CbCを用いたPerl6処理系

清水 隆博[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

1. はじめに

当研究室では Continuation Based C(以下 CbC) という言語を開発している．CbC は C よりきめ細やかな単位で実装する事が可能である為, 言語処理系に応用すれば効率的な開発，実行が出来ると期待される．現在活発に開発が進んでいる言語に Perl6 がある．Perl6 は MoarVM と呼ばれる VM を中心とした Rakudo と呼ばれる実装が現在の主流となっている．Rakudo は処理速度が他のプログラミング言語と比較しても非常に低速である．その為, 現在日本国内では Perl6 を実務として利用するケースは概ね存在しない．Perl6 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．その為本稿では, CbC を用いた言語処理系の実装の一例として MoarVM を CbC で書き換えた CbCMoarVM を提案する．

CbC を MoarVM の実装として利用した場合, CbC の持つ機能によって MoarVM の高速化を中心とした改良に有益な効果があると推測出来る．また, 現在までの CbC を用いた研究においては言語処理系への応用例が少ない．従って, 本稿は CbC をスクリプト言語の実装に適応した場合, どのような利点やプログラミング上の問題点に遭遇するか, CbC の応用としての側面でも行う．この際に CbC を用いた言語処理系のデバッグを行う際には, CbC を使わずに記述されたオリジナルの言語処理系との並列デバッグが必要となる．従って MoarVM に CbC を適応した場合, どのようにすれば並列デバッグが行えるかについても述べる．本稿ではまず CbC, Perl6 の特徴及び現在の実装に

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

ついて述べ, CbC で書き換えた MoarVM についてデバッグ手法も含め解説する. 研究にあたり, 得られた CbC を言語処理系に適応した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.

2. CbC

2.1 CbC の概要

CbC は当研究室で開発しているプログラミング言語である. C レベルでのプログラミングを行う場合, 本来プログラマが行いたい処理の他に malloc などを利用したメモリのアロケートやエラーハンドリングなどを記述する必要がある. これらの処理を meta computation と呼ぶ. これら meta computation と通常の処理を分離することでバグの原因が meta computation 側にあるか処理側にあるかの分離などが可能となる. しかし C 言語などを用いたプログラミングで meta computation の分離を行おうとすると, それぞれ事細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 という単位を導入した. また CodeGear の実行に必要なデータを DataGear という単位で受け渡す. CbC では CodeGear, DataGear を基本単位として記述するプログラミングスタイルを取る.

2.2 CodeGear と DataGear

CbC では C の関数の代わりに CodeGear を導入する. CodeGear は C の関数宣言の型名の代わりに__code と書くことで宣言できる. __code は CbC コンパイラの扱いは void と同じ型であるが, CbC プログラミングでは CodeGear である事を示す識別子としての意味で利用する. CodeGear 間の移動は goto 文によって記述する.

```
extern int printf(const char*,...);

int main (){
    int data = 0;
    goto cg1(&data);
}

__code cg1(int *datap){
    (*datap)++;
    goto cg2(datap);
}

__code cg2(int *datap){
    (*datap)++;
    printf("%d\n",*datap);
}
```

Code 1: cbc_example.cbc

Code1 に示す CbC のコードでは main 関数から cg1, cg2 に遷移し, 最終的に data の値が 2 となる. CodeGear 間の入出力の受け渡しは引数を利用し行う.

ある CodeGear の実行に必要なデータを, DataGear と呼ぶ. DataGear には CodeGear で実行される関数や変数などの情報を含む. Code1 に示す例では, CodeGear に渡す引数 datap が, 一種の DataGear と言える.

2.3 軽量継続

CbC では次の CodeGear に移行する際, C の goto 文を利用する. 通常の C の関数呼び出しの場合, スタックポインタを操作しローカル変数などをスタックに保存する. CbC の場合スタックフレームを操作せず, レジスタの値を変更せずそのまま次の CodeGear に遷移する事が可能である. 通常 Secheme の call/cc などの継続は現在の位置までの情報を環境として所持した状態で遷移する. 対して CbC は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こと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レジスタレベルでのきめ細やかな実装が可能となっている.

2.4 現在の実装

CbC は現在主要な C コンパイラである gcc 及び llvm をバックエンドとした clang 上の 2 種類の実装が存在する. gcc はバージョン 9.0.0 に, clang は 7.0.0 に対応している.

