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 コンパイラの実装を行い, LLVM/clang-3.5 への CbC の具体的な実装について述べる.

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

Kaito Tokumori† and Shinji Kono†

We suggest a programming paradigm which use data segments and code segments. We develop CbC which is a lower language of C and uses that programming paradigm. In this study, we implement CbC compiler on LLVM/clang and introduce implemented Continuation based C Compiler on LLVM/clang-3.5.

1. 研 究 目 的

当研究室では, プログラムをコードセグメント, データセグメント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, これは Tail Call Elimination というコンパイラの持つ最適化の強制によって実現される. CbC では継続前の code segment に戻ることはなく, 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, これは 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.

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. GCC 上に CbC コンパイラを実装した理由の一つに, 当時の UNIX 環境における コンパイラの標準が GCC であったからというものがあった. しかし, Mac OS X の最新版である Mavericks では GCC の代わりに LLVM/clang が用いられるようになり, 環境が変わりつつあることがわかる. このような背景から, LLVM/clang を用いて CbC をコンパイルできるのが良いという考えが生じた. 本研究では LLVM/clang 上に CbC コンパイラの実装を行う.

† 琉球大学
University of the Ryukyu

2. Continuation based C (CbC)

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 C と同じであるが, ループ制御や関数コールが取り除かれる.

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. 図 1 は code segment 間の処理の流れを表している.

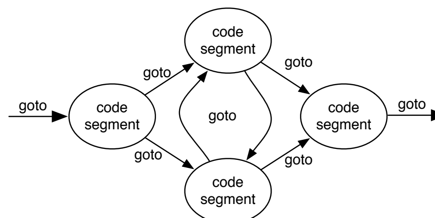

図 1 goto による code segment 間の継続

code segment は C の関数と異なり戻り値を持たず, 処理が終われば次の code segment へと処理を移る. 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 2 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 プログラムである．

```
__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 2　階乗を計算する CbC プログラムの例

3. LLVM/clang で扱われる内部表現

CbC コンパイラの実装の前に, LLVM 及び clang で扱われる内部表現について触れる．LLVM, clang はコンパイル対象コードを Abstract Syntax Tree (AST), LLVM IR, Selection Directed Acycric Graph (SelectionDAG), Machine Code, MCLayer の順に変換し, その後アセンブリ言語へと変換する．図 3 は clang がソースコードを読み込み, アセンブリ言語を出力するまでの流れを表した図である．また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．

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．QualType の持つ Type オブジェクトは getTypePtr 関数を呼び出すことで取

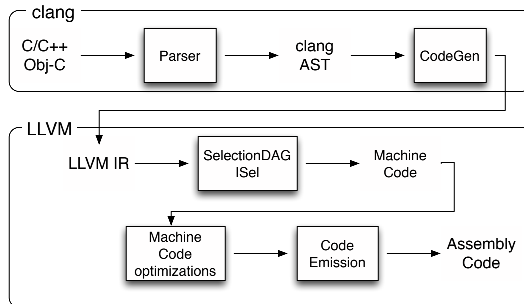

図 3　clang, LLVM によるコンパイルの一連の流れ

得でき, Type クラスは isIntegerType, isVoidType, 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, それを通してポインタの指す型を知ることが可能である．配列や参照等に対しても同様に, それぞれ要素, 参照元の Type へのポインタを持つ QualType を得る関数が存在する．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．

ここで, 以下に一つの例として “const int *” 型に対応する QualType を表した図を示す．

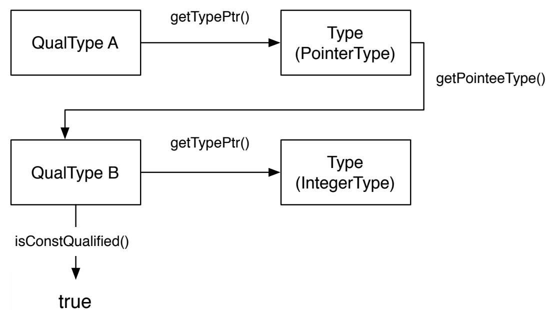

図 4　const int * に対応する QualType

図 4 の QualType A が const int * 型の変数, もしくは関数の持つ QualType である．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．この PointerType がどの型に対するポインタかを知るには前述したとおり getPointeeType を呼び出せば良い．そうして呼び出されたのが QualType B である．この例の QualType は const int * 型に対応するものであるので, ここで取得できた QualType B の getTypePtr 関数を呼び出すと, 当然

IntegerType が得られる．また, この時 int 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ある．AST は "-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 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この 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)]．この言語で記述したプログラムを LLVM 上で実行することも可能である．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 なものの二種類が存在し, 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 LLVM IR よりも機械語に近い形の中間言語であり, 無限の仮装レジスタを持つ 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 での実装

以下の節では LLVM と clang に CbC コンパイラを実装する工程について詳しく説明する．

また, 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 型の追加とその構文解析

まず最初に関数が code segment であることを示す __code 型の追加を行う．そのためには __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．

予約語を定義したことで, clang の字句解析器が各予約語を認識できるようになった．しかし, まだ予約語を認識できるようになっただけで __code という型自体は用意されていない．したがって, 次に clang に __code 型を認識させる必要がある．

clang では型の識別子の管理に TypeSpecType

```
      :
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

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.

