVM がレジスタ情報を取得する際に、 命令を基本に前後に参照できるレジスタを指定出来る為である。参照しているレジスタ集合の変数 reg_base は、MVM_interp_run 中ではローカル変数として宣言されている。

MoarVM のディスパッチ部分は、case 文に変換される可能性がある。従って、 MoarVM の命令コードに対応する処理は、 C ソースファイルの特定の場所に記述せざるを得ない。この方法の場合、命令コードに対応する処理のファイル分割などのモジュール化が行えず、 1 ファイル辺りの記述量が膨大になってしまう。

4.3 CbCによるMoarVMの実装

interp.c 内の MVM_interp_run では、 命令コードのディスパッチはマクロを利用した cur_op の計算及びラベル goto、 もしくはマクロ DISPATCH による switch-case 文で行っていた。このディスパッチ方法では、 case 文を利用する可能性があるため、 ファイルが冗長になる事や、 モジュール化が出来ないという問題が生じる。

CbC によって書き換えを行った MoarVM である、 CbCMoarVM ではこの問題を解決する為に、 CodeGear の概念を導入する。まず、 MoarVM の命令に対応する CodeGear を作成し、 各 CodeGear の名前を要素として持つ CbC の CodeGear テーブルを作成した。CodeGear のテーブルは、 特定の cbc_next という CodeGear から参照する。cbc_next から命令ごとの CodeGear に遷移し、 命令に対応する処理をした後に、 cbc_next に戻り、 別の命令に対応する CodeGear に遷移を繰り返す。この cbc_next は、 元の MVM_interp_run で使用されているマクロ NEXT を、 CodeGear で書き直したものである。実際に書き直したマクロ及び、 cbc_next をソースコード 4.5 に示す。

ソースコード 4.5: cbc_next 及び CbCMoarVM でのマクロ例

```
1  #define NEXT_OP(i) (i->op = *(MVMuint16 *)(i->cur_op), i->cur_op += 2, i->op)
2
3  #define DISPATCH(op) {goto (CODES[op])(i);}
4  #define OP(name) OP_ ## name
5  #define NEXT(i) CODES[NEXT_OP(i)](i)
6  static int tracing_enabled = 0;
7
8  __code cbc_next(INTERP i){
9      goto NEXT(i);
10 }
```

CodeGear 間の軽量継続を中心に設計している為、 switch case 文を利用するマクロは削除した。また、 各マクロの引数に、 変数 i を導入している。変数 i は、 バイトコードインタプリタ内で利用する、 MoarVM のレジスタ情報などが格納された、 構造体へのポインタである。i が示す構造体 INTER、 呼び i の型である構造体 INTERP は、 ソースコード 4.6 の示すように宣言している。これは、マクロ内部で現在の命令を示す op や、命令列 cur_op にアクセスする必要があるが、 従来のマクロの記述では CbC を利用した場合に、変数にアクセス出来なくなる為に導入している。

ソースコード 4.6: MoarVM の情報を格納した構造体 INTER

```
1  typedef struct interp {
2      MVMuint16 op;
3      /* Points to the place in the bytecode right after the current opcode. */
4      /* See the NEXT_OP macro for making sense of this */
5      MVMuint8 *cur_op;
6
7      /* The current frame's bytecode start. */
8      MVMuint8 *bytecode_start;
9
10     /* Points to the base of the current register set for the frame we
11      * are presently in. */
12     MVMRegister *reg_base;
13
14     /* Points to the current compilation unit. */
```

```
15      MVMCompUnit *cu;
16
17      /* The current call site we're constructing. */
18      MVMCallsite *cur_callsite;
19
20      MVMThreadContext *tc;
21  } INTER,*INTERP;
```

4.4 命令実行箇所の CodeGear への変換

命令実行箇所は、 case 文又はラベル goto で移動した先に記述されている。これらの箇所を、 それぞれ専用の CodeGear に変換することで、 命令の実行を CodeGear の遷移として CbC を利用して実装する。MVM_interp_run では、 ソースコード 4.1 中で示すとおり、 マクロ OP を用いて記述されている。