2.5 CbC と C 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 であるかどうかを判断する. この際に

CodeGear を利用していない場合は通常の C プログラムとして動作する．その為今回検証する 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 MoarVM の 2 種類を同一の CbC コンパイラでビルドする事が可能である．

また C から CbC への遷移時に, 再び C の関数に戻るように実装したい場合がある．その際は環境付き 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．

```
__code cg(__code (*ret)(int,void *),void *env
    ){
  goto ret(1,env);
}

int c_func(){
  goto cg(_CbC_return,_CbC_environment);
  return -1;
}

int main(){
  int test;
  test = c_func();
  printf("%d\n",test);
  return 0;
}
```

Code 2: 環境付き継続の例

Code2 に示す例では c_func から環境付き継続で cg に継続している．通常 c_func の返り値は-1 であるが, cg から環境付き継続で main に帰る為に cg から渡される 1 が test の値となる．

2.6 言語処理系における CbC の応用

CbC を言語処理系, 特にスクリプト言語に応用すると幾つかの箇所に置いて利点がある．CodeGear はコンパイラの基本ブロックに相当する．その為従来のスクリプト言語では主に case 文で記述していた命令コードディスパッチの箇所を CodeGear の遷移として記述する事が可能である．通常の言語処理系では命令コードディスパッチ部分は巨大な case 文となり, この部分を実装した C ファイルが巨大化してしまう．CodeGear を導入することで巨大な case 文を CodeGear として分割する事が可能となり, 処理のモジュール化が可能となる．また, CodeGear と CodeGear 間の遷移は軽量継続で行われる為, レジスタレベルでの最適化も可能となる．

CbC は状態を単位として記述が可能であるため, 命令コードなどにおける状態を利用するスクリプト言語の実装は応用例として適していると考えられる．

3. Perl6 の概要

この章では現在までの Perl6 の遍歴及び Perl6 の言語的な特徴について記載する．

3.1 Perl6 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．

3.2 Rakudo

Rakudo とは NQP, NQP に基づく Perl6 を基に

したプロジェクトである．NQP とは, 以前の Perl6 処理系である Parrot[4] で, 構想に上がった 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であると考えても良い．Rakudo は図 1 に示す構成になっている．Rakudo におけるコンパイラとは厳密には 2 種類存在する．まず第 1 のものが Perl6, もしくは NQP を MoarVM, JVM のバイトコードに変換する NQP コンパイラである．次にその NQP が出力したバイトコードをネイティブコードに変換する VM の 2 種類である．この VM は現在 MoarVM, JavaVM を選択可能である．Rakudo 及び NQP project ではこの NQP コンパイラの部分をフロントエンド, VM の部分をバックエンド [13] と呼称している．NQP で主に書かれ, MoarVM など NQP が動作する環境で動く Perl6 のことを Rakudo と呼ぶ．Perl6 は NQP 以外にも NQP を拡張した Perl6 自身で書かれている箇所が存在し, これは NQP コンパイラ側で MoarVM が解釈可能な形へ変換を行う．

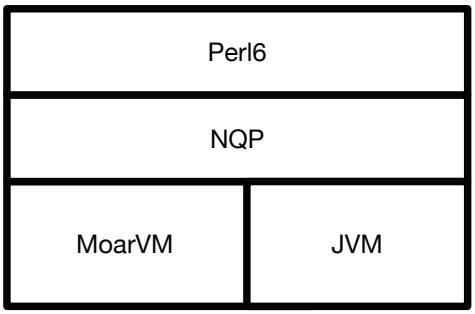

図 1: Rakudo の構成

3.3 MoarVM

MoarVM とは Rakudo で主に開発が進められている VM である．Perl6 及び NQP の専用処理系であり, レジスタマシンである．MoarVM は NQP から与えられた MoarVM のバイトコードを実行する．

MoarVM のバイトコードインタプリタは src/core/interp.c で定義されている．この中の関数 MVM_interp_run で命令に応じた処理を実行する．関数内では命令列が保存されている cur_op, 現在と次の命令を指し示す op, Thread の環境が保存されている Threadcontext などの変数を利用する．命令実行は大きく二種類の動作があり, C の goto が利用できる場合は Code3 に示す MVM_CGOTO フラグが立ちラベル遷移を利用する．goto 文が利用できない場合は巨大な case 文として命令を実行する．

ラベル遷移を利用する場合は Code4 に示すラベルテーブル LABELS にアクセスし, テーブルに登録されているアドレスを取得し, マクロ NEXT で遷移する．Code5 に示す no_op は何もせず次の命令に移動する為, goto NEXT; のみ記述されている．

```
#define NEXT_OP (op = *(MVMuint16 *)(cur_op),
      cur_op += 2, op)

#if MVM_CGOTO
#define DISPATCH(op)
#define OP(name) OP_ ## name
#define NEXT *LABELS[NEXT_OP]
#else
#define DISPATCH(op) switch (op)
#define OP(name) case MVM_OP_ ## name
#define NEXT runloop
#endif
```

Code 3: interp.c のマクロ部分

```
static const void * const LABELS[] = {
    &&OP_no_op,
    &&OP_const_i8,
    &&OP_const_i16,
    &&OP_const_i32,
    &&OP_const_i64,
    &&OP_const_n32,
    &&OP_const_n64,
    &&OP_const_s,
    &&OP_set,
    &&OP_extend_u8,
    &&OP_extend_u16,
    &&OP_extend_u32,
    &&OP_extend_i8,
    &&OP_extend_i16,
```

