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，現在は MoarVM,JavaVM,Javascript が動作環境として選択可能である．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る．本論文では CbC を用いて MoarVM の一部を書き換える事を検討し，得られた知見についてに述べる．

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

1. 研究目的

現在も広く使われているスクリプト言語 Perl こと Perl5 の後継言語として Perl6 が開発されている．Perl6 は設計と実装が区分されており，現在広く使われている実装は Rakudo と呼ばれるプロジェクトである．Rakudo の実装は Perl6 コンパイラ開発者用のサブセットである NQP(NotQuitPerl) で実装されている Perl6 の事を指す．現在 Rakudo は NQP を解釈できる実行環境として,C 言語で実装された MoarVM,JVM,Javascript 上で動作する様に開発されている．Rakudo として主に使われている処理系は MoarVM であるが,MoarVM の処理時間が Perl5 などの多くのスクリプト言語と比較し非常に低速である．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．この問題を解決するために現在当研究室で開発している Continuation Based C(以下 CbC) を用いて改良を行う．CbC は C よりさらにきめ細やかな記述が可能であるためスクリプト言語などのプログラミング言語の記述と親和性が高い事が推測される．故に本研究は 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．本稿ではまず CbC,Perl6 の特徴及び現在の実装について述べ，次に改良を行う MoarVM の一連の処理流れについて述べる．そして今回改良した一部分と今後の展開について記す．

[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

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が存在する．これらを 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 CodeSegment という単位を導入した．CbC では CodeSegment,DetaSegment 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 CodeSegment と DataSegment

CbC では C の関数の代わりに CodeSegment を導入する．CodeSegment 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 CodeSegment である事を示す識別子としての意味で利用する．

2.3 軽量継続

CbC では次の CodeSegment に移行する際，C の

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 コンパイラのバグ

CbC コンパイラは現在も開発中であり幾つかのバグが発見されている．まず CodeSegment 内で宣言した局所変数のポインタを別の変数などで確保した状態で goto してしまうと tail call 最適化が切られる．これはただの関数呼び出しになってしまう為，直接的な被害はないものの CbC としての利点が損なわれてしまう．また本来は操作しないはずのスタック領域の操作が入ってしまうため，プログラマの意図と反したスタックポインタなのど操作が行われてしまいバグが発生する可能性が存在する．

2.6 CbC と C の互換性

CbC コンパイラは内部的に与えられているソースコードが CbC であるかどうかを判断する．この際に CodeSegment 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．これは CbC コンパイラが C コンパイラである gcc と llvm/clang 上に実装している為である．その為 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ターゲットと，手を加えていないオリジナルのターゲットの 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．

2.7 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と推測される．CbC における CS 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 CodeSegment の遷移として記述する事が可能である．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 の構想と初期の処理系

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おり様々な処理系が開発されきた．まず 2005 年に唐鳳によって Haskell で実装された Pugs[6] が登場した．Pugs は最初に登場した Perl6 実装であり，この実装を基にして Perl6 の仕様も修正された．現在 Pugs は歴史的な実装となっており，更新はされていない．

3.2 Parrot

その後 Python との共同動作環境として Parrot[3] が実装された．Parrot は PASM と呼ばれるバイトコードを解釈可能なレジスタマシンである．Parrot での Perl6 の実装は NQP(NotQuitPerl) と呼ばれる Perl6 のサブセットで Perl6 を記述するというアイディアの基実装された．ParrotVM は 2006 年の version8.1.0 が最後のリリースである．こちらも Pugs と同様に現在の Perl6 プロジェクトでは歴史的な実装とされている．現在主に使用されている実装である Rakudo は 2010 年に Rakudo-Star という一連のツール郡としてリリースされた．Perl6 処理系自体は現在も未完成であり，Perl6 プロジェクトとして提供しているテストリポジトリ「Roast[9]」で定義されているテストケースを完全に通化する処理系は現在未だ存在しない．

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` という言語名に変更しようという動きが一部存在している．

3.3 Rakudo

Rakudo とは Parrot で構想に上がった NQP,NQP に基づく Perl6 を基にしたプロジェクトである．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 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,Javascript,GraalVM 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 [8] と呼称している．NQP で主に書かれた Perl6 のことを Rakudo と呼ぶ．Perl6 は NQP 以外にも Perl6 独自の一種のシンタックスシュガーの様な物を持っており，これは NQP コンパイラ側で処理を行う．

3.4 NQP

Rakudo における NQP[1] は現在 MoarVM,JVM 上で動作し，MoarVM を一部利用することで NodeJS からも動作させる事が可能である．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 自身は Stage0 と呼ばれる名前空間上のモジュールのみ動作環境の VM のバイトコードを必要とするが, それ以外は NQP で記述されており Bootstrapping されている言語である．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 MoarVMbytecode,jar ファイルが用意されており，Javascript ではバイトコードの代わりにランタイム独自の ModuleLoader などが設計されている．MoarVM の ModuleLoader は Stage0 ある MoarVMbytecode 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 bytecode 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 は 6model と呼ばれるオブジェクトモデルを採用としているが，これを構築する為に必要な NQPCORE, 正