OP を用いて記述しているそれぞれの命令は、 通常ソースコード 4.3 に示すラベル配列、または switch case 文で遷移する。従来はソースコード 4.2 に示す、 変数 op の値利用してをマクロ NEXT で対象の命令のラベル、 および switch 文に値を引き渡す処理をしていた。CodeGear での実装の際も、 このインターフェイスに揃えて実装する。変数 op の数値は、 ソースコード 4.3 に示すラベル配列の、 命令の登場順と対応している。その為、命令を変換した CodeGear を、 ラベル配列と順序を対応させ、 CodeGear の配列を作成する。順序さえ対応させれば、 CodeGear の名前などは問わない。実際に作成した CodeGear のリストをソースコード 4.7 に示す。変換した CodeGear は、それぞれ CodeGear であることを示す為、 接頭辞として cbc_を付けている。

ソースコード 4.7: CodeGear 配列の一部

```
1  __code (* CODES[])(INTERP) = {
2    cbc_no_op,
3    cbc_const_i8,
4    cbc_const_i16,
5    cbc_const_i32,
6    cbc_const_i64,
7    cbc_const_n32,
8    cbc_const_n64,
9    cbc_const_s,
10   cbc_set,
11   cbc_extend_u8,
12   cbc_extend_u16,
```

変換された各命令に対応する CodeGear の一部を、 ソースコード 4.8 に示す。この CodeGear はソースコード 4.1 と対応している

ソースコード 4.8: CbCMoarVM のバイトコード命令に対応する CodeGear

```
1  __code cbc_no_op(INTERP i){
2      goto cbc_next(i);
3  }
4  __code cbc_const_i8(INTERP i){
5      goto cbc_const_i16(i);
6  }
7  __code cbc_const_i16(INTERP i){
8      goto cbc_const_i32(i);
```

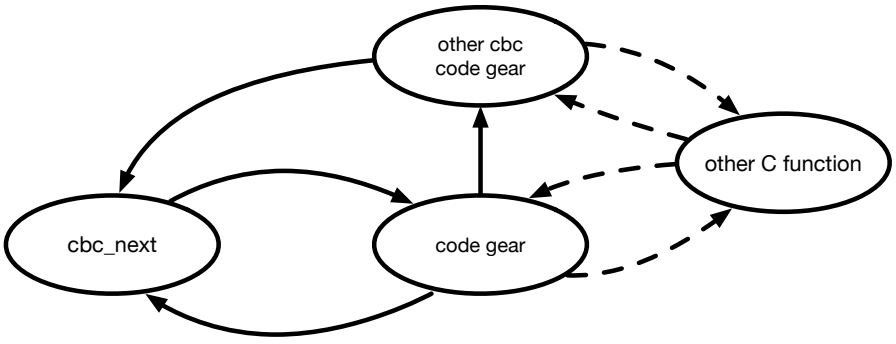
```
}
__code cbc_const_i32(INTERP i){
    MVM_exception_throw_adhoc(i->tc, "const_iX␣NYI");
   goto cbc_const_i64(i);
}
__code cbc_const_i64(INTERP i){
    GET_REG(i->cur_op, 0,i).i64 = MVM_BC_get_I64(i->cur_op, 2);
    i->cur_op += 10;
    goto cbc_next(i);
}
__code cbc_pushcompsc(INTERP i){
    static MVMObject * sc;
    sc = GET_REG(i->cur_op, 0,i).o;
    if (REPR(sc)->ID != MVM_REPR_ID_SCRef)
        MVM_exception_throw_adhoc(i->tc, "Can␣only␣push␣an␣SCRef␣with␣pushcompsc")
            ;
    if (MVM_is_null(i->tc, i->tc->compiling_scs)) {
        MVMROOT(i->tc, sc, {
            i->tc->compiling_scs = MVM_repr_alloc_init(i->tc, i->tc->instance->
                boot_types.BOOTArray);
        });
    }
    MVM_repr_unshift_o(i->tc, i->tc->compiling_scs, sc);
    i->cur_op += 2;
    goto cbc_next(i);
}
```

