LLVM, Clang 上の Continuation based C コンパイラの改良

Improvement of Continuation based C compiler on LLVM and Clang

平成27年度 学位論文(修士)

琉球大学大学院 理工学研究科
情報工学専攻

徳森 海斗

要 旨

Continuation based C (CbC) は本研究室で開発されている code segment, data segment を用いてプログラムを記述する言語である。CbC コンパイラは micro-c ベース、 GCC ベース、 LLVM, clang ベースのものが存在する。LLVM, clang 上に実装された CbC コンパイラはその最適化機能を有効に利用でき、GCC で正しくコンパイル出来ないことのあるコードもコンパイル可能という利点がある反面、実行速度で劣るという面があった。

本研究では LLVM, clang 上に実装された CbC コンパイラに最適化、機能の追加を行った。これにより、以前のバージョンのものよりも高速なアセンブリを出力できるようになる、CbC の記述が楽になる、という結果が得られた。

Abstract

目 次

図 目 次

表 目 次

第1章　研究目的

プログラミングに用いられる単位として関数, クラス, オブジェクト等が存在するが, これらは容易に分割, 結合することは出来ない. また, アセンブリ言語は分割, 結合を行うことは容易であるが, これのみでプログラムを記述することは困難である.

これらの問題を解決するべく, 設計された単位が code segment, data segment である. code segment, data segment は分割, 結合を容易に行うことのできる処理, データの単位として設計されたものであり, 並列プログラミングフレームワーク Cerium[1], 分散ネットワークフレームワーク Alice[2], プログラミング言語 Continuation based C (CbC)[3] はこれらの単位を用いている.

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に加え, 2014 年の研究で LLVM, clang をベースにしたものが存在する. 本研究では, LLVM, clang をベースとした CbC コンパイラにさらなる最適化, 機能の追加, Gears OS の記述をサポートする機能の設計を行った.

第2章　Continuation based C (CbC)

CbC の構文は C と同じであるが, for 文, while 文といったループ制御構文や関数呼び出しを取り除き [注1], code segment と goto による軽量継続を導入している. 以下の図 2.1 は code segment 同士の関係を表したものであり, 図中の丸が code segment を, 矢印が goto による継続を表している.

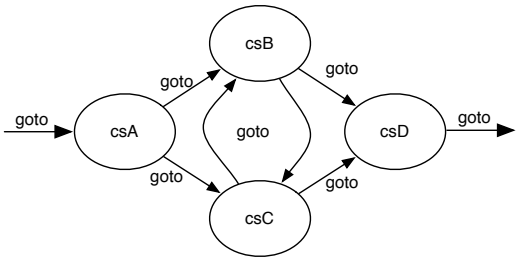

図 2.1: goto による code segment 間の継続

2.1　CbC における Code Segment

CbC では処理の単位として code segment を用いる。code segment は CbC における最も基本的な処理単位であり, C の関数と異なり戻り値を持たない. code segment の宣言は C の関数の構文と同じように行い, 型に __code を用いる. ただし, これは __code 型の戻り値を返すという意味ではない. 前述した通り, code segment は戻り値を持たないので, __code はそれが関数ではなく code segment であることを示すフラグのようなものである. code segment の処理内容の定義も C の関数同様に行うが, 前述した通り CbC にはループ制御構文が存在しないので, ループ処理は自分自身への再帰的な継続を行うことで実現する.

現在の code segment から次の code segment への処理の移動は goto の後に code segment 名と引数を並べて記述するという CbC 独自の構文を用いて行う. この goto による処理の遷移を継続と呼ぶ. 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が, 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

[注1] 言語仕様としては存在しないが while や for を使用することは可能である.

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.

以下に CbC を用いたコードの例として, 与えられた数値の階乗を算出するプログラムを示す. このコードの factorial0 という code segment に注目すると, 条件判別を行い, その結果に応じて自分自身への再帰的な継続を行うか別の code segment への継続を行うかという処理を行っていることがわかる. CbC ではこのようにしてループ処理を制御する.

ソースコード 2.1: 階乗を求める CbC プログラムの例

```
1  __code print_factorial(int prod)
2  {
3     printf("factorial␣=␣%d\n",prod);
4     exit(0);
5  }
6
7  __code factorial0(int prod, int x)
8  {
9     if ( x >= 1) {
10       goto factorial0(prod*x, x-1);
11    }else{
12       goto print_factorial(prod);
13    }
14
15 }
16
17 __code factorial(int x)
18 {
19    goto factorial0(1, x);
20 }
21
22 int main(int argc, char **argv)
23 {
24    int i;
25    i = atoi(argv[1]);
26
27    goto factorial(i);
28 }
```

2.2 環境付き継続

環境付き継続は C との互換性のために必要な機能である. CbC と C の記述を交える際, CbC の code segment から C の関数の呼び出しは問題なく行える. しかし, C の関数から CbC の code segment へと継続する場合, 呼び出し元の環境に戻るための特殊な継続が必要となる. これを環境付き継続と呼ぶ.

