

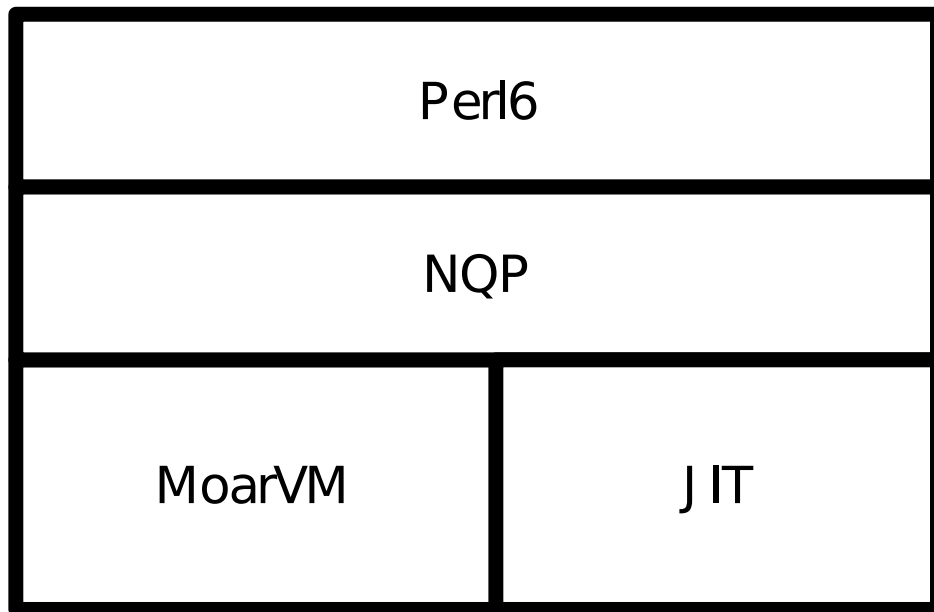
# CbCによるPerl6処理系

清水隆博 並列信頼研

---

## 研究目的

- 現在開発されているPerl6の実装にRakudoがあり, RakudoはNQP(Perl6のサブセット)で記述されたPerl6, NQPで記述されたNQPコンパイラ, NQPを解釈するVMで構成されている
- NQPコンパイラはRakudoのVMであるMoarVM用のバイトコードを生成する
- MoarVMはこのバイトコードを解釈, 実行する

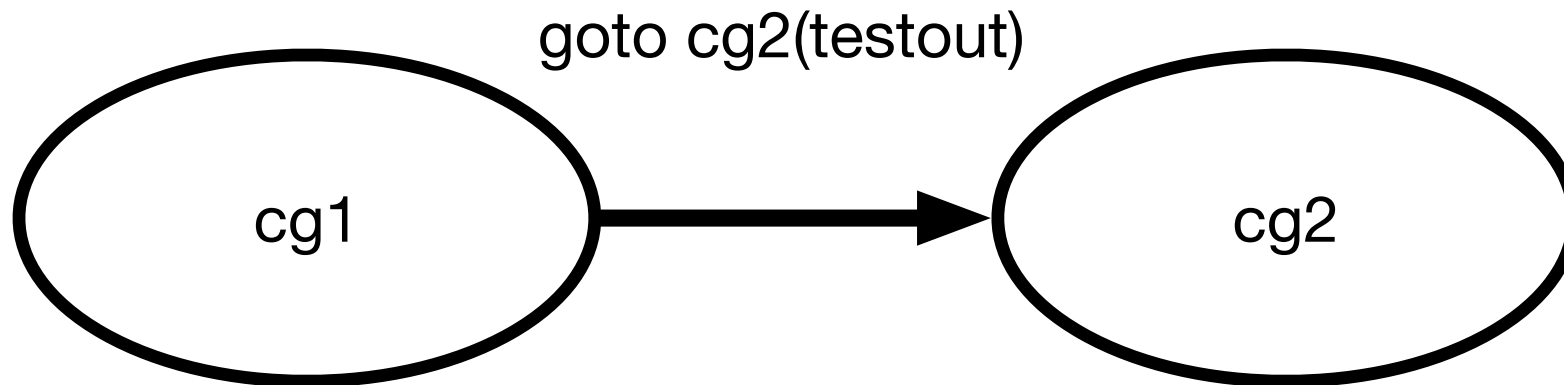


## 研究目的

- Continuation based C (CbC)という言語は継続を基本とするC言語であり, 言語処理系に応用出来ると考えられる
- スクリプト言語などは, バイトコードを扱うが, この実行にcae文や, ラベルgotoなどを利用している。
  - この部分はCbCの機能で書き換える事が可能である
- 命令実行処理部分をモジュール化することで、各命令ごとの最適化や、命令ディスパッチ部分の最適化を行う事が可能であると考ええる。
- 従って, CbC一部用いてPerl6にC処理系であるMoarVMの書き換えを行い, 処理を検討する.