各 CodeGear は入出力として構造体 INTER のポインタを利用する。これは、各命令の中で使用している cur_op や tc などの変数へのアクセスの為である。通常の MoarVM の場合、MVM_interp_run の関数内でタグ goto や、 switch 文を利用する為に、 MVM_interp_run のローカル変数に各命令処理の中でアクセスする事が可能である。しかし、CbCMoarVM の場合、 MVM_interp_run から軽量継続を利用し、 CodeGear に遷移してしまう為、 これらローカル変数にアクセスできない。

作成した CodeGear のリスト CODES は、 ソースコード 4.5 に示すとおり、 cbc_next という CodeGear のみが取り扱う。cbc_next は、 マクロ NEXT を利用しているが、 マクロ NEXT は CODES にアクセスし、 対象となる CodeGear の名前を取得する。CodeGear を取得後、 引数として i を渡し、 goto 文によって命令ごとの CodeGear に遷移する。

命令ディスパッチに関する CodeGear の状態遷移図を図 4.6 に示す。

図 4.6: CbCMoarVM の命令バイトコードディスパッチの状態遷移



図中の破線部分は通常の C 言語の関数呼び出しが行われる。CodeGear から C の関数呼び出しの返り値を利用することなどは通常の C 言語と同様に可能である。各命令に対応した CodeGear は、 cbc_next から遷移し、 処理を行った後 cbc_next に goto 文で軽量継続する。

また、 変換した命令の中には switch case 文での、 break が発生せず、 次の case 文に移行する命令が存在する。各 CodeGear は 1 命令に閉じてしまっている為、 cbc_next ではなく、 明示的に次に次の CodeGear を指定する必要がある。今回は直接 CodeGear 中に遷移先の CodeGear を goto 文を付けて記述した。ソースコード 4.8 中では、 cbc_const_i8 などが該当する。

命令の数は膨大である為、 全てを手作業で変換するのは望ましくない。本研究では Perl スクリプトを用いて、 interp.c から命令を CodeGear にそれぞれ変換し、 CodeGear のリストを自動的に作成するスクリプトを作成した。このスクリプトは以下の修正手続きを実行する。

- OP(.*) の.*部分を CodeGear の名前として、 先頭に cbc_をつけた上で設定する。
- cur_op など、 構造体 INTER のメンバ変数は、 ポインタ i から参照するように修正する。
- GC 対策のためマクロ MVMROOT を利用している箇所は、 一度ダミーの関数を経由する。
- 末尾の goto NEXT を goto cbc_next(i) に修正する。
- case 文で下の case 文に移行する箇所は、 case 文に対応する CodeGear に遷移する様に goto 文を付け加える。

この中の MVMROOT をソースコード 4.9 に示す。

ソースコード 4.9: MVMROOT の定義

```
1 /* Macros related to rooting objects into the temporaries list, and
2  * unrooting them afterwards. */
3 #define MVMROOT(tc, obj_ref, block) do {\
4     MVM_gc_root_temp_push(tc, (MVMCollectable **)&(obj_ref)); \
5     block \
6     MVM_gc_root_temp_pop(tc); \
7  } while (0)
```

MVMROOT は、 行いたい処理の前後で、MVM_gc_root_temp 系統の push と pop のスタック操作の関数を実行する。これは、 MoarVM が所持しているガベージコレクションに、 行いたい処理の間にオブジェクトが回収されるのを防ぐための処理である。ガベージコレクションの回収を防ぐために、 MVM_gc_root_temp_push では、 大域変数の配列に一時的にオブジェクトのアドレスを入れる。このオブジェクトは、 CodeGear 中のローカル変数であるが、 CodeGear 中のローカル変数は通常別の CodeGear に移動する際に破棄する為、 この様な動きを想定していない。その為、 MVMROOT を呼び出す処理の場合は、 MVMROOT を使う命令を別の関数でラップし、 CodeGear から関数呼び出しの形で命令を呼び出す。

4.5 MoarVMのデバッグ

CbCで書き換えたMoarVMであるCbCMoarVMは、現在gcc、LLVM/clang上に実装しているCbCコンパイラでビルドする事が可能である。また、それぞれO3までの最適化オプションをビルド時に指定してもビルドする事が可能である。