この環境付き継続を導入した言語は C with Continuation (CwC) と呼ばれ, C と CbC の両方の機能を持つ言語となる. また, C, CbC は CwC のサブセットと考えられるので, CwC のコンパイラを CbC に使用することができる. これまでに実装されてきた CbC コンパイラは実際には CwC のコンパイラとして実装されている.

環境付き継続を用いる場合, C の関数から code segment へ継続する際に __return, __environment という変数で表される特殊変数を渡す. __return は環境付き継続先が元

の環境に戻る際に利用する code segment, __environment は元の関数の環境を表す. リスト 2.2 では関数 funcB から code segment cs に継続する際に環境付き継続を利用している. cs は funcB から渡された code segment へ継続することで元の C の環境に復帰することが可能となる. 但し復帰先は __return を渡した関数が終了する位置である. このプログラムの例では, 関数 funcA は戻り値として funcB の終わりにある -1 ではなく, 環境付き継続によって渡される 1 を受け取る. 図 2.2 にこの様子を表した.

ソースコード 2.2: 環境付き継続

```
1  __code cs(__code (*ret)(int, void*), void *env){
2    /* C0 */
3    goto ret(1, env);
4  }
5
6  int funcB(){
7    /* B0 */
8    goto cs(__return, __environment);
9    /* B1 (never reached). */
10   return -1;
11 }
12
13 int funcA(){
14   /* A0 */
15   int retval;
16   retval = funcB();
17   /* A1 */
18   printf("retval = %d\n", retval);
19   /* retval should not be -1 but be 1. */
20 }
```

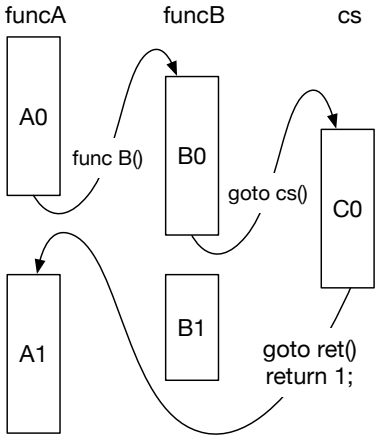

図 2.2: 環境付き継続

このような形にすることで, code segment 側では関数から呼ばれたか, code segment からの継続かを考慮する必要がなくなる. また, funcA から見た場合にも, 呼び出した関数の内部で code segment が使われているかどうかが隠蔽され, code segment の有無を考慮しなくて良い.

2.3 Gears OS サポート

Gears OS は当研究室で開発している並列フレームワークで, CbC で記述している. Gears では通常の CbC には存在しないメタレベルの処理を表す meta code segment, データの単位である data segment, data segment や code segment 等の情報を管理する context 等がある. これらを現在の CbC の機能のみを用いて記述するとリスト 2.3 のようになり, 多くの労力を要する. そのためこの記述を助ける機能が必要であり, 本研究ではこれらを利用するプログラミングをサポートするために以下の機能を提案した.

- code segment から meta code segment への自動接続
- 継続時に context から必要な情報を取得する stub の自動生成
- code segment 内での context の隠蔽

ソースコード 2.3: Gears OS コード例

```
1  __code meta(struct Context* context, enum Code next) {
2      goto (context->code[next])(context);
3  }
4
5  __code code1_stub(struct Context* context) {
6      goto code1(context, &context->data[Allocate]->allocate);
7  }
8
9  __code code1(struct Context* context, struct Allocate* allocate) {
10     allocate->size = sizeof(long);
11     allocator(context);
12     goto meta(context, Code2);
13 }
14
15
16 __code code2(struct Context* context, long* count) {
17     *count = 0;
18     goto meta(context, Code3);
19 }
20
21 __code code2_stub(struct Context* context) {
22     goto code2(context, &context->data[Count]->count);
23 }
```

第3章　LLVM, clang

LLVM とはコンパイラ, 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. 以降は本論文でも, 単に LLVM といった場合は LLVM Core を指す. 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. LLVM を利用する各コンパイラフロントエンドはターゲットとなるプログラミング言語を LLVM IR に変換することで LLVM の持つ最適化機構を利用する.

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. GCC と比較すると丁寧でわかりやすいエラーメッセージを出力する, コンパイル時間が短いといった特徴を持つ.

3.1　clang の基本構造

clang は library-based architecture というコンセプトの元に設計されており, 字句解析を行う liblex, 構文解析を行う libparse といったように処理機構ごとに複数のライブラリに分割されている. clang はこれらのライブラリを与えられた引数に応じて呼び出し, コンパイルを行う. さらに, 必要な場合はリンカを呼び出してリンクを行い, ソースコードを実行可能な状態まで変換することも可能である.

ここで, そのライブラリの中でもコンパイルに関連するものについて説明する.

libast
: Abstract Syntax Tree (AST) や C の型等をクラスとして利用できるようにしたライブラリ. AST の説明は後述する.

liblex
: 字句解析ライブラリ. マクロの展開等の前処理系も担当する.

libparse
: 構文解析ライブラリ. 解析結果を元に後述する libsema を使用して AST を生成する.

libsema
意味解析ライブラリ. parser (libparse) に AST を生成する機能を提供する.

libcodegen
コード生成ライブラリ. 生成された AST を LLVM IR に変換する.

clang
ドライバ. 各ライブラリを用いて求められた処理を行う.