Code 4: ラベルテーブルの一部分

```
       DISPATCH(NEXT_OP) {
          OP(no_op):
```

```
            goto NEXT;
        OP(const_i8):
        OP(const_i16):
        OP(const_i32):
            MVM_exception_throw_adhoc(tc, "
                const_iX␣NYI");
        OP(const_i64):
            GET_REG(cur_op, 0).i64 =
                MVM_BC_get_I64(cur_op, 2);
            cur_op += 10;
            goto NEXT;
        OP(pushcompsc): {
            MVMObject * const sc = GET_REG(
                cur_op, 0).o;
            if (REPR(sc)->ID !=
                MVM_REPR_ID_SCRef)
                MVM_exception_throw_adhoc(
                    tc, "Can␣only␣push␣an␣
                    SCRef␣with␣pushcompsc")
                    ;
            if (MVM_is_null(tc, tc->
                compiling_scs)) {
                MVMROOT(tc, sc, {
                    tc->compiling_scs =
                        MVM_repr_alloc_init
                        (tc, tc->instance->
                        boot_types.
                        BOOTArray);
                });
            }
            MVM_repr_unshift_o(tc, tc->
                compiling_scs, sc);
            cur_op += 2;
            goto NEXT;
        }
    }
```

Code 5: オリジナル版 MoarVM のバイトコードディスパッチ

この為 MoarVM 内の命令コードに対応する処理は, 命令ディスパッチが書かれている C ソースファイルの, 特定の場所のみに記述せざるを得ない．その為命令コードのモジュール化などが行えず, 1 ファイル辺りの記述量が膨大になってしまう．また各命令コードに対応する処理は, ラベルジャンプもしくは switch 文に展開されてしまう為, Threaded Code の実装を考えた場合, 大幅なコードの改修が要求される．デバッグ時には, C レベルでのデバッグ時にはアドレスと実際に呼ばれる箇所を確認する事に手間がかかる．

3.4 NQP

Rakudo における NQP[6] 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．

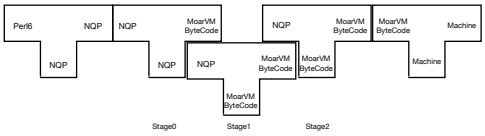

図 2: NQP のビルドフロー

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[10] 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 [7] に記述されているものである．

3.5 処理速度

現在の Perl6 が他のプログラミング言語と比較した場合どのような違いがでるのか計測した．macOS の/var/log/system.log ファイルから正規表現でログ中のプログラムが書き込んだ回数を個別に数え上げるというものである．今回はファイルを 231K と 3GB の二種類用意し, どの様な違いが出るのか測定した．

測定した環境は次の通りである．今回は現在広く使用されているスクリプト言語である Perl5 を計測対象に追加した．また Rakudo の処理系による処理時間の差を計測する為に MoarVM, JVM に構築した Perl6 の処理速度を計測を行った．JVM 自体の処理時間と Rakudo を構築した JVM の速度の差を見るために, 同様のプログラムを Java10 でも行った．

- Perl6 (MoarVM) ver.2018.04.01
- Perl6 (JVM) 2018.06-163-g612d071b8 built on JVM
- Java 10
- Perl5

測定した結果を表 1 に示す． 測定結果の単位は秒である．

FileSize	MoarVM	Perl6 on JVM	Java	Perl5
231K	0.86	21.48	0.27	0.04
3G	2331.08	1665.56	48.85	41.35

表 1: ログファイル処理時間の計測結果

計測結果からファイルサイズが小さい場合は MoarVM より JVM に乗せた Perl6 が低速であるが, ファイルサイズが大きい場合は Java の JIT が働くため MoarVM より高速に動いていると推測できる．

4. CbC による MoarVM

この章では改良を行った Perl6 処理系である MoarVM について述べる．今回改良を行った MoarVM は 2018.04.01 であり, 利用した NQP は 2018.04-3-g45ab6e3 バージョンである．

4.1 方針

MoarVM の中心は, バイトコードを解釈する, バイトコードインタプリタ部分である．その為 CbC を用いて, MoarVM のバイトコードインタプリタ部分を記述し直し, CbCMoarVM として実装する．CbC の CodeGear はコンパイラの基本ブロックに該当する．従って MoarVM における基本ブロックの箇所を CodeGear に書き換える事が可能である．

4.2 MoarVM のバイトコードのディスパッチ

interp.c 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 の計算及びラベルの遷移, もしくはマクロ DISPATCH が展開する switch 文で行われていた．このディスパッチ方法では, ラベルジャンプや巨大な case 文として記述する必要があり, ファイルが冗長になるなどの問題が生じる．

CbCMoarVM ではこの問題を解決するために, それぞれの命令に対応する CodeGear を作成し, 各 CodeGear の名前を要素として持つ CbC の CodeGear のテーブルを作成した．この CodeGear のテーブルを参照する CodeGear は cbc_next であり, この中のマクロ NEXT は interp.c のマクロ NEXT を CbC 用に書き直したものである．

```
#define NEXT_OP(i) (i->op = *(MVMuint16 *)(i
    ->cur_op), i->cur_op += 2, i->op)

#define DISPATCH(op) {goto (CODES[op])(i);}
#define OP(name) OP_ ## name
#define NEXT(i) CODES[NEXT_OP(i)](i)
static int tracing_enabled = 0;
```

```
__code cbc_next(INTERP i){
    goto NEXT(i);
}
```