MoarVMの書き換えに伴って、正常にオリジナルのMoarVMと同じ振る舞いをするか確認をしたい。MoarVM自体には現在テストコードが存在しない。MoarVMのリポジトリ内のメッセージには、MoarVM上で動作するNQP、及びRakudoに付随しているテストコードで、MoarVMの実装をテストする事が推奨されている。NQPやRakudoのテストは、ソースコード4.10に示す様なコードを利用する。実際にNQPとPerl6のインタプリタとしてビルドした、nqpやperl6コマンドを実行する。実行時に出力された結果と、期待する結果が一致するかを確認する方式である。

ソースコード 4.10: NQPのテストコードの例

```
1  #! nqp
2
3  # check control structure 'if'
4
5  say('1..6');
6
7  if 1 { say("ok␣1␣#␣on␣one␣line"); }
8
9  say("ok␣2␣#␣statements␣following␣if␣are␣okay");
10
11 if 1 {
12     say("ok␣3␣#␣multi-line␣if");
13 }
14
15 if 0 {
16     print("not␣");
17 }
```

NQPやRakudoのテストを行うには、セルフビルドしたそれぞれのインタプリタであるnqp、perl6を作らなければならない。しかし、これらをビルドする際にはMoarVMの実行バイナリであるmoarを動かす必要がある。nqpなどのビルド時には、入力として与えられたバイトコードを解析し、命令部分をディスパッチするバイトコードインタプリタを使用せざるを得ない。今回はバイトコードディスパッチ部分を書き換えた為、この部分にバグが生じていると、そもそもnqpやperl6を生成する事が出来ない。その為、利用したいnqpやperl6のテストコードを出力を通して確認する事が出来ない。従って、今回はMoarVM自体にバイトコードを入力として与え、期待する動作をするかどうかを独自に確認する必要がある。

4.6 CbCMoarVMのデバッグ

CbCMoarVMが正常にバイトコードを実行しているかどうかは、正常に動くオリジナルのMoarVMと、実行したバイトコードの差分を検知することで確認する事が可能である。入力として与えたスクリプトは常に同じバイトコードに変換されると考えられるが、

MoarVM はバイトコードに UUID の様な物を埋め込んでしまう為、 同じファイルを与えても生成されるバイトコードが異なってしまう。その為、 NQP インタプリタの REPL の様な機能を使い、 それぞれの VM が同じ命令から処理を開始する様に調整する。

差分は、 バイトコードインタプリタの MVM_interp_run のソースコード中で、 次の命令を計算する箇所で、 命令に対応する数値を出力する様に付け加える。その出力を、 gdb などのデバッガと script コマンドなどを用いてログを取り、 perl などのスクリプトを用いて解析する。実際に差分を確認したスクリプトの実行結果の一部を、 ソースコード 4.11 に示す。

ソースコード 4.11: MoarVM と CbCMoarVM の実行命令の差分検知

```
25 : 25 : cbc_unless_i
247 : 247 : cbc_null
54 : 54 : cbc_return_o
140 : 140 : cbc_checkarity
558 : 558 : cbc_paramnamesused
159 : 159 : cbc_getcode
391 : 391 : cbc_decont
127 : 127 : cbc_prepargs
*139 : 162
cbc_invoke_o:cbc_takeclosure
```

左行がオリジナルの MoarVM の実行命令であり、 右行が CbCMoarVM の実行命令である。出力している命令番号は、 それぞれ LABEL や CODES などの命令リストの配列の番号と対応している為、 対応する CodeGear 名を同時に出力している。* が先頭に付随する行で差異が発生しており、 それぞれ実行している命令の番号が異なる事が確認出来る。この例では、 オリジナルの MoarVM は invoke_o 命令を実行しているのに対し、 CbCMoarVM では takeclosure 命令を実行している。

4.7 CbCMoarVM の現在の実装

CbCMoarVM は現在、 Perl6 のサブセットである NQP、 NQP で書かれた Perl6 のビルドに成功している。各言語のインタプリタである nqp、 perl6 共に、 CbCMoarVM の実行バイナリを利用して動作する。

デバッグ時の利便性などから、 現在はオリジナルのバイトコードインタプリタ部分を実行するか、 CbC で記述されたバイトコードインタプリタを実行するかをオプションを通して選択可能となっている。