これを踏まえて clang が C のコードを LLVM IR に変換する処理について説明する. 尚 LLVM IR が アセンブリ言語にコンパイルされる処理の過程については 3.2 節で LLVM の基本構造とともに説明する.

以下の図 3.1 は clang が C のコードを LLVM IR に変換する処理の過程を簡潔に図示したものである. clang は C のソースコードを受け取るとまずその解析を libparser による parser を用いて行い, libsema を用いて 解析結果から AST を構築する. そしてその AST を libcodegen を用いて LLVM IR に変換する.

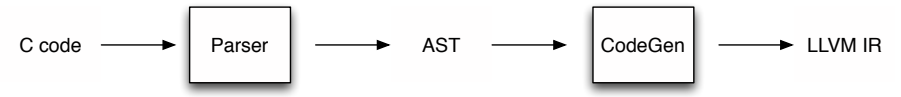

図 3.1: clang の 処理過程

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. 例えばリスト 3.1 コンパイル時にオプション “-Xclang -ast-dump” を付与した場合は出力結果としてリスト 3.2 が得られる. 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. それぞれの簡単な説明を以下に記す.

Decl
宣言や定義を表すクラスであり, 関数の宣言を表す FunctionDecl, 変数の宣言を表す VarDecl 等のサブクラスが存在する.

Stmt
一つの文に対応するクラスであり, if 文と対応する IfStmt, 宣言文と対応する DeclStmt, return 文と対応する ReturnStmt 等のサブクラスが存在する.

Expr
一つの式に対応するクラスであり, 関数呼び出しと対応する CallExpr, キャストと対応する CastExpr 等のサブクラスが存在する.

これらを踏まえて, ソースコード 3.1 と出力された AST(リスト 3.2) に注目する.

1 行目の TranslationUnitDecl が根ノードに当たる. TranslationUnitDecl は翻訳単位を表すクラスであり, この AST が一つのファイルと対応していることがわかる. 実際にソースコードの内容が反映されているのは 5 行目以降のノードで, 5 行目の FunctionDecl がソースコード 3.1 の 1 行目, add 関数の定義部分に当たる. ソースコード 3.1 の 7 行目の add 関数の呼び出しは, AST ではリスト 3.2 の 21 行目, CallExpr で表されている. この CallExpr の下のノードを見ていくと 23 行目の DeclRefExpr が関数のアドレスを持っており, これが add 関数のアドレスと一致することから, CallExpr は呼び出す関数への参照を持っていることがわかる. これらのノード以外についても return 文は ReturnStmt, 変数宣言は VarDecl というように, 各ノードがソースコードのいずれかの部分に対応していることが読み取れる.

ソースコード 3.1: sample.c

```
1 int add(int a, int b){
2   return a + b;
3 }
4
5 int main(){
6   int res;
7   res = add(1,1);
8   return res;
9 }
```

ソースコード 3.2: sample.c の AST

```
1  TranslationUnitDecl 0x102020cd0 <<invalid sloc>>
2  |−TypedefDecl 0x1020211b0 <<invalid sloc>> __int128_t '__int128'
3  |−TypedefDecl 0x102021210 <<invalid sloc>> __uint128_t 'unsigned __int128'
4  |−TypedefDecl 0x102021560 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
5  |−FunctionDecl 0x102021700 <sample.c:1:1, line:3:1> add 'int (int, int)'
6  | |−ParmVarDecl 0x1020215c0 <line:1:9, col:13> a 'int'
7  | |−ParmVarDecl 0x102021630 <col:16, col:20> b 'int'
8  | `−CompoundStmt 0x102021878 <col:22, line:3:1>
9  | `−ReturnStmt 0x102021858 <line:2:3, col:14>
10 | `−BinaryOperator 0x102021830 <col:10, col:14> 'int' '+'
11 | |−ImplicitCastExpr 0x102021800 <col:10> 'int' <LValueToRValue>
12 | | `−DeclRefExpr 0x1020217b0 <col:10> 'int' lvalue ParmVar 0x1020215c0 'a' 'int'
13 | `−ImplicitCastExpr 0x102021818 <col:14> 'int' <LValueToRValue>
14 | `−DeclRefExpr 0x1020217d8 <col:14> 'int' lvalue ParmVar 0x102021630 'b' 'int'
15 `−FunctionDecl 0x1020218f0 <line:5:1, line:9:1> main 'int ()'
16   `−CompoundStmt 0x1020523c0 <line:5:11, line:9:1>
17     |−DeclStmt 0x102052210 <line:6:3, col:10>
18     | `−VarDecl 0x1020219a0 <col:3, col:7> res 'int'
19     |−BinaryOperator 0x102052338 <line:7:3, col:16> 'int' '='
20     | |−DeclRefExpr 0x102052228 <col:3> 'int' lvalue Var 0x1020219a0 'res' 'int'
21     | `−CallExpr 0x102052300 <col:9, col:16> 'int'
22     | |−ImplicitCastExpr 0x1020522e8 <col:9> 'int (*)(int, int)' <FunctionToPointerDecay>
23     | | `−DeclRefExpr 0x102052250 <col:9> 'int (int, int)' Function 0x102021700 'add' 'int
           (int, int)'
24     | |−IntegerLiteral 0x102052278 <col:13> 'int' 1
25     | `−IntegerLiteral 0x102052298 <col:15> 'int' 1
26     `−ReturnStmt 0x1020523a0 <line:8:3, col:10>
27       `−ImplicitCastExpr 0x102052388 <col:10> 'int' <LValueToRValue>
28         `−DeclRefExpr 0x102052360 <col:10> 'int' lvalue Var 0x1020219a0 'res' 'int'
```