Code 6: CbCMoarVM のバイトコードディスパッチ

Code6 に示す変更例では, マクロ NEXT などの引数に変数 i を導入している．この i とは, 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, 及び i の型であるポインタ INTERP は Code7 に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 などにアクセスする必要があるが, CbC の CodeGear を適応した場合に元のマクロの記述方法ではアクセスできない為に導入したものである．

```
typedef struct interp {
    MVMuint16 op;
    /* Points to the place in the bytecode
         right after the current opcode. */
    /* See the NEXT_OP macro for making sense
          of this */
    MVMuint8 *cur_op;

    /* The current frame's bytecode start. */
    MVMuint8 *bytecode_start;

    /* Points to the base of the current
         register set for the frame we
     * are presently in. */
    MVMRegister *reg_base;

    /* Points to the current compilation unit
         . */
    MVMCompUnit *cu;

    /* The current call site we're
         constructing. */
    MVMCallsite *cur_callsite;

    MVMThreadContext *tc;
} INTER,*INTERP;
```

Code 7: MoarVM の情報を格納した構造体 INTER

4.3 命令実行箇所の CodeGear への変換

ラベルテーブルや case 文の switch 相当の命令実行箇所を CbC に変換し, CodeGear の遷移として利用する．interp.c は Code5 に示す様にマクロ OP を利用して記述されている．

OP(.*) の.*に該当する箇所はバイトコードの名前である．通常このブロックには LABEL から遷移, または swtich-case によって分岐する為, バイトコードの名前は配列 LABELS の添字に変換されている．そのため対象となる CodeGear を LABELS の並びと対応させ, Code8 に示す CodeGear の配列 CODES として設定すれば CodeGear の名前は問わない．今回は CodeGear である事を示す為に接頭辞として cbc_をつける．

```
__code (* CODES[])(INTERP) = {
  cbc_no_op,
  cbc_const_i8,
  cbc_const_i16,
  cbc_const_i32,
  cbc_const_i64,
  cbc_const_n32,
  cbc_const_n64,
  cbc_const_s,
  cbc_set,
  cbc_extend_u8,
  cbc_extend_u16,
```

Code 8: CodeGear 配列の一部分

Code9 に示す命令の実行処理で MoarVM のレジスタである reg_base や, 命令列 cur_op などの情報を利用しているが, これらは MVM_interp_run 内のローカル変数として利用している．ラベルを利用しているオリジナル版では同一関数内であるためアクセス可能であるが, CodeGear 間の移動で命令を表現する CbC ではアクセスできない．その為 Code7 に示す様に, インタプリタの情報を集約した構造体 inter を定義する．この構造体へのポインタである INTERP 型の変数 i を CodeGear の入出力として与える．CodeGear 内では INTERP を経由することでインタプリタの各種情報にアクセスする．CodeGear 間の遷移ではレジスタの値の調整は行われない為, 入力引数を使ってレジスタマッピングを管理できる．INTERP のメンバである MoarVM のレジスタそのものをアーキテクチャのレジスタ上に乗せる事が可能となる．

命令実行中の CodeGear の遷移を図 3 に示す．

この中で実線で書かれている部分は CbC の goto 文で遷移し, 波線の箇所は通常の C の関数呼び出しとなっている.

現在の CbCMoarVM は次の命令セットのディスパッチを cbc_next が行っている. cbc_next から命令コードに対応する CodeGear に継続し, CodeGear から cbc_next に継続するサイクルが基本の流れである. CodeGear 内から C の関数は問題なく呼ぶ事が可能であるため, C の関数を利用する処理は変更せず記述する事ができる. また変換対象は switch 文であるため, break せず次の case に移行した場合に対応するように別の CodeGear に継続し, その後 cbc_next に継続するパターンも存在する.

```
__code cbc_no_op(INTERP i){
    goto cbc_next(i);
}
__code cbc_const_i8(INTERP i){
    goto cbc_const_i16(i);
}
__code cbc_const_i16(INTERP i){
    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);
}
```

Code 9: CbCMoarVM のバイトコードに対応する CodeGear

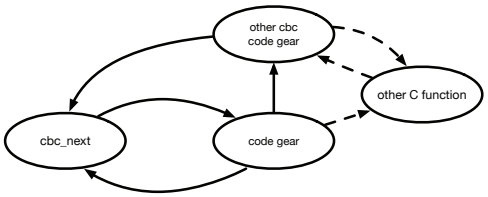

図 3: CbC における MoarVM バイトコードインタプリタ内の状態遷移

バイトコードの数は膨大である為, すべてを手作業で変換する事は望ましくない. 従って PerlScript を用いて interp.c から CbC の CodeGear を自動生成するスクリプトを作成した. このスクリプトで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 を使い局所変数のポインタをスタックに積む箇所は, 局所変数を static 化する
- 末尾の goto NEXT を goto cbc_next(i) に修正する
- case 文で下の case 文に落ちている箇所は, case 文に対応する CodeGear に遷移する様に goto 文を付け加える

上記 Code9 では cbc_const_i8 などが case 文の下の case 部分に該当する cbc_const_i64 に遷移する様に変換されている. また cbc_pushcompsc では MVMROOT に局所変数 sc を渡している為, これを static で宣言し直している.

