Continuation based C の LLVM/clang 3.5 上の実装について

徳森海斗[†]　河野真治[†]

当研究室では並列・分散プログラミングスタイルとして Data Segment, Code Segment を用いるプログラミング手法を提案している．この手法を用いるプログラミング言語として CbC の開発を行っており，これは C の下位の言語になる．本研究では，LLVM/clang-3.5 をベースとした CbC コンパイラの具体的な実装のその評価について述べる．

The implementation of Continuation based C Compiler on LLVM/clang 3.5

Kaito Tokumori[†] and Shinji Kono[†]

A programming paradigm which use data segments and code segments is proposed. CbC is a lower language of C for this paradigm. CbC has standalone compiler and GCC version. In this study, we add an implement CbC compiler on LLVM/clang-3.5. The detail of implementation and evaluation are shown.

1. Continuation based C on LLVM

当研究室では，プログラムを code segment, data segment という単位を用いて書くという手法を提案している．この手法を用いてプログラミングを行う言語として Continuation based C (以下 CbC) というプログラミング言語を開発しており，これは C の下位の言語にあたる．CbC においてコードセグメント間の処理の移動は goto 文を用いた軽量継続によって行われる．CbC の goto 文はハードウェア記述のような状態遷移ベースのプログラミングを行うのに適しており，プログラム生成のターゲットとして優れている．例えば，正規表現検査器を状態遷移系に変換する場合にはアセンブラや LLVM などの中間コードよりもポータブルかつ読みやすい形で生成することができる．CbC では呼び出し前の code segment に戻るための環境がないので，スレッドのコンテキストスイッチを容易に実装することができる．これにより OpenCL, CUDA, そして Cerium といった並列開発環境を用いたプログラムの記述に向いている．また，meta code segment, meta data segment といったメタレベルの単位を用意することで柔軟なメタプログラミングが可能になる．

これまでに開発された CbC のコンパイラは Micro-C をベースにしたものと GCC をベースにしたものの二種がある．CbC での goto 文を実装するには，末尾最適化を利用することができる．実際に，GCC の中間コードやコード生成器を最小限に変更し，最適化と干渉することなく CbC 言語を実現することが可能なことを示した．

今回，広く使われるようになってきた LLVM 上に CbC を実装した．LLVM は GCC と異なり中間コードを生成する clang と，中間コードをアセンブラに変化する llvm の二つのプログラムに完全に分かれている．中間コードには，大域 Jump に相当するコードがないので，GCC と同じ方法では実装することはできない．また，GCC で CbC の code segment から C に戻る時に使う環境付き goto を実現するのに用いた nested function は LLVM には存在しない．

中間コードでは CALL や RETURN は，そのままで，CALL に存在する末尾最適化フラグを利用することで code segment であることを表す．このフラグは，Haskell や Erlang あるいは Scheme など，末尾最適化が言語仕様に含まれるものに対して用意されている．なので，LLVM の 中間コード自体の変更は不要であることがわかった．llvm 側では，フラグに沿って末尾最適化の強制を行う．これにより，LLVM の最適化とさまざまなアーキテクチャのコード生成に干渉することなく CbC を実現できた．

LLVM や最近の GCC では最適化を関数展開に頼っており，それを禁止すると生成されるコードの質が低下する．CbC では goto は単一の jmp 命令になるの

† 琉球大学
University of the Ryukyu

で, 相対的に関数展開の重要性が低下する. 実装した LLVM のコードは十分な性能を持つが, GCC よりは若干性能が落ちている. GCC では末尾最適化は最適化レベル 2 でのみ行われるので, デバッガなどが有効になる最適化レベル 0 では CbC を実行することはできない. 今回作成した LLVM 版では, 最適化レベル 0 でも末尾最適化を矯正できるので, デバッガなどを適切に使うことができる.

2. CbC の例題

CbC では C の関数の代わりに code segment を用いて処理を記述し, code segment 間の移動に goto を用いる. 構文は それ以外は C と同じである. 通常の C の関数をそのまま使用することもできる. 必要であれば, 関数呼び出しは goto を用いた継続に, ループ制御を再帰的な継続に置き換えて, goto のみのプログラムに変換することも可能である. 本論文で用いる例題 conv1 は, そのような変換の例になっている.

code segment の記述は C の関数の構文と同じで, 型に __code を使うことで宣言でき, code segment 間の移動は goto の後に code segment 名と引数を並べて記述することで行える. この goto による処理の遷移を継続と呼ぶ.

code segment は C の関数と異なり戻り値を持たず, 必要な値は継続の際に次の code segment へ引数として渡す. C において関数呼び出しを繰り返し行う場合, 呼び出された関数の引数の数だけスタックに値が積まれていく. しかし, 戻り値を持たない code segment ではスタックに値を積んでいく必要が無く, スタックは変更されない. このようなスタックに値を積まない継続, つまり呼び出し元の環境を持たない継続を軽量継続と呼び, 軽量継続により並列化, ループ制御, 関数コールとスタックの操作を意識した最適化がソースコードレベルで行えるようになる.

以下の図 1 に示されたプログラムは与えられた数値の階乗を算出する CbC プログラムであり, 図 2 はこのときの code segment 間の処理の流れを表している.