## Continuation Based C (CbC)

- Continuation Based C (CbC) はCodeGearを単位として用いたプログラミング言語である.
- CodeGearはCの通常の間数呼び出しとは異なり, スタックに値を積まず, 次のCodeGearにgoto文によって遷移する.
- CodeGear同士の移動は、状態遷移として捉える事が出来る



## Continuation Based C (CbC)

- CodeGearはCの関数宣言の型名の代わりに `__code` と書く事で宣言出来る
- CodeGearの引数は, 各CodeGearの入出力として利用する
- gotoしてしまうと、元のCodeGearに戻る事が出来ない

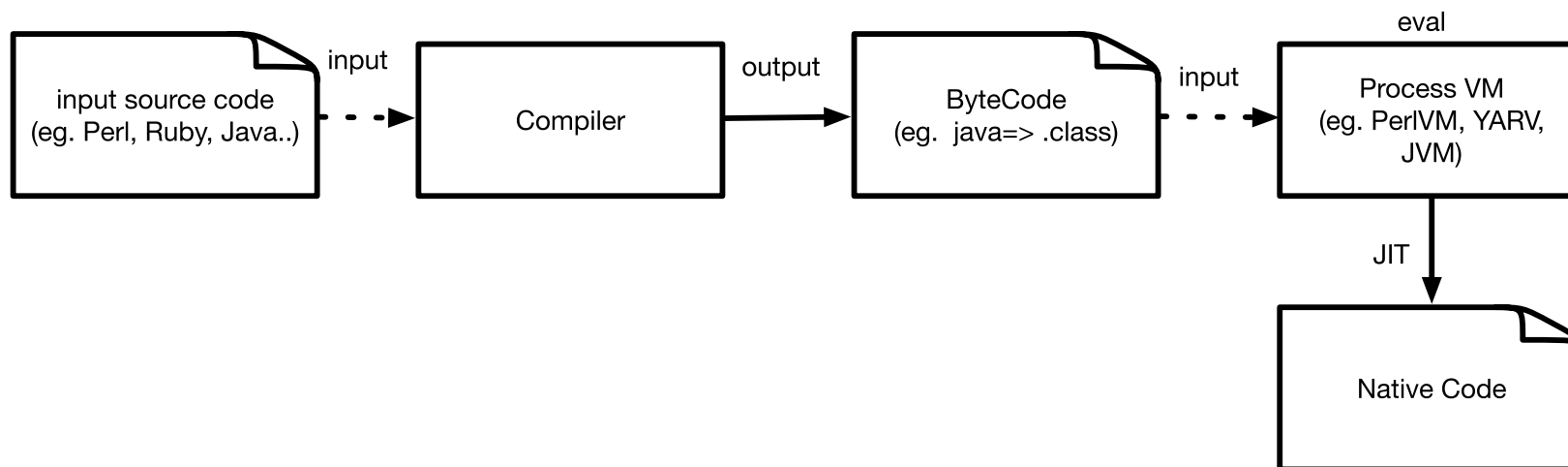
```
__code cg1(TEST testin){
    TEST testout;
    testout.number = testin.number + 1;
    testout.string = "Hello";
    goto cg2(testout);
}

__code cg2(TEST testin){
    printf("number = %d\t string= %s\n",testin.number,testin.string);
}

int main(){
    TEST test = {0,0};
    goto cg1(test);
}
```

## 言語処理系の応用

- スクリプト言語は入力として与えられたソースコードを、直接評価せずにバイトコードにコンパイルする形式が主流となっている
- その為スクリプト言語の実装は大きく2つで構成されている
  - バイトコードに変換するフロントエンド部分
  - バイトコードを解釈する仮想機械



---

# Rakudo

- Rakudoとは現在のPerl6の主力な実装である.
- Rakudoは次の構成になっている
  - 実行環境のVM
  - Perl6のサブセットであるNQP(NotQuitPerl)
  - NQPで記述されたPerl6(Rakudo)

---

## MoarVM

- Perl6専用のVMであり, Cで記述されている
- レジスタマシンとして実装されている.



