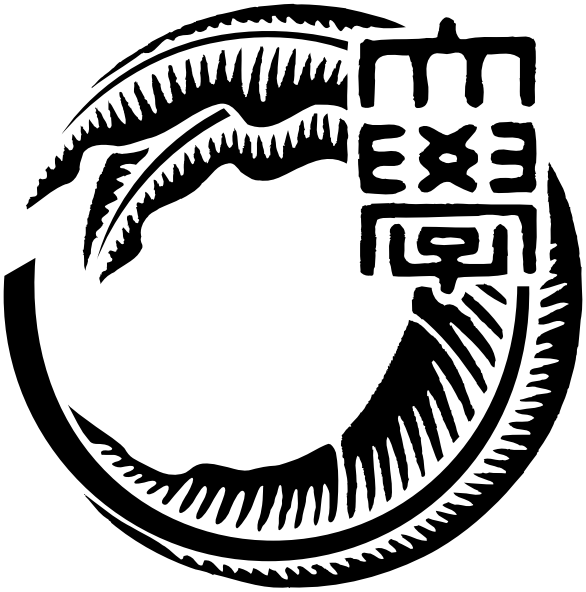
平成30年度 卒業論文

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

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。

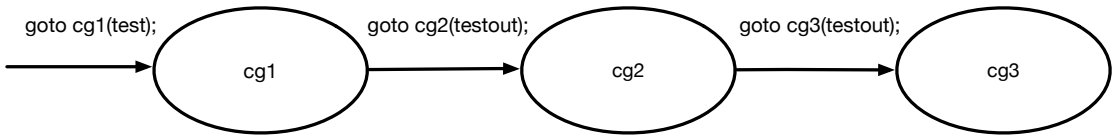
ソースコード 2.1: 加算と文字列を設定する CbC コードの例

```
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 }
```

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。

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。実際にある数の階乗を計算する C プログラムと、 このプログラムを CbC で書き直した場合のソースコードを示す。

図 2.1: ソースコード 2.1 における CodeGear の状態遷移



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 2.2 に示す。

ソースコード 2.2: 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。

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、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ソースコード3.1に示す。

ソースコード 3.1: 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。

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 3.2 に示す。

ソースコード 3.2: 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 3.3 に示す。

ソースコード 3.3: 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。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、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　MoarVMのバイトコード実行

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。

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。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。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などがあり、 内部的に利用しており直接は出力されない言語にruby、 pythonなどがある。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も可能であるが、 基本的にはrubyなどの様に内部的にのみバイトコードを利用する。主に利用されている仮想機械に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

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、 バイトコードに埋め込まれている命令に応じた処理を実行する。

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。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 cur_op が 0 を指し、 const_i8 は 1 を指し示す。

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。

またソースコード 4.1 の

case 文に変換される可能性がある為、 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。その為、 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。

第5章　Context、stub Code Segment の自動生成

Gears OS では 3 章で述べたようにノーマルレベルの計算の他に Context や stub などのメタ計算を記述する必要がある。Gears OS を現在の CbC の機能のみを用いて記述するとこのメタ計算の記述を行わなくてはならず、これには多くの労力を要する。この記述を助けるために Context を生成する generate_context と stub Code Gear を生成する generate_stub を perl スクリプトで作成した。

5.1　stub Code Segment の生成

stub Code Gear は Code Gear 間の継続に挟まれる Code Gear が必要な Data Gear を Context から取り出す処理を行うものである。stub Code Gear は Code Gear 毎に記述する必要があり、その Code Gear の引数を見て取り出す Data Gear を選択する。stub Code Gear を 自動生成することによって Code Gear の記述量を約半分にすることができる。

stub を生成するために generate_stub は指定された cbc ファイルの __code 型である Code Gear を取得し、引数から必要な Data Gear を選択する。generate_stub は引数と interface を照らし合わせ、Gearef または GearImpl を決定する (リスト 5.1)。この時既に stub Code Gear が記述されている Code Gear は無視される。

