Continuation Based Cによる Perl6処理系

並列信頼研

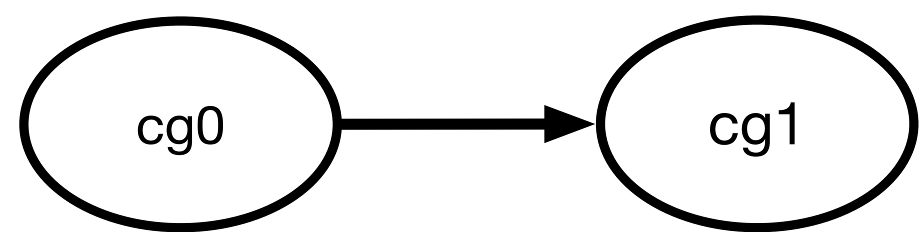
清水 隆博，河野 真治(琉球大学)

研究概要

・プログラミング言語Perl5の後継言語として，Perl6が開発されている
・Perl6の主流な実装はRakudoであり, MoarVMというVMと, Perl6のサブセットであるNQP, NQPで記述されたPerl6という構成になっている
・MoarVMは起動時間及び全体的な処理速度が他のスクリプト言語と比較して低速である.
・当研究室で開発しているプログラミング言語に継続を中心としたC言語であるContinuation Based C(CbC)がある
・MoarVMのバイトコードインタプリタ部分の巨大なswitch文などが, CbCのCodeGearに変換可能である為, CbCを用いて記述した場合, モジュール化や高速化などが期待される.

Continuation Based C

・Code Gearを処理の単位として用いるプログラミング言語
・CodeGearから次のCodeGearへの遷移を軽量継続と呼ぶ
・軽量継続はCの関数呼び出しとは異なり, フレームポインタ, スタックポインタの操作によるスタックへの状態保存を行わない
・軽量継続はgoto文を用いて行う



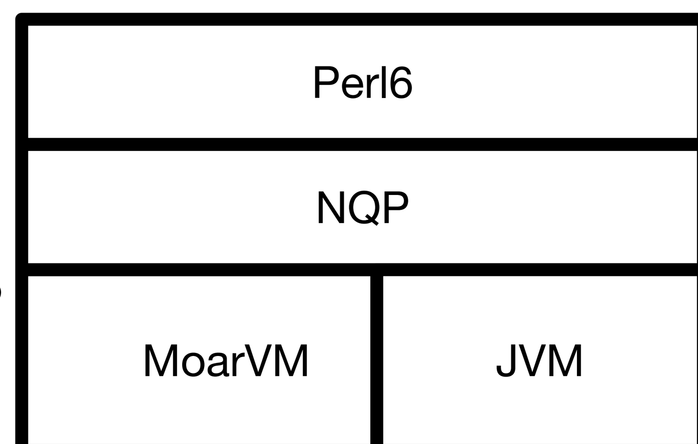

```
__code cg0(int n){
    goto cg1(++n);
}

__code cg1(int n){
    n *= 2;
    goto cg2(n);
}
```

Perl6

・Perl5の後継として,開発が進められているスクリプト言語
・仕様と実装が区分されており, 仕様はテストスーツ, Roastである
・Pugs, Parrotなどの実装が歴史的に存在し, 現在はRakudo実装が主流となっている
・Perl5とは互換性が存在しない
・漸進的型付け言語である

Rakudo

・現在の主流なPerl6の実装である
・RakudoはVM, NQP(NotQuitPerl) というPerl6のサブセット, NQPで主に記述されたPerl6という構成になっている
・Rakudoで選択可能なVMはMoarVMというPerl6専用のVM, JVMを選択可能である
・MoarVM + Rakudoの組み合わせが現在主に使用されている
・Rakudoにおけるコンパイラは, フロントエンドと呼ばれる, Perl6, NQPからVMのバイトコードの変換箇所と, VMがバイトコードを実行する, バックエンドと呼ばれる箇所の2種類が存在する



MoarVMへのCbCの応用

・MoarVMの中心は与えられたバイトコードに基づいて命令を処理するバイトコードディスパッチ部分である
・CbCにおけるCodeGearは関数よりも細かな単位であり， コンパイラの基本ブロックとみなせる為, 命令に対応する処理をCodeGearとして変換する事が可能である.
・その為, CbCを用いてMoarVMのバイトコードディスパッチ部分の書き換えを検討する

MoarVMのバイトコードディスパッチ

・MoarVMはsrc/core/interp.c内でバイトコードディスパッチを行う様に実装されている
・MoarVMは, Cのラベルに対してgotoが出来る場合は,ラベルを利用したgotoで次の命令に遷移し, 使えない場合は, switch文によって遷移する
・次の命令へはマクロNEXTを利用する事で計算され, 遷移する
・この方法では命令に対応する処理を, Cのソースファイルの指定された場所に記述する必要があり, モジュール化などができない

```
DISPATCH(NEXT_OP) {
    OP(no_op):
        goto NEXT;
    OP(cont_i8):
    OP(cont_i16):
    OP(cont_i32):
        MVM_exception_throw_adhoc(tc, "cont_iX NYI");
    OP(cont_i64):
        GET_REG(cur_op, 0).i64 = MVM_BC_get_I64(cur_op, 2);
        cur_op += 10;
        goto NEXT;
}
```

CbCMoarVM

・CodeGearを用いて命令処理部分をCbCに変換する
・各CodeGearの遷移は, 直接次のCodeGearに継続するか, cbc_nextというCodeGearを経由して, 次のCodeGearを求めて継続する
・レジスタなどの本来ローカル変数で利用していたものは, CodeGearの入出力して与える事で, CodeGear内で使用出来る様にする

```
__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);
}
```

デバッグ手法

・MoarVMはUUIDのような物をバイトコードに埋め込む為, ランダムなバイトコードを実行している様に見える
・その為, 一度MoarVMのバイトコードにNQPなどのソースを変換し, そのバイトコードをオリジナルとCbCで修正した両者で並列に実行し, gdbなどのCデバッガを用いてデバッグを行う
・実行するバイトコードの数は膨大である為, 実行したバイトコードの番号を出力するようにし, scriptなどのコマンドを用いてログを取り, 差分を解析する

```
391 : 391
749 : 749
53 : 53
*54 : 8
```

現在のCbCMoarVM

・NQP, Rakudoともにセルフビルドを実現できた
・速度的には再帰呼出しを多用するフィボナッチの例題ではオリジナルより劣るが, 単純なループ処理などではオリジナルを上回るケースがある