現在 CbC で記述された OS である GearsOS には Interface が導入されている. これは Java の interface, Haskell の型クラスに該当する概念であり,

次の CodeGear に Interface 経由で継続する事が可能である．Interface は現在の CbCMoarVM の実装には用いていないが, 今後 ThreadedCode の実装を行うにあたり命令コードディスパッチ箇所に導入を検討している．

5. MoarVM のデバッグ

MoarVM 自体のデバッグは MoarVM のリポジトリにテストコードが付随していない為単体では実行不可能である．また, CbC を用いて言語処理系の改良時を行う際に, 処理系のデバッグを行う場合は, CbC を用いないオリジナルの処理系との並列デバッグが必要となる．MoarVM 自体にはデバッグを支援するツールが存在しない為, MoarVM 自体のデバッグ方法や, CbC を用いた処理系との並列デバッグについて独自の手法を考案する必要がある．本稿では MoarVM のデバッグにおける C デバッガの使用方法と MoarVM のテスト方法についても示す．

5.1 MoarVM のバイトコードのデバッグ

MoarVM の実行バイナリである moar に対して, MoarVM のバイトコードを dump オプションを付けて読み込ませると, バイトコードが MoarVM によるアセンブリコードとして出力される．しかしこれは MoarVM が実行したバイトコードのトレースではなく, MoarVM のバイトコードを変換したものに過ぎない．また, 明らかに異なる挙動を示すオリジナルの MoarVM と, CbC で書き換えた CbCMoarVM 両者の moar を利用しても同じ結果が返ってきてしまう．そのため今回の MoarVM のバイトコードインタプリタの実装のデバッグにはこの方法は適さない．従って実際に実行した命令を確認するには gdb などの C デバッガを利用して MoarVM を直接トレースする必要がある．

CbC 側は Code10 に示す様に cbc_next に break point を設定する．オリジナル側は次のオペコードの設定のマクロにダミーの関数を呼び出すように修正し, そこに break point を設定する．CbC 側では CodeGear の名前をデバッガ上で直接確認できるが, オリジナル版は LABLE の配列の添え字から自分でどのオペコードに対応しているかをデバッガの外で探す必要がある．

添字を確認するためには Code11 に示すようにオリジナルの MoarVM の場合 cur_op の値を MVMuint16 のポインタでキャストし, これが指す値を出力する．break point を掛けているダミー関数では cur_op にアクセスする事が出来ない為, スタックフレームを一つ up する必要がある．

```
(gdb) b cbc_next
Breakpoint 2 at 0x7ffff7560288: file src/core
    /cbc-interp.cbc, line 61.
(gdb) command 2
Type commands for breakpoint(s) 2, one per
    line.
End with a line saying just "end".
>p CODES[*(MVMuint16 *)i->cur_op]
>p *(MVMuint16 *)i->cur_op
>c
>end
```

Code 10: CbCMoarVM に対しての break point 設定

```
dalmore gdb --args ../../MoarVM_Original/
    MoarVM/moar --libpath=src/vm/moar/stage0
    gen/moar/stage1/nqp
(gdb) b dummy
Function "dummy" not defined.
Make breakpoint pending on future shared
    library load? (y or [n]) y
Breakpoint 1 (dummy) pending.
(gdb) command 1
Type commands for breakpoint(s) 1, one per
    line.
End with a line saying just "end".
>up
>p *(MVMuint16 *)(cur_op)
>c
>end
```

Code 11: オリジナル版 MoarVM に対しての break point 設定

5.2 MoarVM の並列デバッグ手法

しかし MoarVM が実行する命令は膨大な数がある．その為 gdb などの C デバッガで, オリジナルの MoarVM と, 一部を CbC で記述した CbCMoarVM の並列デバッグを手動で全て行うことは困難である．Perl などのスクリプトを用いて自

動的に解析したいため, ログを残す為に script コマンドを実行した状態で gdb を起動する．トレースでは実行した命令名のみ取得できれば良い為, Code10, 11 で break point に command として設定している様に，設定された cur_op の値を出力し続けるのみの動きを導入する．

実際に実行したログ・ファイルの一部をそれぞれ Code12, 13 に示す．

```
Breakpoint 1, dummy () at src/core/interp.c
    :46
46 }
#1 0x00007ffff75608fe in MVM_interp_run (tc=0
    x604a20,
   initial_invoke=0x7ffff76c7168 <
       toplevel_initial_invoke>, invoke_data
       =0x67ff10)
   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=0
    x604a20,
   initial_invoke=0x7ffff76c7168 <
       toplevel_initial_invoke>, invoke_data
       =0x67ff10)
   at src/core/interp.c:1169
1169 goto NEXT;
$2 = 162
```

Code 12: オリジナル版 MoarVM のバイトコードのトレース

```
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
```