規表現系の QRegex,MoarVM の ModuleLoader などが moarvmbytecode で記述されている．これら MoarVMBytecode の拡張子は.moarvm である．MoarVM に対して Stage0 のディレクトリにライブラリパスを設定し，nqp.moarvm を実行させることで nqp の対話型環境が起動する．

実際に 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．

Roast やドキュメントなどによって設計が定まっている Perl6 とは異なり NQP 自身の設計は今後も変更になる可能性が開発者から公表されている．現在の公表されている NQP のオペコードは NQP の GitHub リポジトリ [2] に記述されているものである．

3.5 Rakudo Perl6

Rakudo 実装上における Perl6 は Rakudo Perl6 と呼ばれている Git リポジトリで管理されているプログラムのことである．前述した通り Rakudo Perl6 は Perl6 のサブセットである NQP を用いて記述されている．従って yacc や lex と言った Perl5 の文字解析，構文解析に利用していたプログラムは利用せず，NQP 側で構文定義などを行っている．NQP は NQP 自身で Bootstrapping されている為，Rakudo Perl6 の build 時には NQP の実行環境として要した VM, それに基づいて build した NQP がそれぞれ必要となる．

言語的な特徴としては Perl5 とは違いアトミックに演算を行う事が可能な絵文字で実装された atom 演算子や，すべてがオブジェクトである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．

3.6 現在の Perl6

Perl6 の言語仕様 [4] とその時点での実装状況を纏めた公式ドキュメント [5] は分離している．従来は言語仕様は自然言語の仕様書であったが，現在はテストスイートである「Roast[9]」そのものと定義されている．現在の Perl6 の仕様を読む場合 Roast を確認し，現在 rakudo 上に実装されている機能を見る場合は公式ドキュメントを確認する必要がある．

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．

4.1 方針

MoarVM そのものを CbC で書き換えることも考えられるが MoarVM 自体既に巨大なプロジェクトである為すべてを書き換える事は困難である．その為まず比較的 CbC で書き換えることが容易な箇所を修正する．前章までに述べた通り CbC の CodeSegment 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 CodeSegment に書き換える事が可能である．MoarVM における基本ブロックはインタプリタが実行するバイトコードごとの処理に該当する．

4.2 MoarByteCode のディスパッチ

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 MVM_CGOTO フラグが立ちラベル遷移を利用する．それ以外の場合は巨大な 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ラベルテーブルにアクセスし，テーブルに登録されているアドレスを取得し，NEXT で遷移する．このラベルテーブルの中身はラベルが変換されたアドレスであるため，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．巨大

な case 文として実行された場合，実行時間が遅いだけでなく，ラベル遷移と共存させて記述を行っている為 C のソースコードにおける可読性も低下する．

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 CodeSegment を作成し，CodeSegment 名前を要素として持つ CbC の CodeSegment のテーブルを作成した．

4.3 命令実行箇所の CodeSegment 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，CodeSegment の遷移として利用する．interp.c は?に示す様なスタイルで記述されている．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 から遷移する為，バイトコードの名前は LABELS の配列の添字に変換されている．そのため対象となる CodeSegment を LABLES の並びと対応させ，配列 CODES に設定すれば CodeSegment の名前は問わない．今回は CodeSegment である事を示す為に suffix として cbc_をつける．