AST の他に本研究において重要な clang のクラスに QualType がある. このクラスは後に説明する環境付き継続の実装に関わるクラスである.

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, * (ポインタ) 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.

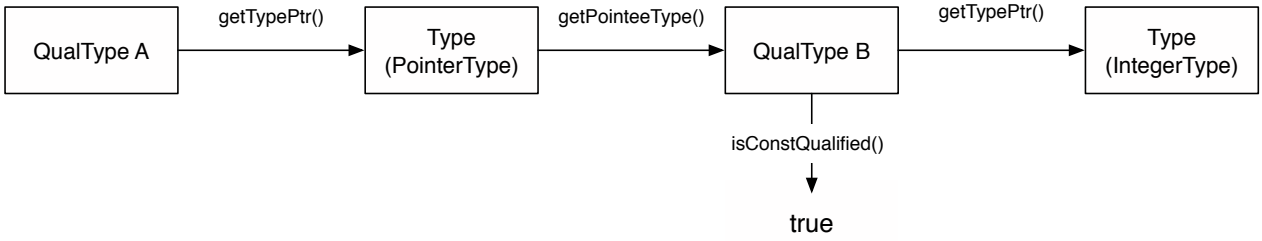

図 3.2: const int * に対応する QualType

図 3.2 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 LLVM の基本構造

LLVM は LLVM IR をターゲットのアセンブリ言語に直接的に変換を行うわけではない. 中間表現を何度か変え, その度に最適化を行い, そして最終的にターゲットのアセンブリ言語に変換するのである. また LLVM では, 最適化や中間表現の変換といったコンパイラを構成する処理は全て pass が行う. 多くの pass は最適化のために存在し, この pass を組み合わせることにより, LLVM の持つ機能の中から任意のものを利用することができる.

LLVM がターゲットのアセンブリ言語を生成するまでの過程を簡潔に記すと以下のようになる.

SelectionDAG Instruction Selection (SelectionDAGISel)
LLVM IR を SelectionDAG (DAG は Directed Acycric Graph の意) に変換し, 最適化を行う. その後 Machine Code を生成する.

SSA-based Machine Code Optimizations
SSA-based Machine Code に対する最適化を行う. 各最適化はそれぞれ独立した pass になっている.

Register Allocation
仮装レジスタから物理レジスタへの割り当てを行う. ここで PHI 命令が削除され, SSA-based でなくなる.

Prolog/Epilog Code Insertion
Prolog/Epilog Code の挿入を行う. どちらも関数呼び出しに関わるものであり, Prolog は関数を呼び出す際に呼び出す関数のためのスタックフレームを準備する処理, Epilog は呼び出し元の関数に戻る際に行う処理である.

Late Machine Code Optimizations
Machine Code に対してさらに最適化を行う.

Code Emission
Machine Code を MC Layer での表現に変換する. その後さらにターゲットのアセンブリ言語へ変換し, その出力を行う.

これらの処理の流れを図示したものが以下の図 3.3 である. 前述した通りこれらの処理は全て pass によって行われる. pass にはいくつかの種類があり, 関数単位で処理を行うもの, ファイル単位で処理を行うもの, ループ単位で処理を行うもの等がある.