Code 13: CbCMoarVM のバイトコードのトレース

オリジナル版では実際に実行する命令処理はラベルに変換されてしまう為名前をデバッガ上では出力できないが, CbC では出力する事が可能である．CbC とオリジナルの CODES, LABEL の添字は対応している為, ログの解析を行う際はそれぞれの添字を抽出し違いが発生している箇所を探索する．これらは script コマンドが作成したログを元に異なる箇所を発見するスクリプトを用意し自動化する．(Code 14)

```
131 : 131
139 : 139
140 : 140
144 : 144
558 : 558
391 : 391
749 : 749
53 : 53
*54 : 8
```

Code 14: バイトコードの差分検知の一部分

違いが生じている箇所が発見できた場合, その前後の CodeGear 及びディスパッチ部分に break point をかけ, それぞれの変数の挙動を比較する．主に cbc_return 系の命令が実行されている場合は, その直前で命令を切り替える cbc_invoke 系統の命令が呼ばれているが, この周辺で何かしらの違いが発生している可能性が高い．また主に次の CodeGear に遷移する際に CbC コンパイラのバグが生じている可能性もある為, アセンブラレベルの命令を確認しながらデバッグを進めることとなる．

5.3 MoarVM のテスト方法

MoarVM は単体で実行する事が不可能である．また NQP のリポジトリに付随するテストは NQP で書かれている．従って NQP を解釈可能な, NQP のセルフビルド時に生成されるシェルスクリプト nqp が必要である．その為, シェルスクリプト nqp を生成出来ない場合, MoarVM のテストを行

う事が出来ない．CbCMoarVM は NQP のセルフビルドが現時点では達成出来ない為, 通常ではテストが実行出来ない．しかし, MoarVM のバイナリ moar は MoarVM のバイトコードを読み込むことは NQP をセルフビルドしなくとも可能である．

その為, 正常に動作している MoarVM のバイナリ moar と nqp を用意し, この nqp 側から MoarVM のバイトコードに NQP で記述されたテストを変換する．変換された MoarVM のバイトコードはバイナリ moar に渡す事で実行可能であり, テストを行う事が出来る．

6. CbCMoarVM の利点と欠点

MoarVM の様な巨大なスクリプト言語処理系に CbC を適応した所現在までに複数の利点と欠点が発見された．本章ではまず利点を述べ, 次に現段階での CbC を適応した場合の欠点について考察する．

オリジナルの MoarVM では命令コードに対応する箇所はラベルジャンプ, もしくは switch 文で実装されていた．その為同じ C ファイルに命令コードの実行の定義が存在しなければならない．今後 MoarVM に新たなバイトコードが導入されていく事を考えると interp.c が巨大になる可能性がある．関数単位での処理の比重が偏る事に加え, switch 文中に書かれている処理は他の関数から呼ぶ事が出来ないため, 余計な手間がかかる箇所が発生すると考えられる．

CbCMoarVM の場合, CodeGear として基本ブロックを記述出来る為オリジナルの MoarVM の様に swtich 文のブロック中に書く必要性が無くなる．その為類似する命令系をコード分割し, モジュール化する事が可能である．また CbC は goto 文で遷移する以外は通常の C の関数と同じ扱いをする事が可能である．従って CodeGear 内部の処理を別の箇所から使用する事も可能となる為再利用性も向上する．

ThrededCode を実装する場合, 通常命令ディスパッチの箇所と, 実際に実行される命令処理を大幅に変更しなければならない．CbC を用いた実装の場合, 命令処理はただの CodeGear の集合である．その為 CodeGear を ThrededCode に対応した並びとして選択する事ができれば命令処理部分の修正をほぼせずに ThrededCode を実現する事が可能である．

また CodeGear はバイトコードレベルと同じ扱いができるため, ThrededCode そのものを分離して最適化をかける事が可能である．これも CodeGear が関数単位として分離できる事からの利点である．

MoarVM のバイトコードインタプリタの箇所はオリジナルの実装ではラベルジャンプを用いて実装されている．その為, 直接ラベルに break point をかける事が出来ない．作業者がデバッガが読み込んでいる C ソースコードの位置を把握し, 行番号を指定して break point を設定する必要があった．

CbCMoarVM の場合, CodeGear 単位でバイトコードの処理単位を記述している為, 通常の関数と同じく直接 CodeGear に break point をかける事が可能である．これは C プログラミングの関数に対してのデバッグで, 状態ごとに break point をかける事が出来ることを意味する．通常の C 言語で言語処理系を実装した場合と比較して扱いやすくなっていると言える．さらにラベルテーブルでの管理の場合, 次のバイトコード箇所は数値でしか確認できず, 実際にどこに飛ぶのかはラベルテーブル内と数値を作業者が手作業で確認する必要があった．スクリプトなどを組めば効率化は出来るがデバッガ上で完結しない為手間がかかる．CbC 実装では CODES テーブル内は次の CodeGear の名前が入っている為, 数値から CodeGear の名前をデバッガ上で確認する事が出来る．

現在 MoarVM は LuaJit[3] を搭載し JIT コンパイルを行っている．LuaJIT そのものを CbC に適応させるわけではないが, CbC の ABI に JIT されたコードを合わせる事が可能であると推測できる．