3. LLVM/clang

LLVM はコンパイラ, ツールチェーン技術等を開発するプロジェクトの名称である. 単に LLVM といった場合は LLVM Core を指し, これはコンパイラの基板となるライブラリの集合である. LLVM IR や LLVM BitCode と呼ばれる独自の言語を持ち, この言語で書かれたプログラムを実行することのできる仮想機械も持つ. また, LLVM IR を特定のターゲットの機械語

```
__code print_factorial(int prod)
{
  printf("factorial = %d\n",prod);
  exit(0);
}
__code factorial0(int prod, int x)
{
  if ( x >= 1) {
    goto factorial0(prod*x, x-1);
  }else{
    goto print_factorial(prod);
  }
}
__code factorial(int x)
{
  goto factorial0(1, x);
}
int main(int argc, char **argv)
{
  int i;
  i = atoi(argv[1]);

  goto factorial(i);
}
```

図 1 階乗を計算する CbC プログラムの例

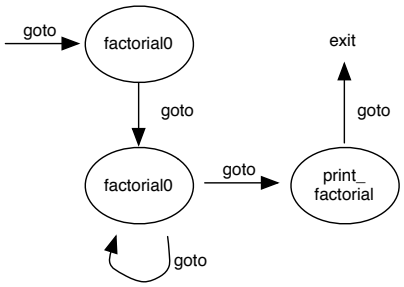

図 2 goto による code segment 間の継続

に変換することが可能であり, その際に LLVM の持つ最適化機構を利用することができる. clang は バックエンドに LLVM を利用する C/C++/Objective-C コンパイラである. 具体的には与えられたコードを解析し, LLVM IR に変換する部分までを自身で行い, それをターゲットマシンの機械語に変換する処理と最適化に LLVM を用いる.

図 3 は clang がソースコードを読み込み, アセンブリコードを出力するまでの流れを表した図である. clang はソースコードを与えられると, パーサーで構文解析を行い, AST を生成する. そして AST を CodeGen で LLVM IR に変換する. LLVM IR をアセンブリコードに変換するまでの処理は LLVM が用いられる. LLVM IR はまず, SelectionDAGISel によって Machine Code に変換される. この時, 直接変換されるのではなく SelectionDAG という内部表現に一度変換され, 最適化された後に変換される. Machine Code は Machine code に対する最適化をかけられた後, Code Emission を通じてアセンブリコードに変換

される．これらの内部表現の他に, clang が型を表現するのに用いる QualType というクラスについても説明する．

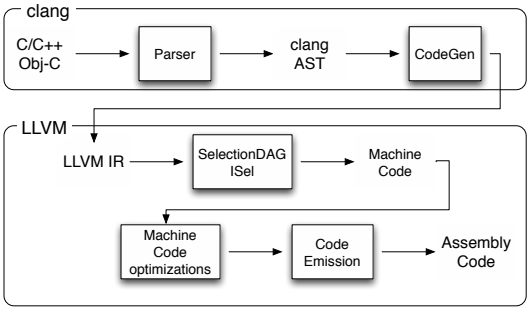

図 3 clang, LLVM によるコンパイルの一連の流れ

3.1 QualType

Qualtype は変数や関数等の型情報を持つクラスで, const, volatile 等の修飾子の有無を示すフラグと, int, char, * (参照) 等の型情報を持つ Type オブジェクトへのポインタを持つ．Type クラスは isIntegerType や isPointerType と言った関数を持つのでこれを利用して型を調べることができる．また, ポインタ型である場合には getPointeeType という関数を呼び出すことでそのポインタが指す型の Type を持つ QualType を得ることができる．修飾子の有無は const なら isConstQualified, volatile なら isVolatileQualified といった関数を用いて確認できる．

図 4 は “const int *” 型に対応する QualType を表した図である．

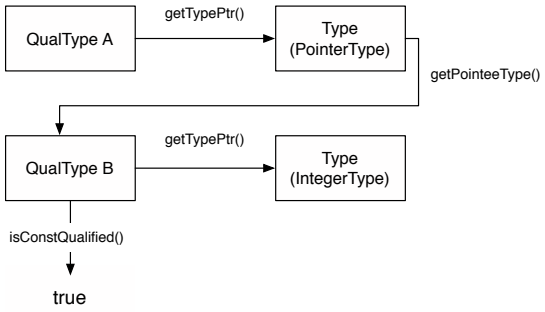

図 4 const int * に対応する QualType

QualType A が const int * 型に対応する．これの持つ getTypePtr 関数を呼び出すことで, PointerType を得ることができる．さらにここで getPointeeType を呼び出すと QualType B が得られる．これの持つ IntegerType には const がかかっているので, QualType B の isConstQualified 関数を呼ぶと true が返る．

このように, clang では複雑な型を持つ関数, 変数でもその型を表すために持つ QualType は一つであり, それが指す Type を辿ることで完全な型を知ることができる．

3.2 Abstract Syntax Tree (AST)

AST はソースコードの解析結果を保持したツリーで, コンパイル時に “-Xclang -ast-dump” というオプションを付加することで表示することもできる．出力された AST の各行が AST のノードとなっており, 各ノードは Decl, Stmt, Expr といったクラスを継承したものになっている．CbC コンパイラの実装ではパーサーが AST を生成する部分に手を加えている．

3.3 LLVM IR

LLVM IR は LLVM BitCode とも呼ばれ, リファレンスが公開されている[1)]．各変数が一度のみ代入される Static Single Assignment (SSA) ベースの言語であり, LLVM 内部で扱うためのメモリ上での形式, 人が理解しやすいアセンブリ言語形式, JIT 上で実行するための bitcode 形式の三種類の形を持ち, いずれも相互変換が可能で同等なものである．ループ構文は存在せず, 一つのファイルが一つのモジュールという単位で扱われる．CbC コンパイラの実装では特に変更を行っていない．