---

## MoarVMのバイトコード

- MoarVMは16ビットのバイナリを命令バイトコードとして利用している
- 命令にはその後に16ビットごとにオペランド(引数)を取るものがある

```
add_i loc_3_int, loc_0_int, loc_1_int  
set loc_2_obj, loc_3_obj
```

---

## MoarVMのバイトコードインタプリタ

- バイトコードは連続したメモリに確保されている
- その為次の処理を繰り返す必要がある
  - 16ビットごとで読み込み
  - 読み込んだビットから、命令に対応する処理を呼び出し
  - その処理を実行する
- この処理をバイトコードディスパッチと呼び、 実行する部分をバイトコードインタプリタと呼ぶ

# MoarVMのバイトコードインタプリタ

- MoarVMは関数 `MVM_interp_run` でバイトコードに応じた処理を実行する
- マクロDISPATCHで、ラベルgotoかcase文に変換が行われる
  - バイトコードは数値として見る事が出来る為、 case文に対応する事が出来る
  - この中の `OP` で宣言されたブロックがそれぞれバイトコードに対応する処理となっている。
- `cur_op` は次のバイトコード列が登録されており、マクロ `NEXT` で決められた方法で次のバイトコードに対応した処理に遷移する。

```
DISPATCH(NEXT_OP) {  
  OP(const_i64):  
    GET_REG(cur_op, 0).i64 = MVM_BC_get_I64(cur_op, 2);  
    cur_op += 10;  
    goto NEXT;  
}
```

## MVM\_interp\_runで使用されているマクロ

```
DISPATCH(NEXT_OP) {  
    OP(const_i64):
```

- マクロ `OP` 及び `NEXT` は次の様に定義している

```
#define OP(name) OP_ ## name  
#define NEXT *LABELS[NEXT_OP]
```

- マクロ `DISPATCH` は, ラベルgotoが利用できる場合は無視される
- マクロ `OP` が, 対応するバイトコード命令を, ラベル列に変換する

```
OP_const_i16:  
OP_const_i32:  
    MVM_exception_throw_adhoc(tc, "const_iX NYI");  
OP_const_i64:
```

## MVM\_interp\_runで使用されているマクロ

- 次のバイトコード命令に遷移するマクロ `NEXT` は、ラベルgotoが使用可能な場合次の様に記述されている
- `NEXT` 自体はラベルテーブルにアクセスし、ラベルを取り出す
- 次のバイトコードを取り出すのは、`NEXT_OP` というマクロが担っている

```
#define NEXT_OP (op = *(MVMuint16 *) (cur_op), cur_op += 2, op)
#define NEXT *LABELS[NEXT_OP]
```

- マクロ `NEXT` は次の様に展開される

```
goto *LABELS[(op = *(MVMuint16 *) (cur_op), cur_op += 2, op)];
```

## MVM\_interp\_runのラベルテーブル

- 利用するCコンパイラが、ラベルgotoをサポートしている場合に実行される
- 配列 `LABELS` にアクセスし、ラベル情報を取得する
- ラベル情報を取得出来ると、そのラベルに対してラベルgotoを利用する

```
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,
```

## MVM\_interp\_run

- Cの実装の場合, switch文に展開される可能性がある
  - 命令ディスパッチが書かれているCソース・ファイルの指定の場所にのみ処理を記述せざるを得ない
  - 1ファイルあたりの記述量が膨大になり, 命令のモジュール化ができない
- 高速化手法の、 Threaded Codeの実装を考えた場合, この命令に対応して大幅に処理系の実装を変更する必要がある.
- デバッグ時には今どの命令を実行しているか, ラベルテーブルを利用して参照せざるを得ず, 手間がかかる.

---

## CbCでの変換

- CbCのCodeGearは関数よりも小さな単位である
- その為、従来は関数化出来なかった単位をCodeGearに変換する事が出来る
- CbCをMoarVMに適応すると、ラベルなどで制御していた命令に対応する処理をCodeGearで記述する事が可能である



## CbC MoarVMのバイトコードディスパッチ

- オリジナルでは、マクロ `NEXT` が担当していた、次のバイトコードへの移動は、NEXT相当のCodeGear `cbc_next` で処理を行う
- CodeGearの入出力として、MoarVMなどの情報をまとめた構造体を利用する

```
__code cbc_next(INTERP i){
    __code (*c)(INTERP)
    c = CODES[(i->op = *(MVMuint16 *) (i->cur_op), i->cur_op += 2, i->op)]; // c = NEXT(i)
    goto c(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);
}
```

## CodeGearの入出インターフェイス

- MoarVMではレジスタの集合や命令列などをMVM\_interp\_runのローカル変数として利用し、各命令実行箇所参照している
- CodeGearに書き換えた場合、このローカル変数にはアクセスする事が不可能となる。
- その為、入出力としてMoarVMの情報をまとめた構造体interpのポインタであるINTERPを受け渡し、これを利用してアクセスする

```
typedef struct interp {
    MVMuint16 op;
    MVMuint8 *cur_op;
    MVMuint8 *bytecode_start;
    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;
```