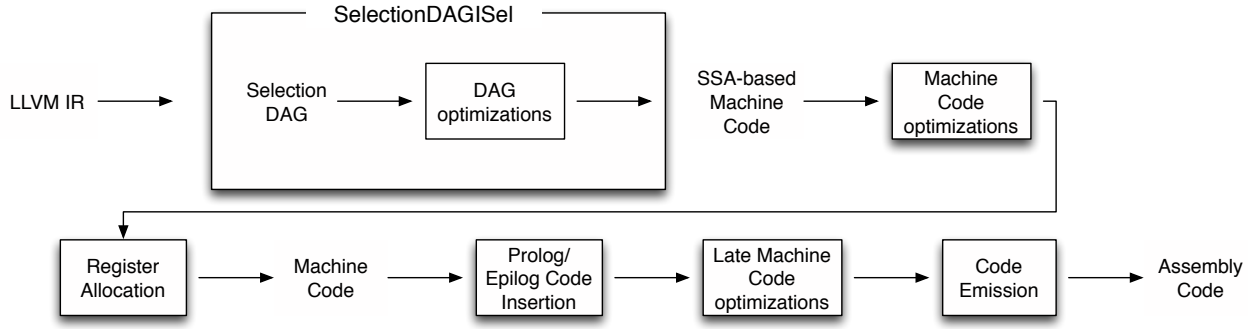

図 3.3: LLVM の 処理過程

3.3 LLVM の中間表現

この節では LLVM の中間表現である LLVM IR, SelectionDAG, Machine Code, MC Layer[注1] と LLVM の最適化について簡単に説明する. なお, 詳しくは LLVM Documantation[4] を参照していただきたい.

LLVM のメインとなる中間表現はフロントエンドの出力, バックエンドの入力に対応する LLVM IR である.

LLVM IR は LLVM BitCode とも呼ばれ, リファレンスが公開されている [5]. 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, コンパイラ中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 (公開されているリファレンスはこの形式に対するものである), 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.

LLVM IR の一例として c 言語の関数を clang を用いて LLVM IR に変換したものをリスト 3.3, 3.4 に示す. LLVM IR に変換された後の関数 test を見ると, while 文によるループ構文がなくなっていることがわかる. while 文は while.cond, while.body という 2 つのブロックに分けられており, while.cond が while 文の条件文, while.body が while 文の中身を表している. while.end は while という名が付いているが, while 文と直接は関係しておらず, これは while 文によるループ処理が終わった後の処理が置き換わったものである.

ソースコード 3.3: c での関数 test

```
1 int test(int a, int b){
2   int i, sum = 0;
3   i = a;
4   while ( i <= b) {
5     sum += i;
6     i++;
7   }
8   return sum − a ∗ b;
9 }
```

ソースコード 3.4: LLVM IR での関数 test

```
1  define i32 @test(i32 %a, i32 %b) #0 {
2  entry:
3    br label %while.cond
4
5  while.cond:
6    %i.0 = phi i32 [ %a, %entry ], [ %inc, %while.body ]
7    %sum.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
8    %cmp = icmp sle i32 %i.0, %b
9    br i1 %cmp, label %while.body, label %while.end
10
11 while.body:
12   %add = add nsw i32 %sum.0, %i.0
13   %inc = add nsw i32 %i.0, 1
14   br label %while.cond
15
```

[注1] MC Layer は正確には中間表現ではない. 詳しくは本節で後述する.

```
16| while.end:
17|   %mul = mul nsw i32 %a, %b
18|   %sub = sub nsw i32 %sum.0, %mul
19|   ret i32 %sub
20| }
```

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 は初め illegal SelectionDAG に変換され, legalization を含む多くの段階を踏んで次の中間表現である Machine Code になる. 以下に SelectionDAG が Machine Code に変換されるまでに行われる処理の過程を示す.

Build initial DAG
LLVM IR を illegal SelectionDAG に変換する.

Optimize
illegal SelectionDAG に対して最適化を行う.

Legalize SelectionDAG Types
ターゲットのサポートしていない型を除去し, ターゲットのサポートする型だけで構成された SelectionDAG に変換する.

Optimize
最適化. 型変換が行われたことで表面化した冗長性の解消を行う.

Legalize SelectionDAG Ops
ターゲットのサポートしていない命令を除去し, ターゲットのサポートする命令だけで構成された SelectionDAG に変換する. これで SelectionDAG の legalization が完了となる.

Optimize
最適化. 命令を変更したことによって発生した非効率性を排除する.

Select instructions from DAG
SelectionDAG を元に, 現在の命令をターゲットのサポートする命令に置き換えた DAG を生成する.

SelectionDAG Scheduling and Formation
命令のスケジューリングを行い, DAG を Machine Code に変換する.

SelectionDAG を確認したい場合は clang に “-mllvm -view-***-dags” オプションを与えることで生成される dot ファイルを見れば良い. *** には legalize などの文字列が入り,

どの段階の DAG を出力するか選択することが出来る. 図 3.4 はリスト 3.3 の add 関数に対応する legalize 直前の DAG である. この図より, + 演算子に対応する add ノードや return 命令に対応するノードその戻り値を受けるためのレジスタが指定されているのがわかる.