3.4 SelectionDAG

SelectionDAG は LLVM IR が SelectionDAG Instruction Selection Pass によって変換されたものである．SelectionDAG は非巡回有向グラフであり, そのノードは SDNode クラスによって表される．SDNode は命令と, その命令の対象となるオペランドを持つ．SelectionDAG には illegal なものと legal なものの二種が存在し, illigal SelectionDAG の段階ではターゲットがサポートしていない方や命令が残っている．LLVM IR は illegal SelectionDAG, legal SelectionDAG の順に変換されていき, その都度最適化が行われる．CbC コンパイラの実装では, ここで行われる最適化のうちの一つである Tail Call Elimination を code segment に対して強制するように変更を加えている．

3.5 Machine Code

Machine Code は中間言語で無限の仮想レジスタを持つ SSA 形式と物理レジスタを持つ non-SSA 形式がある．LLVM IR より抽象度は低いが, この状態でもまだターゲットに依存しない抽象度を保っている．Machine Code は LLVM 上では MachineFunction, MachineBasicBlock, MachineInstr クラスを用いて

管理される．MachineInstr は一つの命令と対応し，MachineBasicBlock は MachineInstr のリスト，そして MachineFunction が MachineBasicBlock のリストとなっている．CbC コンパイラの実装では特に変更を行っていない．

3.6 MC Layer

MC Layer は正確には中間表現を指すわけではなく，コード生成などを抽象化して扱えるようにした層である．関数やグローバル変数といったものは失われており，MC Layer を用いることで，Machine Code からアセンブリ言語への変換，オブジェクトファイルの生成，JIT 上での実行と言った異なった処理を同一の API を用いて行うことが可能になる．CbC コンパイラの実装では特に変更を行っていない．

4. LLVM/clang 3.5 での CbC コンパイラの実装

以下の節では LLVM と clang に CbC コンパイラを実装する方法について説明する．

以降に示される LLVM, clang のファイルパスについて，$(CLANG) を clang のソースコードを展開したディレクトリのパス，$(LLVM) を LLVM のソースコードを展開したディレクトリのパスとする．

4.1 clang への __code 型の追加

関数が code segment であることを示す __code 型の追加を行うためには __code を予約語として定義する必要がある． clang では予約語は全て $(CLANG)/include/ clang/Basic/TokenKinds.def に定義されており，ここで定義した予約語の頭に kw_ を付けたものがその予約語の ID となる．ここに，図 5 のように変更を加えて __code を追加した．ここで使われている KEYWORD マクロは予約語の定義に用いられるもので，第一引数が登録したい予約語，第二引数がその予約語が利用される範囲を表す．KEYALL は全ての C, C++ でサポートされることを示す．code segment は C のバージョンに関わらずサポートされるべきであるので KEYALL を設定した．

次に clang に __code 型を認識させる． clang では型の識別子の管理に TypeSpecType という enum を用いる． この enum の定義は $(CLANG)/include/clang/Basic/Specifiers.h で行われており，これを図 6 のように編集した．

これに加えてさらに QualType が用いる Type の作成を行う．この定義は $(CLANG)/include/clang/AST/BuiltinTypes.def で行われているので，これを図 7 のように編集した．ここで使用されているマクロには符号付き整数であることを示す SIGNED_TYPE や符号無し整数であることを示す UNSIGNED_TYPE 等があり，それらは BUILTIN_TYPE マクロを拡張するものである．__code 型は符号無し，有りといった性質を保つ必要はなく，また void 型が BUILTIN_TYPE を利用していることから __code 型も BUILTIN_TYPE を使うべきだと判断した．

```
    :
KEYWORD(__func__          , KEYALL)
KEYWORD(__objc_yes        , KEYALL)
KEYWORD(__objc_no         , KEYALL)

#ifndef noCbC // CbC Keywords.
KEYWORD(__code            , KEYALL)
KEYWORD(__return          , KEYALL)
KEYWORD(__environment     , KEYALL)
#endif
    :
```

図 5 TokenKinds.def

```
enum TypeSpecifierType {
  TST_unspecified,
  TST_void,
    :
#ifndef noCbC
   TST___code,
#endif
    :
```

図 6 Specifiers.h

```
    :
// 'bool' in C++, '_Bool' in C99
UNSIGNED_TYPE(Bool, BoolTy)

// 'char' for targets where it's unsigned
SHARED_SINGLETON_TYPE(UNSIGNE
D_TYPE(Char_U, CharTy))

// 'unsigned char', explicitly qualified
UNSIGNED_TYPE(UChar,
UnsignedCharTy)

#ifndef noCbC
BUILTIN_TYPE(__Code, __CodeTy)
#endif
    :
```

図 7 BuiltinTypes.def

これで clang が __code 型を扱えるようになり，code segment を解析する準備が整った．次に __code 型を解析できるようパーサーに変更を加える． clang では型の構文解析は Parser クラスの ParseDeclarationSpecifiers 関数で行われる． この関数の定義は $(CLANG)/lib/Parse/ParseDecl.cpp で行われてお