cbc ファイル (リスト 5.2) から、生成した stub Code Gear を加えて stub を加えたコード (5.3) に変換を行う。

ソースコード 5.1: generate stub

```
1  sub generateStubArgs {
2      my($codeGearName, $varName, $typeName, $typeField, $interface,$output) = @_;
3      my $varname1 = $output?"O_$varName":$varName;
4      for my $n ( @{$dataGearVar{$codeGearName}} ) {
5          # we already have it
6          return 0 if ( $n eq $varname1);
7      }
8      push @{$dataGearVar{$codeGearName}}, $varname1;
9      if ($typeName eq $implementation) {
10         # get implementation
11         $dataGearName{$codeGearName} .= "\t$typeName*␣$varName␣=␣($typeName*)
               GearImpl(context,␣$interface,␣$varName);\n";
12     } else {
13         for my $ivar (keys %{$var{$interface}}) {
14             # input data gear field
```

```
15              if ($varName eq $ivar) {
16                  if ($typeName eq $var{$interface}->{$ivar}) {
17                      if ($output) {
18                          $dataGearName{$codeGearName} .= "\t$typeName**␣O_$varName␣=
                                ␣&Gearef(context,␣$interface)->$varName;\n";
19                          $outputVar{$codeGearName} .= "\t$typeName*␣$varName;\n";
20                          return 1;
21                      }
22
23                      $dataGearName{$codeGearName} .= "\t$typeName*␣$varName␣=␣Gearef
                            (context,␣$interface)->$varName;\n";
24                      return 1;
25                  }
26              }
27          }
28          for my $cName (keys %{$code{$interface}}) {
29              if ($varName eq $cName) {
30                  # continuation field
31                  $dataGearName{$codeGearName} .= "\tenum␣Code␣$varName␣=␣Gearef(
                        context,␣$interface)->$varName;\n";
32                  return 1;
33              }
34          }
35          # global variable case
36          $dataGearName{$codeGearName} .= "\t$typeName*␣$varName␣=␣Gearef(context,␣
                $typeName);\n";
37          return 1;
38      }
39  }
```

ソースコード 5.2: cbc ファイルの例

```
1   #include "../context.h"
2
3   Stack* createSingleLinkedStack(struct Context* context) {
4       struct Stack* stack = new Stack();
5       struct SingleLinkedStack* singleLinkedStack = new SingleLinkedStack();
6       stack->stack = (union Data*)singleLinkedStack;
7       singleLinkedStack->top = NULL;
8       stack->push = C_pushSingleLinkedStack;
9       stack->pop = C_popSingleLinkedStack;
10      stack->get = C_getSingleLinkedStack;
11      stack->isEmpty = C_isEmptySingleLinkedStack;
12      stack->clear = C_clearSingleLinkedStack;
13      return stack;
14  }
15
16  __code clearSingleLinkedStack(struct SingleLinkedStack* stack,__code next(...)) {
17      stack->top = NULL;
18      goto next(...);
19  }
20
21  __code pushSingleLinkedStack(struct SingleLinkedStack* stack,union Data* data,
        __code next(...)) {
22      Element* element = new Element();
23      element->next = stack->top;
24      element->data = data;
25      stack->top = element;
26      goto next(...);
27  }
28
29  __code popSingleLinkedStack(struct SingleLinkedStack* stack, __code next(union
        Data* data, ...)) {
30      if (stack->top) {
```

```
31          data = stack->top->data;
32          stack->top = stack->top->next;
33      } else {
34          data = NULL;
35      }
36      goto next(data, ...);
37  }
38
39  __code getSingleLinkedStack(struct SingleLinkedStack* stack, __code next(union
        Data* data, ...)) {
40      if (stack->top)
41          data = stack->top->data;
42      else
43          data = NULL;
44      goto next(data, ...);
45  }
46
47  __code isEmptySingleLinkedStack(struct SingleLinkedStack* stack, __code next(...)
        , __code whenEmpty(...)) {
48      if (stack->top)
49          goto next(...);
50      else
51          goto whenEmpty(...);
52  }
```

ソースコード 5.3: 生成される stub

```
1   __code clearSingleLinkedStack(struct Context *context,struct SingleLinkedStack*
        stack,enum Code next) {
2       stack->top = NULL;
3       goto meta(context, next);
4   }
5
6   __code clearSingleLinkedStack_stub(struct Context* context) {
7       SingleLinkedStack* stack = (SingleLinkedStack*)GearImpl(context, Stack, stack
            );
8       enum Code next = Gearef(context, Stack)->next;
9       goto clearSingleLinkedStack(context, stack, next);
10  }
```

5.2 Context の生成

Context は Meta Data Gear に相当し、Code Gear や Data Gear を管理している。Data Gear を取得するために generate_context は context の定義 (リスト 5.4) を読み宣言されている Data Gear を取得する。