## CbC MoarVMのCodeGearテーブル

- CodeGearテーブルは引数としてINTERPを受け取るCodeGearの配列として定義する
- テーブルとして宣言することで、バイトコードの値をそのままテーブルに反映させる事が可能である

```
__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,  
}
```

## MoarVMとCbCMoarVMのトレース

- MoarVMのデバッグ時には、次の命令が何であるかは直接は判断出来なかった

```
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
```

- CbCMoarVMの場合は、次に実行する命令名を確認する事が出来る

```
Breakpoint 2, cbc_next (i=0x7ffffffffffdc30) at src/core/cbc-interp.cbc:61
61     goto NEXT(i);
$1 = (void (*)(INTERP)) 0x7ffff7566f53 <cbc_takeclosure>
$2 = 162
```

## MoarVMのデバッグ

- `cur_op`のみをPerlスクリプトなどを用いて抜き出し, 並列にログを取得したオリジナルと差分を図る
- この際に差異が発生したバイトコードを確認し, その前の状態で確認していく

```
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
```

## 現在のCbCMoarVM

- 現在はNQP, Rakudoのセルフビルドが達成でき, オリジナルと同等のテスト達成率を持っている
  - その為、 NQP, Rakudoの実行コマンドであるnqp perl6が起動する様になった
- moarの起動時のオプションとして `--cbc` を与えることによりCbCかオリジナルを選択可能である
- Perl6の実行バイナリperl6, NQPの実行バイナリnqp は, それぞれmoarを起動するシェルスクリプトである
- `--cbc` オプションをシェルスクリプト内に書き加えることで, Perl6, NQPがそれぞれCbCで起動する

```
#!/bin/sh
exec /mnt/dalmore-home/one/src/Perl6/Optimize/llvm/build_perl6/bin/moar --cbc \
  --libpath=/mnt/dalmore-home/one/src/Perl6/Optimize/llvm/build_perl6/share/nqp/lib \
  /mnt/dalmore-home/one/src/Perl6/Optimize/llvm/build_perl6/share/nqp/lib/nqp.moarvm "$@"
```

---

## ThreadedCodeの実装

- MoarVM内のバイトコードに対応する処理が分離出来たことにより, バイトコードに該当するCodeGearを書き連ねることによってThreadedCodeが実装可能となる

## CbCMoarVMと通常のMoarVMの比較

- CbCMoarVMと通常のMoarVMの速度比較を行った
- 対象として, 単純なループで数値をインクリメントする例題と, フィボナッチ数列を求める例題を選択した
- NQPで実装し, 速度を計測した

```
#! nqp

my $count := 100_000_000;

my $i := 0;

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

```
#! nqp

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

my $N := 30;

my $z := fib($N);

say("fib($N) = " ~ fib($N));
```





---

## フィボナッチの例題

- オリジナル
  - 1.379 sec
  - 1.350 sec
  - 1.346 sec
- CbCMoarVM
  - 1.636 sec
  - 1.804 sec
  - 1.787 sec
- フィボナッチの例題ではCbCMoarVMが劣る結果となった

---

## 単純ループ

- オリジナル
  - 7.499 sec
  - 7.844 sec
  - 6.746 sec
- CbCMoarVM
  - 6.135 sec
  - 6.362 sec
  - 6.074 sec
- 単純ループではCbCMoarVMの方が高速に動作する場合もある

---

## CbCMoarVMの利点

- バイトコードインタプリタの箇所をモジュール化する事が可能となった
  - CodeGearの再利用性や記述生が高まる
  - CodeGearは関数の様に扱える為、命令ディスパッチの最適化につながる実装が可能となった
- デバッグ時にラベルではなくCodeGearにbreakpointを設定可能となった
  - デバッグが安易となる
- CPUがキャッシュに収まる範囲の命令の場合、通常のMoarVMよりも高速に動作する

---

## CbCMoarVMの欠点

- MoarVMのオリジナルの更新頻度が高い為, 追従していく必要がある
- CodeGear側からCに戻る際に手順が複雑となる
- CodeGearを単位として用いる事で複雑なプログラミングが要求される.

---

## まとめと今後の課題

- 継続と基本としたC言語 Continuation Based Cを用いてPerl6の処理系の一部を書き直した
- CbCの持つCodeGearによって, 本来はモジュール化出来ない箇所をモジュール化する事が出来た