り, これが持つ巨大な switch 文に kw___code が来た時の処理を加えてやれば良い. 具体的には switch 文内に以下の図 8 ように記述を加えた. ここで重要なのは SetTypeSpecType 関数であり, これによって __code 型が DeclSpec に登録される. DeclSpec は型の識別子を持つためのクラスで, 後に QualType に変換される.

```
case tok::kw___code: {
  LangOptions* LOP;
  LOP = const_cast<LangOptions*>(&getLangOpts());
  LOP->HasCodeSegment = 1;
  isInvalid = DS.SetTypeSpecType(DeclSpec::TST___code, Loc,
                                 PrevSpec, DiagID);
  break;
}
```

図 8 __code の parse

その他の処理について, 最初にある LangOptions はコンパイル時のオプションのうち, プログラミング言語に関わるオプションを管理するクラスであり, このオプションの値を変更しているのはコード内に code segment が存在することを LLVM に伝え, tailcallopt を有効化するためである. LangOptions が管理するオプションは $(CLANG)/include/clang/Basic/LangOptions.def で定義される. ここに以下の図 9 のような変更を加え, HasCodeSegment というオプションを追加した. LANGOPT マクロの引数は第一引数から順にオプション名, 必要ビット数, デフォルトの値, オプションの説明 となっている.

```
    :
#ifndef noCbC
LANGOPT(HasCodeSegment        , 1, 0, "CbC")
#endif
    :
```

図 9 オプションの追加

4.2 LLVM 側での __code 型の追加

LLVM でも clang と同様に __code 型の追加を行う. 型の追加を行うと言っても LLVM IR の持つ type の拡張を行うわけではなく, コンパイル時に内部で code segment であることを知るためだけに型の定義を行い, LLVM IR として出力した場合には 型は void となる. LLVM では型の情報は Type というクラスで管理しており, Type の定義は $(LLVM)/lib/IR/LLVMContextImpl.h で行う. これに加えて TypeID の登録も行う必要があり, これは $(LLVM)/include/llvm/IR/Type.h で定義されている. それぞれ, 以下の図 10, 11 ように編集した.

さらに, __CodeTy は VoidTy としても扱いたいため, 型判別に用いられる isVoidTy 関数の編集も行った. この関数は Type が VoidTy の場合に真を返す関数である. この関数を Type が __CodeTy の場合にも真を返すようにした. ここで変更を行ったのは if 文の条件文のみなので, ソースコードの記載はしない.

```
    :
  // Basic type instances.
  Type VoidTy, LabelTy, HalfTy, FloatTy, DoubleTy,
MetadataTy;
  Type X86_FP80Ty, FP128Ty, PPC_FP128Ty,
X86_MMXTy;
#ifndef noCbC
  Type __CodeTy;
#endif
    :
```

図 10 LLVM での __code の追加

```
enum TypeID {
    :
  StructTyID,     ///< 12: Structures
  ArrayTyID,      ///< 13: Arrays
  PointerTyID,    ///< 14: Pointers
  VectorTyID      ///< 15: SIMD 'packed' format, or
other vector type
#ifndef noCbC
   ,__CodeTyID     /// for CbC
#endif
    :
```

図 11 LLVM での Type ID の追加

4.3 継続のための goto syntax の構文解析

継続のための goto syntax は, goto の後に関数呼び出しと同じ構文が来る形になる. clang が goto 文の構文解析を行っているのは, Parser クラスの ParseStatementOrDeclarationAfterAttributes 関数であり, この関数は $(clang)/lib/Parse/ParseStmt.cpp で定義されている. この関数内にも switch 文があり, この中の kw_goto が来た時の処理に以下の図 12 のように手を加えた.

ifndef, endif マクロで囲まれた部分が追加したコードである. 初めの if 文は, token の先読みを行い, この goto が C の goto 文のためのものなのか, そうでないのかを判断している. C のための goto でないと判断した場合のみ ParseCbCGotoStatement 関数に入り, 継続構文の構文解析を行う. この関数は独自に

```
    :
case tok::kw_goto:
#ifndef noCbC
  // if it is not C's goto syntax
   if (!(NextToken().is(tok::identifier) && PP.LookAhead(1).is(tok::semi)) &&
     NextToken().isNot(tok::star)) {
       SemiError = "goto code segment";
       return ParseCbCGotoStatement(Attrs, Stmts);
    }
#endif
  Res = ParseGotoStatement();
  SemiError = "goto";
  break;
    :
```

図 12　継続を行う goto syntax の構文解析

定義した関数で, その内容を以下の図 13 に示す.

```
StmtResult Parser::ParseCbCGotoStatement(
ParsedAttributesWithRange &Attrs,StmtVector &Stmts) {
  assert(Tok.is(tok::kw_goto) && "Not a goto stmt!");
  ParseScope CompoundScope(this, Scope::DeclScope);
  StmtVector CompoundedStmts;

  SourceLocation gotoLoc = ConsumeToken();  // eat the 'goto'.
  StmtResult gotoRes;
  Token TokAfterGoto = Tok;
  Stmtsp = &Stmts;

  gotoRes = ParseStatementOrDeclaration(Stmts, false);
  if (gotoRes.get() == NULL)
    return StmtError();
// if it is not function call
  else if (gotoRes.get()->getStmtClass() != Stmt::CallExprClass) {
    Diag(TokAfterGoto, diag::err_expected_ident_or_cs);
    return StmtError();
  }

  assert((Attrs.empty() || gotoRes.isInvalid() || gotoRes.isUsable()) &&
         "attributes on empty statement");
  if (!(Attrs.empty() || gotoRes.isInvalid()))
    gotoRes = Actions.ProcessStmtAttributes(gotoRes.get(), Attrs.getList(),
Attrs.Range);
  if (gotoRes.isUsable())
    CompoundedStmts.push_back(gotoRes.release());

  // add return; after goto codesegment();
  if (Actions.getCurFunctionDecl()->getResultType().getTypePtr()
->is__CodeType()) {
    ExprResult retExpr;
    StmtResult retRes;
    retRes = Actions.ActOnReturnStmt(gotoLoc, retExpr.take());
    if (retRes.isUsable())
      CompoundedStmts.push_back(retRes.release());
  }
  return Actions.ActOnCompoundStmt(gotoLoc, Tok.getLocation(),
CompoundedStmts, false);
}
```