ソースコード 5.4: context の定義

```
1   struct Context {
2       enum Code next;
3       struct Worker* worker;
4       struct TaskManager* taskManager;
5       int codeNum;
6       __code (**code) (struct Context*);
7       void* heapStart;
8       void* heap;
```

```
 9      long heapLimit;
10      int dataNum;
11      int idgCount; //number of waiting dataGear
12      int odg;
13      int maxOdg;
14      int workerId;
15      union Data **data;
16  };
17
18  union Data {
19      struct Meta {
20          enum DataType type;
21          long size;
22          struct Queue* wait; // tasks waiting this dataGear
23      } meta;
24      struct Task {
25          enum Code code;
26          struct Queue* dataGears;
27          int idsCount;
28      } Task;
29      // Stack Interface
30      struct Stack {
31          union Data* stack;
32          union Data* data;
33          union Data* data1;
34          enum Code whenEmpty;
35          enum Code clear;
36          enum Code push;
37          enum Code pop;
38          enum Code isEmpty;
39          enum Code get;
40          enum Code next;
41      } Stack;
42      // Stack implementations
43      struct SingleLinkedStack {
44          struct Element* top;
45      } SingleLinkedStack;
46      struct Element {
47          union Data* data;
48          struct Element* next;
49      } Element;
50      struct Node {
51          int key; // comparable data segment
52          union Data* value;
53          struct Node* left;
54          struct Node* right;
55          // need to balancing
56          enum Color {
57              Red,
58              Black,
59          } color;
60      } Node;
61  }; // union Data end this is necessary for context generator
```

Code Gear の取得は指定された stub を加えたコードから __code 型を見て行う。取得した Code/Data Gear の enum の定義は enumCode.h、enumData.h に生成される。

Code/Data Gear の名前とポインタの対応は generate_context によって生成される enum Code、enum Data と関数ポインタによって表現される。実際に Code/Data Gear に接続する際は enum Code、enum Data を指定することで接続を行う。

また、generate_context は取得した Code/Data Gear から Context の生成を行うコード (リスト 5.5) も生成する。

Context には Allocation 等で生成した Data Gear へのポインタが格納されている。Code Gear は Context を通して Data Gear へアクセスする。Data Gear の Allocation を行うコードは dataGearInit.c に生成される。

Data Gear は union Data とその中の struct によって表現される。Context には Data Gear の Data Type の情報が格納されている。この情報から確保される Data Gear のサイズなどを決定する。

ソースコード 5.5: 生成された context

```
1
2  #include <stdlib.h>
3
4  #include "../context.h"
5
6  void initContext(struct Context* context) {
7      context->heapLimit = sizeof(union Data)*ALLOCATE_SIZE;
8      context->code = (__code(**) (struct Context*)) NEWN(ALLOCATE_SIZE, void*);
9      context->data = NEWN(ALLOCATE_SIZE, union Data*);
10     context->heapStart = NEWN(context->heapLimit, char);
11     context->heap = context->heapStart;
12
13     context->code[C_clearSingleLinkedStack] = clearSingleLinkedStack_stub;
14     context->code[C_exit_code] = exit_code_stub;
15     context->code[C_getSingleLinkedStack] = getSingleLinkedStack_stub;
16     context->code[C_isEmptySingleLinkedStack] = isEmptySingleLinkedStack_stub;
17     context->code[C_popSingleLinkedStack] = popSingleLinkedStack_stub;
18     context->code[C_pushSingleLinkedStack] = pushSingleLinkedStack_stub;
19     context->code[C_stack_test1] = stack_test1_stub;
20     context->code[C_stack_test2] = stack_test2_stub;
21     context->code[C_stack_test3] = stack_test3_stub;
22     context->code[C_stack_test4] = stack_test4_stub;
23     context->code[C_start_code] = start_code_stub;
24
25 #include "dataGearInit.c"
26
27 }
28
29 __code meta(struct Context* context, enum Code next) {
30     // printf("meta %d\n",next);
31     goto (context->code[next])(context);
32 }
33
34 __code start_code(struct Context* context) {
35     goto meta(context, context->next);
36 }
37
38 __code start_code_stub(struct Context* context) {
39     goto start_code(context);
40 }
41
42 __code exit_code(struct Context* context) {
43     free(context->code);
44     free(context->data);
45     free(context->heapStart);
46     goto exit(0);
47 }
48
49 __code exit_code_stub(struct Context* context) {
50     goto exit_code(context);
51 }
52
```

```
53 // end context_c
```

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、及び CbC の言語処理系への追求を行う。

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