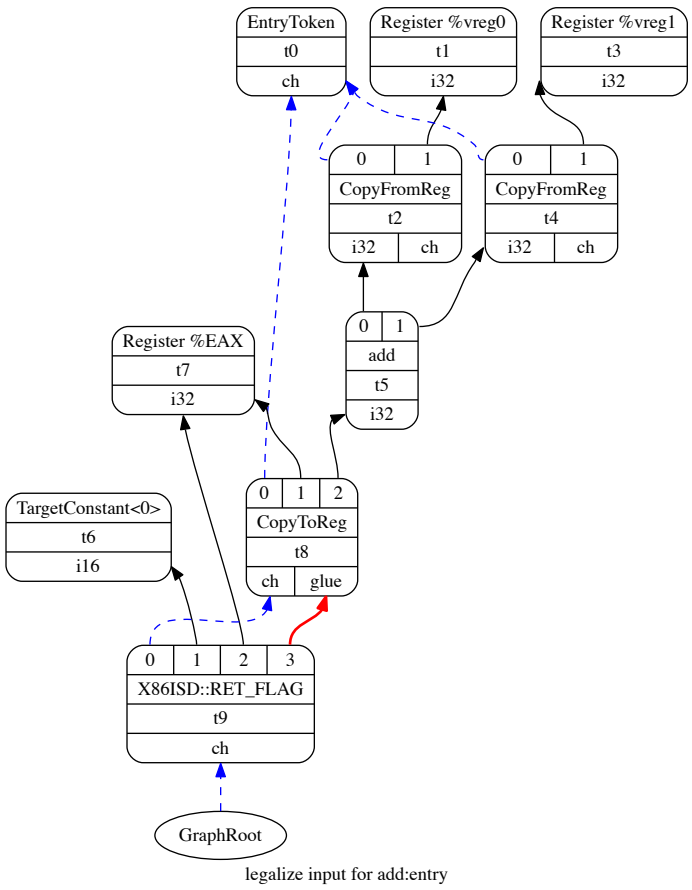

図 3.4: add 関数に対応する legalize 直前の SelectionDAG

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 Single Static Assignment (SSA) 形式と物理レジスタを持つ non-SSA 形式がある. SSA 形式とは全ての変数が一度のみ代入されるように記述した形式であり. この形式を取ることで LLVM は効率よく最適化を行うことが出来る.

Machine Code は 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.

Machine Code の一例を以下のリスト 3.5, 3.6 に示す. リスト 3.5 が SSA 形式, リスト 3.6 が non-SSA 形式であり, 元となるコードはリスト 3.3 である. %varg1, %varg2 といったものが仮想レジスタであり, リスト 3.5 に多く存在することが確認できる. しかし, リスト 3.6 には 1 行目を除いてそれが存在しない. 1 行目はこの関数の引数に対応する物理レジスタと仮想レジスタを並べて表記しているだけなので, ここに仮想レジスタが残っていることについて問題はなく, non-SSA 形式の Machine Code では仮想レジスタが取り除かれていることがわかる.

ソースコード 3.5: Machine Code (SSA)

```
1  Function Live Ins: %EDI in %vreg4, %ESI in %vreg5
2
3  BB#0: derived from LLVM BB %entry
4      Live Ins: %EDI %ESI
5  %vreg5<def> = COPY %ESI
6  %vreg4<def> = COPY %EDI
7  %vreg6<def> = MOV32r0 %EFLAGS<imp-def,dead>
8      Successors according to CFG: BB#1
9
10 BB#1: derived from LLVM BB %while.cond
11     Predecessors according to CFG: BB#0 BB#2
12 %vreg0<def> = PHI %vreg4, <BB#0>, %vreg3, <BB#2>
13 %vreg1<def> = PHI %vreg6, <BB#0>, %vreg2, <BB#2>
14 %vreg7<def,tied1> = SUB32rr %vreg0<tied0>, %vreg5, %EFLAGS<imp-def>
15 JG_4 <BB#3>, %EFLAGS<imp-use>
16 JMP_4 <BB#2>
17     Successors according to CFG: BB#2(124) BB#3(4)
18
19 BB#2: derived from LLVM BB %while.body
20     Predecessors according to CFG: BB#1
21 %vreg2<def,tied1> = ADD32rr %vreg1<tied0>, %vreg0, %EFLAGS<imp-def,dead>
22 %vreg3<def,tied1> = INC64_32r %vreg0<tied0>, %EFLAGS<imp-def,dead>
23 JMP_4 <BB#1>
24     Successors according to CFG: BB#1
25
26 BB#3: derived from LLVM BB %while.end
27     Predecessors according to CFG: BB#1
28 %vreg8<def,tied1> = IMUL32rr %vreg4<tied0>, %vreg5, %EFLAGS<imp-def,dead>
29 %vreg9<def,tied1> = SUB32rr %vreg1<tied0>, %vreg8<kill>, %EFLAGS<imp-def,dead>
30 %EAX<def> = COPY %vreg9
31 RET %EAX
```