しかし, 言語処理系は広く使われる為に著名な OSS などを利用して開発するのが望ましいが, CbC プロジェクトの認知度が低いという現状がある．

また, CbC コンパイラが現在非常にバグを発生させやすい状態になっている．CbC コンパイラは gcc と llvm/clang 上に実装している為, これらのアップデートに追従する必要がある．しかしコン

パイラのバージョンに応じて CbC で利用するコンパイラ内の API が異なる場合が多く, API の変更に伴う修正作業などを行う必要がある．

CbCMoarVM では C から CodeGear へ, CodeGear から C への遷移などが複数回繰り返されているが, この処理中の CodeGear での tail call の強制が非常に難関である．tail call の強制には関数定義の箇所や引数, スタック領域のサイズ修正などを行う必要がある．現在の CbC コンパイラのバグでは CodeGear 内部での不要なスタック操作命令を完全に排除しきれていない．

また CodeGear から C に帰る場合, 環境付き継続を行う必要がある．C の関数の末尾で CodeGear を呼び出している場合など環境付き継続を使用しなくても良いケースは存在するが, 頻繁に C と CbC を行き来する場合記述が冗長になる可能性はある．

7. Threaded Code

現在の MoarVM は次の命令をバイトコードからディスパッチし決定後, ラベルジャンプを利用し実行している．この処理ではディスパッチの箇所にコストが掛かってしまう．CbC を MoarVM に導入することで, バイトコード列を直接サブルーチンコールの列に置き換えてしまう事が可能である．これは CbC が基本ブロックの単位と対応している為である．CbC では現在ディスパッチを行う CodeGear である cbc_next を利用しているが, Threaded Code を実装するにあたり, cbc_next と次の CodeGear に直接遷移する cbc_fixt_next の実装を予定している．

また段階的に現在 8 バイト列を 1 命令コードとして使用しているが, これを 16 バイトなどに拡張し 2 命令を同時に扱えるように実装する事なども検討している．

Perl5 においては perlcc というモジュールが開発されている．これは Perl5 内部で利用している Perl バイトコードを, Perl の C API である XS 言語の様な C のソースファイルに埋め込み, それを C コンパイルでコンパイルするというものである．perlcc を利用することで Perl インタプリタが無い状況でも可動するバイナリファイルを作成する事が可能である．しかし perlcc は Perl スクリプトが複雑になるほど正確に C に移植を行う事が出来ず, 現在では Perl のコアモジュールから外されている．perlcc は Perl のバイトコードを C への変換のみ行う為, C で実装されている Perl 経由で実行した場合と処理速度はほぼ変わらない．また perlcc で生成された C のソースコードは難解であり, これをデバッグするのが困難でもある．MoarVM で threaded code を実現出来た場合, その箇所のみ CbC プログラムとして切り出す事が可能である為 perlcc と似たツールを作成することも可能である．C 言語でも perlcc の様に内部構造を C の関数化すれば ThrededCode の様な物を構築できるが, CbC と比較して処理の単位が明確ではない為高速化は見込めない．また, CbC の CodeGear は基本ブロックそのものである為, CbC プログラムとして切り出す場合, CodeGear をそのまま出力すればよく, 生成された CbC プログラム自体も perlcc と比較して扱いやすい．CbC を用いた ThrededCode で perlcc の様なツールを作成した場合, CodeGear の単位が正常に機能すれば CbC の CodeGear が ThrededCode をより効率化出来ると推測できる．

CbC の CodeGear は goto 文で遷移するため, 次の CodeGear が一意に決定している場合 C コンパイラ側でインライン展開する事が可能である．CodeGear がインライン展開される限界については別途研究する必要があるが, CbC を利用した場合 CodeGear 単位でインライン展開が可能である．その為, ThrededCode を実装する場合に決定した次の CodeGear をインライン展開する事が可能である．従って ThreadedCode を実現するにあたり新たな処理系を開発する必要がなく, 既存の資源を利用して ThreadedCode が実現出来る．これを繰り返す事で perlcc などと比較してより高速化した ThrededCode が実現できる．

CbC を使わずにバイトコードディスパッチの箇所を改良する際に, 関数ポインタを利用する場合も考えられる．この場合は, 関数ポインタの配列を作成し, 次の命令コードに対応する関数をポインタ経由で実行する．C の関数ポインタを利用した場合, CbC と同様に処理のモジュール化は可能である．

しかし, CbC とは違い軽量継続ではなく関数呼び出しで処理をする為, C のスタックフレームが非常に巨大になる．C の関数呼び出しのコストから, 通常の case 文やラベルジャンプを利用した場合と速度差的に優位にならない．また, ThreadedCode の観点では, 命令列に対応した関数を ThreadedCode 用に大幅に修正する必要がある．その為, CbC の様に関数そのものの並びで ThreadedCode に対応させることは出来ない．

8. まとめ

本稿では CbC によって Perl6 の処理系である MoarVM インタプリタの一部改良とその手法を示した．CbCMoarVM ではオリジナルの MoarVM と比較して以下の様な利点が見られた．