図 13　ParseGotoStmt 関数

この関数では, goto の後の構文を解析して関数呼び出しの Stmt を生成する. その後, tail call elimination の条件を満たすために直後に return statement の生成も行う. 関数呼び出しの解析部分は ParseStatementOrDeclaration 関数に任せ, goto の後に関数呼び出しの構文がきていない場合にはエラーを出力する.

4.4　clang/LLVM 間の __code 型の変換

clang での Type が LLVM のものに置き換えられるのは $(CLANG)/lib/CodeGen/CGCall.cpp の GetFunctionType 関数である. ここを以下の図 14 のように編集した. 図のコード中の ABIArgInfo は clang が関数にどのように引数を渡すかの情報を保持するクラスである. void 型ではこれが必ず Ignore になり, 同じことが __code 型にも言える. したがって switch 文内のこの箇所だけで型のチェックを行い, __code 型の時のみ LLVM 側でも __code として扱えば良い.

```
 case ABIArgInfo::Ignore:
#ifndef noCbC
   if (FI.getReturnType().getTypePtr()-
>is__CodeType())
     resultType =
llvm::Type::get__CodeTy(getLLVMContext());
   else
     resultType =
llvm::Type::getVoidTy(getLLVMContext());
#else
   resultType =
llvm::Type::getVoidTy(getLLVMContext());
#endif
   break;
```

図 14　clang/LLVM 間の型変換

4.5　Tail call elimination の強制

Tail call elimination は最適化の一つで, これにより code segment 間の移動を call でなく jmp 命令で行うようになる. 図 15 は Tail call elimination が行われた際のプログラムの処理を表している. caller は funcB を call でなく jmp で呼び出し, funcB は caller でなく main に戻る.

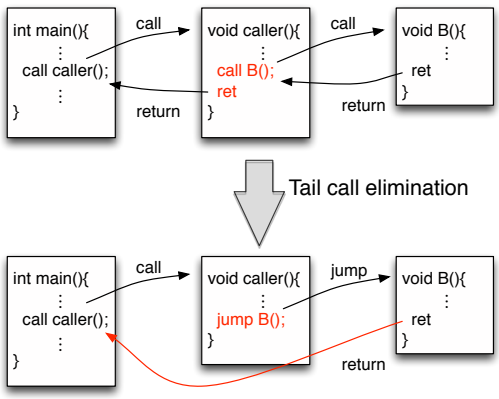

図 15　Tail call elimination

これを強制するためにコンパイラ側では以下の条件

を満たす必要がある．

(1) tail フラグを立てる tail call eliminatoin pass の追加．

(2) 呼び出し元と呼び出す関数の呼び出し規約が fastcc, cc 10 (GHC calling convention), cc 11 (HiPE calling convention) のいずれかである．

(3) 最適化のオプションである tailcallopt が有効になっている．

まず tail call elimination pass 追加の処理について述べる． clang は最適化の pass の追加を $(CLANG)/lib/CodeGen/BackendUtil.cpp の CreatePasses 関数内で行っている．clang では最適化レベルを 2 以上にした場合に tail call elimination が有効化されるが, その pass の追加はこの関数から呼び出される populateModulePassManager 関数で行われる．この関数は LLVM が用意した最適化に用いられる主要な pass を追加するものである．この関数を以下の図 16 のように変更し, 最適化レベルに関わらず追加するようにした．また, createTailCallEliminationPass 関数に対して引数を受け取れるように変更を加え, これによって最適化レベルが低い時に code segment 以外の関数に対して tail call elimination しないようにしている．

```
    :
  if (OptLevel == 0) {
    :
#ifndef noCbC
    MPM.add(createTailCallEliminationPass(true));  // Eliminate tail calls
#endif
    :
  }
    :
#ifndef noCbC
  MPM.add(createTailCallEliminationPass(false));  // Eliminate tail calls
#else
  MPM.add(createTailCallEliminationPass());  // Eliminate tail calls
#endif
    :
```

図 16 pass の追加

これで code segment の呼び出しに対して tail フラグが付与されるようになった．しかし実際にはこれだけでは不十分でさらに二つの pass を追加する必要がある．追加する pass は SROA pass と codeGenPrepare pass である．一つ目の SROA pass は メモリ参照を減らすスカラー置換を行う pass でこれにより LLVM IR の alloca 命令を可能な限り除去できる．tail call elimination の条件に直接記されてはいないが, tail call elimination pass を用いて tail フラグを付与する場合には 呼び出し元の関数に alloca がないことが求められるのである．二つ目の codeGenPrepare pass は名前の通りコード生成の準備を行う pass で, これを通さないと if 文を用いた時に call の直後に配置した return 文が消えてしまう．これら二つの pass も最適化のレベルにかかわらず追加するように変更した．