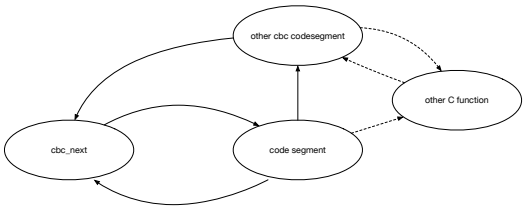

図 1: CbC におけるインタプリタの関数遷移

4.4 MoarVM の Bytecode のデバッグ

moar に対して moarvm bytecode を dump オプションを付けて読み込ませると moarvm の bytecode がアセンブラの様に出力される．しかしこれは moarvm が実行した bytecode のトレースではなく moarvm bytecode 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 MoarVMBytecode 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．

```
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
(gdb) c
The program is not being run.
(gdb) run
Starting program: /mnt/dalmore-home/one/src/
    Perl6/nqp/../../MoarVM_Original/MoarVM/
    moar --libpath=src/vm/moar/stage0 gen/
    moar/stage1/nqp.moarvm
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/
    libthread_db.so.1".
[New Thread 0x7ffff629a700 (LWP 176412)]

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 1: オリジナル版 MoarVM のバイトコードのトレース

4.5 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 で MoarVM を CbC とオリジナル版での並列デバッグを人間が同時に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．CbC 側は cbc_next に break point を設定し，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 CodeSegment の名前を直接確認できるが，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探す必要がある．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．違いが生じている箇所が発見できた場合，その前後の CodeSegment 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．その為，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．

4.6 CbC コンパイラによるバグ

現在までの CbC は複数個の入出力を CodeSegment に与えるユースケースで利用していた．CbC コンパイラ自身はそれぞれ用意したテストスイートを通化するものの，MoarVM の様な巨大なプロジェクトの CS をコンパイルを実行する場合，予期せぬバグが発生した．主に CS 間の goto における tail call フラグの除去や，DS として渡している構造体の変数のアドレスがスタックポインタの値より上位に来てしまい，通常の C の関数を call した際にローカル変数の領域が DS のアドレスの周辺を利用してしまう．その為 DS の構造体の値が書き換わり，C から DS に return した際に DS の構造体が破壊されるバグである．このバグは先程の並列デバッグを行いながらプログラムカウンタや変数の動きをトレースする事などで発見することが出来る．現状では CbC コンパイラがプログラマの意図と反する挙動を取るため CbC コンパイラのバグを回避するプログラミングが要求されている．本来コンパイラ側のバグを回避するプログラミングをプログラマに要求するスタイルは好ましくない．従って CbC コンパイラ自身の信頼性を向上させる事も今後の課題となっている．

4.7 Threaded Code

CbC は CodeSegment で末尾最適化 (Tail call optimization) を行う．これは CodeSegment は必ず関数呼び出しではなく goto で次の状態に遷移する為にスタック領域の操作が必要とならない為である．現在の CbC コンパイラの実装では CodeSegment から C の関数に戻る場合は末尾最適化を切り，CodeSegment 間の遷移では末尾最適化が行われる．末尾最適化を応用することで Continuation-passing スタイルの Threaded Code の実装が可能となる．[10]

現在の CbCMoarVM は次の命令セットのディスパッチを cbc_next という CodeSegment で処理している．これは元の MoarVM の命令ディスパッチで行われる現在のオペコードを示す cur_op と命令列 op の操作及び次のラベルに遷移するマクロに該当する．CbCMoarVM ではラベルに対しての遷移の代わりに MoarVM の命令の CodeSegment の集合体である配列 CODES にアクセスし，その要素である CodeSegment に対して遷移する形を取っている．この一連の処理がオーバーヘッドになる為，今後は cbc_fixt_next という CodeSegment を導入し直接次の命令に該当する CodeSegment へ goto する様に実装する予定である．

4.7.1 perlcc

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 に変換しただけであり，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．この場合,Perl6 を通常動かした際とは異なりバイトコードインタプリタに到達する前の処理が無くなる為多少の高速化が望めると推測できる．

5. CbC を用いる事についての利点と欠点

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．

5.1 利点

5.2 コード分割

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，CodeSegment 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．これは通常のインタプリタの実装と比べ可読性と言う意味とモノリシックアーキテクチャをマイクロ化出来るという意味でも利点である．

5.2.1 MoarVM のデバッグ

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 debug point を設定する必要があった．

CbCMoarVM の場合，CodeSegment 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 CodeSegment にデバッグポイント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 CodeSegment の名前が入っている為，数値から CodeSegment の名前をデバッガ上で確認する事が出来る．

5.3 欠点

5.3.1 CbC コンパイラ

前章までに複数述べた通り 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 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 CbC へ，CbC から C への遷移などが複数回繰り返されているが，CodeSegment 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．この実装の為にはどのケースで関数が定義されているかを CbC コンパイラで明確に分割する必要がある．

6. 今後の課題

本論文では 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．CbC の CodeSegment 部分を用いることできめ細やかな記述が出来，デバッグし辛い箇所も breakpoint の設定などが容易になった．

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%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．今後はさらに複雑な例題や Perl6 の独自文法でも動くかどうかの実験を行う．

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 CodeSegment 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．

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 CodeSegment 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開発していく．

参考文献

[1] https://github.com/perl6/nqp.
[2] https://github.com/perl6/nqp/blob/master/docs/ops.markdown.
[3] http://parrot.org/.
[4] https://design.perl6.org/.
[5] https://docs.perl6.org/.
[6] http://hackage.haskell.org/package/Pugs.
[7] http://edumentab.github.io/rakudo-and-nqp-internals-course/.
[8] http://edumentab.github.io/rakudo-and-nqp-internals-course/slides-day1.pdf.
[9] https://github.com/perl6/roast.
[10] https://www.complang.tuwien.ac.at/forth/threaded-code.html.
[11] ShinjiKONOImplementing Continuation based language in LLVM and Clang (2015).
[12] 河野真治 Gears OS のモジュール化と並列 API (2018).
[13] 河野真治第 53 回プログラミング・シンポジウム (2012).
[14] 河野真治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] 並木美太郎 Ruby 用仮想マシン YARV の実装と評価 (2006).