またそれぞれのテストコードは、 移植元の MoarVM と同等のテスト通過率を示している。

第5章　CbCMoarVMの評価

前章までに、 MoarVM の一部処理を CbC での書き換えを検討した。本章では、 CbC-MoarVM の評価を行う。

5.1　命令処理のモジュール化

MoarVM の命令コードディスパッチ部分は、 当初は case 文やラベル got に変換されるため、 1 ファイルの記述せざるを得なかった。書き換えた CodeGear は、 関数の様に CbC から扱う事が可能である。MoarVM の命令コードディスパッチでは、 命令に対応する数値を利用して、 case 文又は配列から実行するラベルなどを取り出していた。CbCMoarVM では、 CodeGear の集合である配列を用意している。この配列の登録順のみ対応させれば、 CbCMoarVM 内の命令に対応する CodeGear の書く場所は問わなくなる。そのため、命令処理部分を別ファイルに書き出すなどのモジュール化が可能となった。

モジュール化が可能となったことで、 ディスパッチ部分の処理と実際に実行する命令で関数を分離出来た。これにより、 ソースコード上の可読性や、 適切なスコープ管理などがオリジナルの MoarVM と比較し可能となった。

5.2　CbCMoarVMのデバッグ

主要なデバッガである gdb や lldb では、 関数には直接 break point を設定する事が可能である。MoarVM のバイトコードディスパッチ部分は case 文やラベル goto に変換され、関数として扱う事が出来ない。従って、 命令に対応する処理部分で break point を設定を行う場合、 まず処理が書かれている MVM_interp_run 関数に break point を設置する必要がある。プロセス起動後、 関数が書かれている interp.c ファイル中の行番号を指定して break point を付けなければならない。また、ディスパッチでは数値又はラベルを利用している。この事から、 ソースコード 5.1 に示すように、 デバッガ上で直接どの命令を実行するか確認をする事が困難である。

ソースコード 5.1: MoarVM の break point トレース時の表示

```
1 Breakpoint 1, dummy () at src/core/interp.c:46
2 46 }
3 #1 0x00007ffff75608fe in MVM_interp_run (tc=0x604a20,
4     initial_invoke=0x7ffff76c7168 <toplevel_initial_invoke>, invoke_data=0x67ff10
          )
5     at src/core/interp.c:119
```

```
119 goto NEXT;
$1 = 159

Breakpoint 1, dummy () at src/core/interp.c:46
46 }
#1 0x00007ffff75689da in MVM_interp_run (tc=0x604a20,
    initial_invoke=0x7ffff76c7168 <toplevel_initial_invoke>, invoke_data=0x67ff10
        )
    at src/core/interp.c:1169
1169 goto NEXT;
$2 = 162
```

CodeGear は関数として扱う事ができる為、 break point として直接設定する事が可能である。また、gdb などのデバッガは、実行するべき関数名を表示する事が可能であるため、CodeGear の名前も表示可能である。CodeGear 名は命令と対応している為、 CodeGear 名から命令名を推測する事が可能となった。実際にデバッガ上の表示を、 ソースコード 5.2 に示す。

ソースコード 5.2: CbCMoarVM の break point トレース時の表示

```
Breakpoint 2, cbc_next (i=0x7fffffffdc30) at src/core/cbc-interp.cbc:61
61 goto NEXT(i);
$1 = (void (*)(INTERP)) 0x7ffff7566f53 <cbc_takeclosure>
$2 = 162

Breakpoint 2, cbc_next (i=0x7fffffffdc30) at src/core/cbc-interp.cbc:61
61 goto NEXT(i);
$3 = (void (*)(INTERP)) 0x7ffff7565f86 <cbc_checkarity>
$4 = 140

Breakpoint 2, cbc_next (i=0x7fffffffdc30) at src/core/cbc-interp.cbc:61
61 goto NEXT(i);
$5 = (void (*)(INTERP)) 0x7ffff7579d06 <cbc_paramnamesused>
$6 = 558
```

5.3 パフォーマンス

オリジナルのバイトコードインタプリタ、 CbCMoarVM バイトコードインタプリタの両方で速度を測定した。検証には、Perl6 よりも軽量であることから NQP を選択した。