次に, 呼び出し規約の問題を解消する．条件を満たす呼び出し規約は fastcc, cc 10, cc 11 の三種類があるが, LLVM のドキュメントに cc 10 と cc 11 積極的に利用するのは好ましくないとあった．よって本実験では fastcc を使用する．fastcc を指定すると, 情報をレジスタを用いて渡す等して, 可能な限り高速な呼び出しを試みるようになる．加えて可変引数の使用が不可になり, 呼び出される関数のプロトタイプと呼び出される関数が正確に一致する必要があるという要件を持つ．fastcc の追加は clang が関数情報の設定処理を行っている箇所で行った．関数情報は CGFunctionInfo というクラスを用いて管理される．関数情報の設定は $(CLANG)/lib/CodeGen /CGCall.cpp 内の arrangeLLVMFunctionInfo という関数で行われる．この関数に図 17 に示されるコードを加えた．変数 CC への代入文が fastcc を設定している箇所である．

```
    :
#ifndef noCbC
  if(resultType.getTypePtr()->is__CodeType()){
    if(!required.allowsOptionalArgs())
      CC = llvm::CallingConv::Fast;
  }
#endif
    :
```

図 17 fastcc の付与

最後に, tailcallopt の有効化を行う．clang と LLVM は指定されたオプションを管理するクラスを別々に持っており, clang はユーザーに指定されたオプションを LLVM に引き継ぐ処理を持つ．その処理が行われているのが $(CLANG)/lib/CodeGen/BackendUtil.cpp 内の CreateTargetMachine 関数である．この関数のオプションの引き継ぎを行っている箇所を以下の図 18 のように変更する．tailcallopt は内部では GuaranteedTailCallOpt となっており, code segment を持つ場合にこれを有効化する．また, LLVM 側でも HasCodeSegment というオプションを追加している．これは先ほど述べた codeGenPrepare pass を追加する際に利用する．

LLVM でのオプションの追加方法についてもここで述べておく． LLVM のオプションは TargetOptions というクラスが管理しており， その定義は $(LLVM)/include/llvm/Target/ TargetOp-

```
    :
  Options.PositionIndependentExecutable = LangOpts.PIELevel != 0;
  Options.EnableSegmentedStacks =
CodeGenOpts.EnableSegmentedStacks;
#ifndef noCbC
  Options.HasCodeSegment = LangOpts.HasCodeSegment;
  Options.GuaranteedTailCallOpt = LangOpts.HasCodeSegment;
#endif
    :
```

図 18 tailcallopt の有効化

tions.h で行われている．こちらはマクロは使っておらずビットフィールドを用いて定義されている．TargetOptions クラスの中で変数を宣言するだけで追加できるので, コードは省略する．

4.6 環境付き継続

CbC には通常の C の関数から継続する際, その関数から値を戻す処理への継続を得ることが出来, これを環境付き継続という．これには __return, __environment という特殊変数を用いる．図 19 は環境付き継続の使用例で, この場合 caller が func を呼び出した結果得られる値 0 はでなく 1 となる．

```
__code cs(int retval,__code(*ret)(int,void *),void *env){
  goto ret(n, env);
}

int func (){
  goto cs(1, __return, __environment);
  return 0;
}

int caller (){
  int retval;
  retval = func(); // retval should be 1.
}
```

図 19 環境付き継続の使用例

GCC 上に実装した CbC コンパイラでは nested function を用いていた[2] が, LLVM/clang ではこの拡張構文は対応していない．そこで今回の実装には setjmp, longjmp を用いた実装を行った．__return, __environment が宣言されると, 環境付き継続を用いる関数内での継続前に setjmp の構文が追加され, 環境を保持して継続を行うようになる．このとき, __return には元の環境に戻るための code segment へのアドレスが, __environment には元の環境が保持され, __return の指す code segment に __environment を渡して継続することで元の環境に戻ることができる．__return へ継続することで元の環境に戻ることができるのは __return の指す code segment が longjmp を呼び出すためである．また, 環境付き継続を行う際には戻り値を返すために継続元の関数の型をコピーしなければならないという問題があるが, clang では QualType をコピーするだけで解決する．

setjmp/longjmp を使って C の関数に戻ることは CbC コンパイラとは関係なくプログラムレベルでも実現できる．しかし, 本来は取っておいた C の stack pointer を回復するだけでよく, register の save などを伴う比較的重い作業である setjmp は必要ないはずである．Micro C 実装ではそのように実装されている．将来的により軽い処理で戻れる構文を維持することが望ましいと考えられるので, LLVM/GCC 内部で, C への復帰のコードを提供するようにしている．

5. 評価と考察

評価は, コンパイルして出力されたアセンブリコードの確認と，CbC プログラムを Micro-C，GCC, LLVM/clang でコンパイルして得られたプログラムの実行速度を計測により行う．コンパイル, 計測は x86-64 アーキテクチャの Mac OS X 上で行った．なお, このときの GCC のバージョンは 4.9.0 である．末尾最適化が行われない場合は, 警告が出るようになっており, CbC の仕様を満たしているかどうかは, すぐにわかるようになっている．