ソースコード 3.6: Machine Code (non-SSA)

```
1  Function Live Ins: %EDI in %vreg4, %ESI in %vreg5
2
3  0B BB#0: derived from LLVM BB %entry
4          Live Ins: %EDI %ESI
5  48B %EAX<def> = MOV32r0 %EFLAGS<imp-def,dead>
6  64B %ECX<def> = COPY %EDI
7          Successors according to CFG: BB#1
8
9  96B BB#1: derived from LLVM BB %while.cond
10         Live Ins: %ESI %EDI %ECX %EAX
11         Predecessors according to CFG: BB#0 BB#2
12 144B CMP32rr %ECX, %ESI, %EFLAGS<imp-def>
13 160B JG_4 <BB#3>, %EFLAGS<imp-use,kill>
14 176B JMP_4 <BB#2>
15         Successors according to CFG: BB#2(124) BB#3(4)
```

```
16
17 192B BB#2: derived from LLVM BB %while.body
18           Live Ins: %ESI %EDI %ECX %EAX
19           Predecessors according to CFG: BB#1
20 224B %EAX<def,tied1> = ADD32rr %EAX<kill,tied0>, %ECX, %EFLAGS<imp−def,dead>
21 256B %ECX<def,tied1> = INC64_32r %ECX<kill,tied0>, %EFLAGS<imp−def,dead>
22 304B JMP_4 <BB#1>
23           Successors according to CFG: BB#1
24
25 320B BB#3: derived from LLVM BB %while.end
26           Live Ins: %ESI %EDI %EAX
27           Predecessors according to CFG: BB#1
28 352B %EDI<def,tied1> = IMUL32rr %EDI<kill,tied0>, %ESI<kill>, %EFLAGS<imp−def,dead>
29 384B %EAX<def,tied1> = SUB32rr %EAX<kill,tied0>, %EDI<kill>, %EFLAGS<imp−def,dead>
30 416B RET %EAX
```

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. MC Layer が扱うデータ構造は複数あるが, ここでは MCInst, MCStreamer, MCOperand について説明する.

MCStreamer は アセンブラ API であり, アセンブリファイルの出力や, オブジェクトファイルの出力はこの API を通して行われる. ラベルや .align 等のディレクティブの生成はこの API を利用するだけで可能になる. しかし MCStreamer は機械語に対応する命令は持っておらず, それらの命令を出力するには MCInst クラスを用いる.

MCInst はターゲットに依存しないクラスである. 一つの機械語の命令を表し, 命令とオペランドから構成される.

MCOperand はオペランドに対応し, MCInst はこのクラスを用いる.

MC Layer で用いられる各クラスも “-mllvm -asm-show-inst” オプションを用いることで他の中間表現のように確認することが出来る. MCInst はアセンブリの各命令に対応しているので, アセンブリファイルにコメントとして出力される. リスト 3.7 は 3.3 をコンパイルして得られるアセンブリコードの一部である. 各命令の隣にコメントで記されているのが MCInst, 下に記されているのが MCOperand である.

ソースコード 3.7: アセンブリコードと MCInst

```
1  _add: ## @add
2      .cfi_startproc
3  ## BB#0: ## %entry
4      pushq %rbp ## <MCInst #2300 PUSH64r
5  ## <MCOperand Reg:36>>
6  Ltmp0:
7      .cfi_def_cfa_offset 16
8  Ltmp1:
9      .cfi_offset %rbp, −16
10     movq %rsp, %rbp ## <MCInst #1684 MOV64rr
11 ## <MCOperand Reg:36>
12 ## <MCOperand Reg:44>>
13 Ltmp2:
14     .cfi_def_cfa_register %rbp
15     addl %esi, %edi ## <MCInst #97 ADD32rr
16 ## <MCOperand Reg:23>
```

```
17    ## <MCOperand Reg:23>
18    ## <MCOperand Reg:29>>
19        movl %edi, %eax ## <MCInst #1665 MOV32rr
20    ## <MCOperand Reg:19>
21    ## <MCOperand Reg:23>>
22        popq %rbp ## <MCInst #2178 POP64r
23    ## <MCOperand Reg:36>>
24        retq ## <MCInst #2460 RETQ>
```

3.4 Tail call elimination

前述した通り, LLVM の処理は最適化を含め全て pass によって行われるため, pass を選択することで任意の最適化をかけることができる. 独自の pass を作ることも可能であり, pass 作成のチュートリアルは LLVM のドキュメント [4] にも記されている. また, pass の雛形となるクラスも用意されており, 独自の pass を作成する環境は整っていると言える.

最適化機構は本研究においてはほとんど関係しないが, 継続の実装に関わる Tail Call Elimination, 関数呼び出し時のフレームポインタ操作の最適化を行う omit leaf frame pointer だけは別である.

Tail call elimination の説明の前にまず, tail call について簡単に説明する. tail call は, ある関数の持つ処理の一番最後に位置し, 呼び出し元が戻り値を返す場合は呼び出された関数の戻り値がそのまま呼び出し元の戻り値として返されるような関数呼び出しのことを指す. 具体的には以下のリスト 3.8 の 関数 B, 関数 C の呼び出しがそれに当たる. Tail call elimination はこの末尾にある関数呼び出しの最適化を行う. 通常, 関数呼び出しはアセンブルされると call 命令に置き換わるが, tail call elimination ではその代わりに jmp 命令を用いる. その様子を関数 B の呼び出しに注目し, 関数 caller が main 関数から呼ばれるとして図示したものが以下の図 3.5 である. 通常の関数呼び出しの場合, 関数 caller に呼び出された関数 B はその処理を終えると ret 命令により一度関数 caller に戻る. そして再び ret 命令を用いることで main 関数に戻る. Tail call elimination ではこの関数 B から関数 caller に戻る無駄な処理を低減する. 図 3.5 より, 関数 caller が関数 B を呼び出す時の命令が call から jmp にかわり, 関数 B は処理を終えると直接 main 関数に戻っていることがわかる.