使用する NQP プログラムは、ソースコード 3.4 に示すフィボナッチ数列を求める例題と、 ソースコード 5.3 に示すインクリメントを単純ループで続ける例題を選択した。

ソースコード 5.3: インクリメントを繰り返す NQP のサンプルコード

```
#! nqp

my $count := 100_000_000;

my $i := 0;

while ++$i <= $count {
}
```

複数回計測を行った結果を、 それぞれ表 5.1 と表 5.2 に示す。

表 5.1: フィボナッチ数列を求める例題

MoarVM	CbCMoarVM
1.379sec	1.636sec
1.350sec	1.8043sec
1.346sec	1.787sec

表 5.2: 単純ループを計算する例題

MoarVM	CbCMoarVM
7.499sec	6.135sec
7.844sec	6.362sec
6.746sec	6.074sec

結果から、 再帰呼び出しを多用しているフィボナッチ数列を求める例題では、 オリジナルの VM より低速なパフォーマンスが得られた。CbCMoarVM では現在モジュール化は行っているが、 具体的な速度上昇に対する実装は行っていない。その為、 通常の MoarVM より低速になる事を想定していたが、 表 5.2 に示すように、 単純ループを求める場合は高速化した。これはループ文を利用する際に実行される処理が、 CPU のキャッシュに収まったために高速化したと考えられる。この事から、CbCMoarVM では、 CPU キャッシュに命令を乗せる事が可能であれば、 現在の状況でも MoarVM よりパフォーマンスが向上する利点がある。

5.4 欠点

CbC は C 言語の下位言語であり、 一種のアセンブラの様な言語である。通常の C 言語では、 関数呼び出しで隠蔽されているデータを意識して CodeGear に接続しなければならない。その為、 C 言語で実装を行うより実装で利用する処理や CodeGear の数が増えてしまう。

今回の実装では、 MoarVM の情報をまとめた構造体のポインタを受け渡す実装で行った。別の実装法として、構造体のポインタでなく、 構造体そのものを受け渡す実装に変更し、 挙動を確認した。この構造体のメンバは、 一部メンバの内部で同じメンバを参照している。ポインタの受け渡しでは同期が出来ているが、 構造体そのものを渡してしまうと同期が出来ない。その為、 構造体のメンバを、 ソースコード 5.4 に示すように、 都度代入しなおす必要が生じた。この様に CbC を適応した場合、 データの受け渡しを非常に意識する必要がある。

ソースコード 5.4: CbCMoarVM のデータ同期

```
1  __code cbc_next(INTER i){
2 + i.tc->interp_cur_op = &i.cur_op;
3 + i.tc->interp_bytecode_start = &i.bytecode_start;
4 + i.tc->interp_reg_base = &i.reg_base;
5 + i.tc->interp_cu = &i.cu;
```

```
6       __code (*c)(INTER);
7       c = CODES[NEXT_OP(i)];
```

現在 CbC コンパイラでは、 CodeGear での不要なスタック操作命令を完全に排除できていない為、 CbC の利点が損なわれている箇所が存在する。CbC から C 言語レベルの処理に戻る際は環境付き継続を行う必要があるケースが有り、 この実装を行うためにコードの量や複雑さが高まってしまうとも考えられる。

CbCMoarVM そのものも、 MoarVM の実装に手を加えている為、 MoarVM のアップデートに追従する必要がある。

第6章　今後の課題

本研究では Perl6 の処理系である MoarVM において、命令コードディスパッチ部分を CbC で書き換えた。CbC は C の関数よりも細かな単位を扱えるため、命令コードのモジュール化などが可能となった。今後は MoarVM などの言語処理系に対して、動的に命令コードと対応する CbC のコードを生成し、gcc などの C コンパイラを用いて共有ライブラリの形にコンパイルし、MoarVM と紐付ける JIT などの開発を検討している。また、入力として与えられたソースファイルを解析し、プログラムの入力変数などを記号として表現し、変数の代入などを論理式に変換した記号実行 (symbolick execution) などの手法を検討し、MoarVM 自体の高速化などを通して、CbC の言語処理系への応用を考察する。

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