5.1 アセンブリコードの評価

以下の図 20,21 はそれぞれコンパイル前の CbC の code segment とコンパイル後, それに対応するアセンブリコードを示している．

```
__code factorial(int x)
{
  goto factorial0(1, x);
}
```

図 20 コンパイル前の code segment

```
_factorial:                             ## @factorial
        .cfi_startproc
## BB#0:                                ## %entry
        subq    $24, %rsp
Ltmp5:
        .cfi_def_cfa_offset 32
        movl    $1, %eax
        movl    %edi, 20(%rsp)          ## 4-byte Spill
        movl    %eax, %edi
        movl    20(%rsp), %esi          ## 4-byte Reload
        addq    $24, %rsp
        jmp     _factorial0             ## TAILCALL
        .cfi_endproc
```

図 21 対応するコード

コンパイル前のコードが持つ factorial0 への継続はコンパイル後, call ではなく jmp 命令により実装されており, このことから tail call elimination が正しく

行われていることがわかる．これより, CbC コンパイラを LLVM/clang 上で実装できたことがわかる．

5.2 実行速度の評価

今回測定に使用したプログラムは conv1 と呼ばれるもので, これは Micro-C での測定や, GCC 上に CbC コンパイラを実装した際の評価にも使用されたものである.☆ conv1 の行う処理は CbC の継続と計算を交互に行うもので, 引数 1 の時には CbC の継続を使用, 引数 2, 3 の時には Micro-C のための最適化が手動で施されたコードを使用するようになっている．測定結果は表 1 に示される．尚, GCC は最適化無しでは末尾最適化を行わないので除外している．この例題は末尾最適化抜きでは stack overflow で動作しない．

	./conv1 1	./conv1 2	./conv1 3
Micro-C	6.875	2.4562	3.105
GCC -O2	2.9438	0.955	1.265
LLVM/clang -O0	5.835	4.1887	5.0625
LLVM/clang -O2	3.3875	2.29	2.5087

表 1 Micro-C, GCC, LLVM/clang の実行速度比較 (単位：秒)

LLVM/clang の最適化の有無で比較すると, 最適化を有効化した方が最適化を用いない場合より全てのケースで二倍以上速い．これは LLVM の最適化の恩恵を受けていることを示していると考えられる．Micro-C と LLVM/clang との実行速度を比較すると, 最適化なしでは LLVM/clang が遅い場合もあるが最適化を有効化すると全ての場合で LLVM/clang が速く, 最適化が大きな効果をもたらしていることがわかる．GCC と比較した場合には速度面では劣るが, LLVM/clang は最適化を無効化しても正常にプログラムが動くという点では優位である．

6. まとめと今後の課題

今回の実装により, LLVM/clang を CbC のコンパイルに利用できるようになった．また, 環境付き継続を GCC の拡張構文である nested function でなく, setjmp/longjmp を用いて実装することに成功した．これより, 今後 nested function をサポートしないコンパイル上に CbC コンパイラを実装する必要がでてきた場合でも実装可能である．

今後の課題の課題として, data segment の設計及び実装と環境付き継続の実装をアセンブリコードを用いて行うことに二つがある．data segment について, code segment が処理を表す単位であるのに対し data segment は処理に必要なデータに対応する．我々は code segment と data segment の組みがひとつの task に相当すると考えており, data segment は, 内部表現と外部表現を持つ, priority を持ち処理順序の切り替えが可能, task の待ち合わせ制御に依存される, といった要件を満たすように設計されるべきであると考えている．

環境付き継続のアセンブリコードを用いた実装について, 今回の実装では setjmp/longjmp を用いたが, インラインアセンブリを用いてアセンブリコードを直接書き出すことで速度の向上が見込めるのではないかと考えている．GCC, LLVM はそれぞれ nested function, setjmp/longjmp と言った機能を利用して環境付き継続を実装しているが, Micro-C では直接対応するアセンブリコードを出力する．この方法だと各アーキテクチャ毎に対応しなければならないという欠点はあるが, 他の実装方法よりも楽に実装でき, 速度も勝るだろうと考えている．

☆ conv1 のソースコードは付録 A に掲載する．

参 考 文 献

1) LLVM Language Reference Manual. *http://llvm.org/docs/LangRef.html*.
2) 与儀健人, 河野真治. Continuation based c コンパイラの gcc-4.2 による実装. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), April 2008.
3) 大城信康, 河野真治. Continuation based c の gcc 4.6 上での実装について. 第 53 回プログラミング・シンポジウム, Jan 2011.
4) LLVM Documentation. *http://llvm.org/docs/index.html*.
5) clang 3.5 documentation. *http://clang.llvm.org/docs/index.html*.
6) clang API Documentation. *http://clang.llvm.org/doxygen/*.