- CodeGear 単位で命令処理を記述する事が可能となり, モジュール化が可能となった．
- ThreadedCode を実装する際に効率的に実装ができる見込みが立った．
- CodeGear を導入した命令単位での最適化が可能となった．
- break point を命令の処理単位でかける事が可能となった．
- 現在は命令処理部分を CodeGear に書き換えたのみである為, ラベルを利用した場合と比較して速度としては同等である．

今後 CbC での開発をより深く行っていくにあたり, CbC コンパイラそのものの信頼性を向上させる必要がある．MoarVM の開発を行うにあたり新たに発見された複数のバグを修正し, より安定するコンパイラにする為に改良を行う．

現在 CbCMoarVM で直接バイトコードを入力した場合の NQP のテストは JVM などのテストを除く中で 80%パスする．また数値の計算と出力などの簡単な NQP の例題を作成し, オリジナルの NQP,MoarVM でバイトコード化したものを入力した際も正常に動作している．しかし NQP のセルフビルドは現在オブジェクトの生成に一部失敗している為成功していない．今後はさらに複雑な例題や NQP のセルフビルド, Perl6 の動作を行っていく．

MoarVM では GC からオブジェクトを守る為に MVMROOT というマクロを利用し, 局所変数のポインタをスタックに登録する処理を行っている．GC の制御を効率的に行えば本来は必要ない処理であり, 実行すると CodeGear の優位性が損なわれてしまう．従って MoarVM の GC の最適化を行う．

また高速化という面では, Perl の特徴である正規表現に着目し, 正規表現の表現のみ高速で動く最適化の導入なども検討している．他に rakudo のコンパイラ系統から CbC のコードを直接生成させ, それを llvm でコンパイルすることによって LLVM の最適化フェーズを得て高速化することも可能であると推測できる．

Perl6 の開発は非常に活発に行われている為, CbCMoarVM の最新版の追従も課題となっている．現在は interp.c から Perl スクリプトを用いて自動で CbC の CodeGear を生成している．今後の開発領域の拡大と共により効率的に CbC コードへの自動変換も複数の C コードに対応する様に開発を行っていく．

参考文献

[1] Bell, J. R.: Threaded Code, *Commun. ACM*, Vol. 16, No. 6, pp. 370–372 (online), DOI: 10.1145/362248.362270 (1973).

[2] Ertl, A.: Threaded Code, Technische Universitat Wien (online), available from ⟨`https://www.complang.tuwien.ac.at/forth/threaded-code.html`⟩ (accessed 2018-11-21).

[3] Pall, M.: The LuaJIT Project, luajit.org (online), available from ⟨`http://luajit.org/`⟩ (accessed 2018-11-21).

[4] ParrotFoundation: Parrot, ParrotFoundation (online), available from ⟨`http://parrot.org/`⟩ (accessed 2018-11-21).

[5] Piumarta, I. and Riccardi, F.: Optimizing Direct Threaded Code by Selective Inlining, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, New York, NY, USA, ACM, pp. 291–300 (online), DOI: 10.1145/277650.277743 (1998).

[6] ThePerlFoundation: NQP - Not Quite Perl (6), GitHub (online), available from ⟨`https://github.com/perl6/nqp`⟩ (accessed 2018-11-

21).

[7] ThePerlFoundation: NQP Opcode List, GitHub (online), available from ⟨https://github.com/perl6/nqp/blob/master/docs/ops.markdown⟩ (accessed 2018-11-21).

[8] ThePerlFoundation: Perl 6 Design Documents, ThePerlFoundation (online), available from ⟨https://design.perl6.org/⟩ (accessed 2018-11-21).

[9] ThePerlFoundation: Perl6 Documentation, ThePerlFoundation (online), available from ⟨https://docs.perl6.org/⟩ (accessed 2018-11-21).

[10] ThePerlFoundation: Roast – Perl6 test suite, GitHub (online), available from ⟨https://github.com/perl6/roast⟩ (accessed 2018-11-21).

[11] TOKUMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA* (2015).

[12] Worthington, J.: Rakudo and NQP internals, EDUMENT (online), available from ⟨http://edumentab.github.io/rakudo-and-nqp-internals-course/⟩ (accessed 2018-11-21).

[13] Worthington, J.: Rakudo and NQP internals - day1, EDUMENT (online), available from ⟨http://edumentab.github.io/rakudo-and-nqp-internals-course/slides-day1.pdf⟩ (accessed 2018-11-21).

[14] 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).

[15] 光希宮城，優 桃原，真治河野：Gears OS のモジュール化と並列 API，技術報告 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 (2018).

[16] 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）(2006).

[17] 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 (2012).

[18] 唐鳳：Pugs: A Perl 6 Implementation, Hackage (online), available from ⟨http://hackage.haskell.org/package/Pugs/⟩ (accessed 2018-11-21).

[19] 並列信頼研究室：CbC_gcc, 琉球大学 (online), available from ⟨http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/⟩ (accessed 2018-11-21).

[20] 並列信頼研究室：CbC_llvm, 琉球大学 (online), available from ⟨http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_llvm/⟩ (accessed 2018-11-21).