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

図 6 Specifiers.h

これに加えてさらに 3.1 節で説明した QualType が用いる Type を作らなければならない. 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.

これで clang が __code 型を扱えるようになり, __code 型の関数, 即ち code segment を解析する準備が整った. よって次に __code 型を解析できるよう clang 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

```
      :
// '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

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.

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

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

図 9 オプションの追加

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

続いて, 継続のための新しい goto syntax の構文解析を行えるようにする．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．したがって, goto の構文解析を行う際にこの構文も解析できるように変更を加える必要がある．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 が来た時の処理に手を加える．具体的には以下の図 12 ように変更した．

```
    :
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

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．ParseCbCGotoStatement 関数は独自に定義した関数で, その内容を以下の図 13 に示す．

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 Tail call elimination の強制

前述したように, CbC の継続の実装には Tail call elimination を用いる．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 14 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．

code segment に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．

```
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

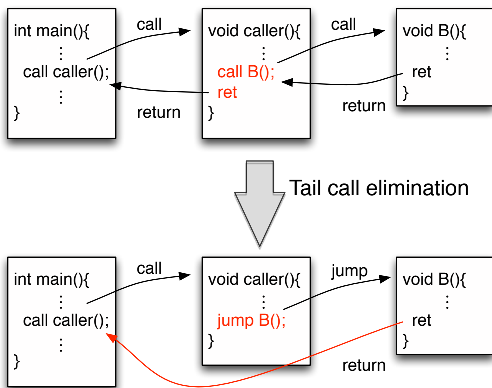

図 14 Tail call elimination

（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 15 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．MPM は pass を管理するクラスのインスタンスで, add 関数によって pass の追加が可能になる．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 15 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．これらの pass 追加されるように変更する必要がある．SROA pass は tail call elimination と同じようにして追加される pass なので同様にして追加することができる．codeGenPrepare

pass はこれら二つとは異なり, addPassesToEmitFile という関数を とおして LLVM によって追加される. この時の追加されるかどうか条件が最適化レベルに依存するものであったため, code segment を持つ場合には最適化レベルに関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 16 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 16 fastcc の付与

最後に, tailcallopt を有効化を行う. 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 17 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/ TargetOptions.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.

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

図 17 tailcallopt の有効化

4.5 環境付き継続

CbC には通常の C の関数からコードセグメント継続する際, その関数から値を戻す処理への継続を得ることができる. これを環境付き継続とい. これには__return, __environment という特殊変数を用いる. GCC 上に実装した CbC コンパイラでは GCC による C の拡張構文である内部関数を用いていたが, clang ではこの拡張構文は対応しておらず, 別の方法をとる必要がある. そこで今回の実装には setjmp, longjmp を用いた実装を行った. 環境付き継続は, __return, __environment が使用された時に自動的にあるコードを生成することで実現される. 具体的には, 図 18 のようなコードを, 内部で図 19 のように解釈することで実現される.

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

int func (){
  goto cs(30, __return, __environment);
  return 0;
}
```

図 18 環境付き継続の使用例

追加された処理は, setjmp ヘッダのインクルード, 環境と戻り値を保持する構造体 CbC_env の定義, 元の環境に戻るための特殊 code segment return1 の定義, これに加えて __return, __environment が環境付き継続の前準備を行う処理に変更される. __return は内部では元の環境に戻るための code segment へのポインタの宣言文と, アドレスの代入を行う文に変換され, __environment は内部では環境を保持する構造体, setjmp, longjmp が用いる jmp_buf, 関数の戻り値と

```
#include <setjmp.h>

struct CbC_env {
  void *ret_p,*env;
};

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

__code return1 (int retval, void* env){
  *(int*)((struct CbC_env *)(env))->ret_p = retcal;
  longjmp((int*)(((struct CbC_env *)env)->env),1);
}

int func (){
   goto code1(30,
       ({
         __code (*__CbC_return)();
            __CbC_return = return1;
            __CbC_return;
       }),
       ({
         struct CbC_env __CbC_environment;
         jmp_buf env;
         int retval;
         __CbC_environment.ret_p = &retval;
         __CbC_environment.env = &env;
         if (setjmp(__CbC_environment.env)){
             return retval;
         }
         &__CbC_environment;
       })
     );
  return 0;
}
```

図 19 内部での解釈

なる retval の宣言文, 構造体のメンバの値の設定を行う代入文, そして特殊 code segment return1 の戻り先となる setjmp を用いた構文に変換される．また, __return, __environment に置き換わる処理は Statement Exprs を利用しており, これによって変数への代入も可能となっている．

5. 評価と考察

今回の研究により実装した LLVM/clang 上での CbC コンパイラの評価を試みる．評価は, CbC プログラムをコンパイルさせ, 出力されたアセンブリコードを確認することで行う．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

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

```
_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

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．

参 考 文 献

1) LLVM Language Reference Manual. *http://llvm.org/docs/LangRef.html.*