付　　録

A.1 conv1.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  static int loop;
4
5  #if 1 // def __micro_c__
6  #define CC_ONLY 0
7  #else
8  #define CC_ONLY 1
9  #endif
10
11 #ifdef CLANG // for clang/LLVM
12 #define _CbC_return __return
13 #define _CbC_environment __environment
```

```
#endif

typedef char *stack;
#include "conv1.h"

/* classical function call case (0) */
int f0(int i) {
  int k,j;
  k = 3+i;
  j = g0(i+3);
  return k+4+j;
}

int g0(int i) {
  return h0(i+4)+i;
}

int h0(int i) {
  return i+4;
}

#if !CC_ONLY

/* straight conversion case (1) */


struct cont_interface { // General Return Continuation
  __code (*ret)(int,stack);
};

__code f(int i,stack sp) {
  int k,j;
  k = 3+i;
  goto f_g0(i,k,sp);
}

struct f_g0_interface { // Specialized Return Continuation
  __code (*ret)(int,stack);
  int i_,k_,j_;
};

__code f_g1(int j,stack sp);

__code f_g0(int i,int k,stack sp) { // Caller
  struct f_g0_interface *c =
    (struct f_g0_interface *)(sp -= sizeof(struct f_g0_interface));

  c->ret = f_g1;
  c->k_ = k;
  c->i_ = i;

  goto g(i+3,sp);
}

__code f_g1(int j,stack sp) { // Continuation
  struct f_g0_interface *c = (struct f_g0_interface *)sp;
  int k = c->k_;
  sp+=sizeof(struct f_g0_interface);
  c = (struct f_g0_interface *)sp;
  goto (c->ret)(k+4+j,sp);
}

__code g_h1(int j,stack sp);

__code g(int i,stack sp) { // Caller
  struct f_g0_interface *c =
    (struct f_g0_interface *)(sp -= sizeof(struct f_g0_interface));

  c->ret = g_h1;
  c->i_ = i;

  goto h(i+3,sp);
}

__code g_h1(int j,stack sp) { // Continuation
  struct f_g0_interface *c = (struct f_g0_interface *)sp;
  int i = c->i_;
  sp+=sizeof(struct f_g0_interface);
  c = (struct f_g0_interface *)sp;
  goto (c->ret)(j+i,sp);
}

__code h(int i,stack sp) {
  struct f_g0_interface *c = (struct f_g0_interface *)sp;
  goto (c->ret)(i+4,sp);
}

struct main_continuation { // General Return Continuation
  __code (*ret)(int,stack);
  __code (*main_ret)(int,void*);
  void *env;
};

__code main_return(int i,stack sp) {
  if (loop-->0)
    goto f(233,sp);
  printf("#0103:%d\n",i);
  goto (( (struct main_continuation *)sp)->main_ret)(0,
  ((struct main_continuation *)sp)->env);
}

/* little optimzation without stack continuation (2) */

__code f2(int i,char *sp) {
  int k,j;
  k = 3+i;
  goto g2(i,k,i+3,sp);
}

__code g2(int i,int k,int j,char *sp) {
  j = j+4;
  goto h2(i,k+4+j,sp);
}

__code h2_1(int i,int k,int j,char *sp) {
  goto main_return2(i+j,sp);
}

__code h2(int i,int k,char *sp) {
  goto h2_1(i,k,i+4,sp);
}

__code main_return2(int i,stack sp) {
  if (loop-->0)
    goto f2(233,sp);
  printf("#0132:%d\n",i);
  goto (( (struct main_continuation *)sp)->main_ret)(0,
      ((struct main_continuation *)sp)->env);
}

/* little optimizaed case (3) */

__code f2_1(int i,char *sp) {
  int k,j;
  k = 3+i;
```

```
  goto g2_1(k,i+3,sp);
}

__code g2_1(int k,int i,char *sp) {
  goto h2_11(k,i+4,sp);
}

__code f2_0_1(int k,int j,char *sp);
__code h2_1_1(int i,int k,int j,char *sp) {
  goto f2_0_1(k,i+j,sp);
}

__code h2_11(int i,int k,char *sp) {
  goto h2_1_1(i,k,i+4,sp);
}

__code f2_0_1(int k,int j,char *sp) {
  goto (( (struct cont_interface *)sp)->ret)(k+4+
       j,sp);
}

__code main_return2_1(int i,stack sp) {
  if (loop-->0)
    goto f2_1(233,sp);
  printf("#0165:%d\n",i);
  exit(0);
  //goto (( (struct main_continuation *)sp)->
       main_ret)(0,
  //((struct main_continuation *)sp)->env);
}

#define STACK_SIZE 2048
char main_stack[STACK_SIZE];
#define stack_last (main_stack+STACK_SIZE)

#endif

#define LOOP_COUNT 500000000
int
main(int ac,char *av[])
{
#if !CC_ONLY
  struct main_continuation *cont;
  stack sp = stack_last;
#endif
  int sw;
  int j;
  if (ac==2) sw = atoi(av[1]);
  else sw=3;

  if (sw==0) {
    for(loop=0;loop<LOOP_COUNT;loop++) {
      j = f0(loop);
    }
    printf("#0193:%d\n",j);
#if !CC_ONLY
  } else if (sw==1) {
    loop = LOOP_COUNT;
    sp -= sizeof(*cont);
    cont = (struct main_continuation *)sp;
    cont->ret = main_return;
    cont->main_ret = _CbC_return;
    cont->env = _CbC_environment;
    goto f(loop,sp);
  } else if (sw==2) {
    loop = LOOP_COUNT;
    sp -= sizeof(*cont);
    cont = (struct main_continuation *)sp;
    cont->ret = main_return2;
    cont->main_ret = _CbC_return;
    cont->env = _CbC_environment;
    goto f2(loop,sp);
  } else if (sw==3) {
    loop = LOOP_COUNT;
    sp -= sizeof(*cont);
    cont = (struct main_continuation *)sp;
    cont->ret = main_return2_1;
    cont->main_ret = _CbC_return;
    cont->env = _CbC_environment;
    goto f2_1(loop,sp);
#endif
  }
  return 0;
}

/* end */
```