ソースコード 3.8: Tail call の例

```
1 void caller(){
2   A();
3   if ( condition )
4     B();
5     return;
6   else
7     C();
8     return;
9 }
```

では次に, Tail call elimination によって実際にアセンブリコードがどのように変化するかを確認する. この例では x64 形式のアセンブリ命令セットを使用する. リスト 3.8 のコードでは分かり辛いので, 図 3.5 の関数をそれぞれリスト 3.9 のように再定義する.

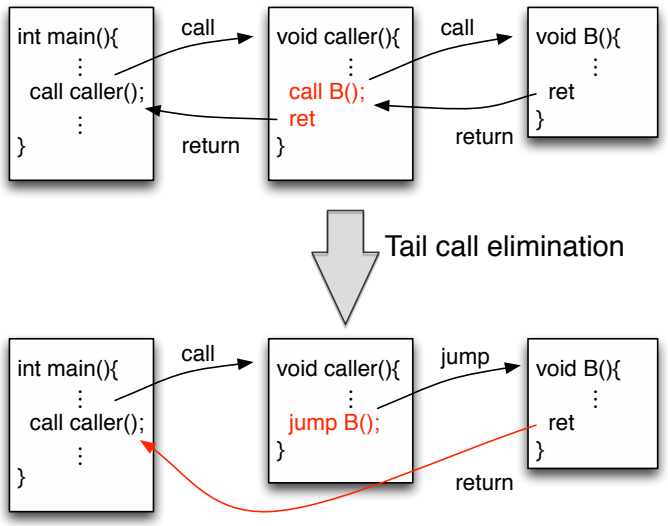

図 3.5: Tail call elimination

ソースコード 3.9: caller, B, main 関数の定義

```
1  void B(int a, int b, int c, int d){
2     return;
3  }
4
5  void caller(int a, int b, int c){
6     B(a, b, c, 40);
7     return;
8  }
9
10 int main(){
11    caller(10, 20, 30);
12    return 0;
13 }
```

これを tail call elimination を行わずにコンパイルしたものが以下のリスト 3.10, tail call elimination を行いコンパイルしたものがリスト 3.11 である. なお, tail call elimination の影響を受けるのは関数 caller のみなのでその部分のみ記載する.

ソースコード 3.10: 関数 caller (tail call elimination 無し)

```
1  _caller: ## @caller
2          .cfi_startproc
3  ## BB#0: ## %entry
4          pushq %rbp
5  Ltmp7:
6          .cfi_def_cfa_offset 16
7  Ltmp8:
8          .cfi_offset %rbp, −16
9          movq %rsp, %rbp
10 Ltmp9:
11         .cfi_def_cfa_register %rbp
12         movl $40, %ecx
13         callq _B
14         popq %rbp
15         ret
16         .cfi_endproc
```

ソースコード 3.11: 関数 caller (tail call elimination 有り)

```
1  _caller: ## @caller
2          .cfi_startproc
3  ## BB#0: ## %entry
4          pushq %rbp
5  Ltmp7:
6          .cfi_def_cfa_offset 16
7  Ltmp8:
8          .cfi_offset %rbp, −16
9          movq %rsp, %rbp
10 Ltmp9:
11         .cfi_def_cfa_register %rbp
12         movl $40, %ecx
13         popq %rbp
14         jmp _B ## TAILCALL
15         .cfi_endproc
```

二つのアセンブリを比較すると, tail call elimination を行った方では call 命令を用いずに jmp 命令で 関数 B に移動していることがわかる. 移動の前に popq を %rbp に対して行っているのはベースポインタの値を戻しておき, 関数 B から直接 main 関数に戻れるようにするためである. 尚, GCC では tail call elimination を行っていない場合には関数呼び出し前にスタックの値を操作する処理が入っていたが, LLVM ではそれが行われていない. これについて, LLVM では tail call elimination を行っていない場合でも関数呼び出しが末尾にある場合には, その関数に戻ることがないことが自明であることから, 現在のスタックに割り当てられたスタックフレームをそのまま使用するようにしているのだろう.

3.5 Tail call elimination の要件

Tail call elimination は 全ての tail call に対して行えるわけではなく, 行うためにはいくつかの条件を満たさなければならない. この条件は LLVM Document[4] に記されている. 現在 LLVM では x86/x86-64, PowerPC を tail call elimination のサポート対象としている. 今回の実装では x86/x86-64 を対象としたので, そちらの条件について考える. x86/x86-64 での tail call elimination の要件は以下のようになっている.

1. 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.
2. 対象となる関数呼び出しが tail call 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.
4. 対象関数が可変長引数を持たない.
5. GOT (Global Offset Table) / PIC (Position Independent Code) を生成する場合, visibility が hidden か protect でなければならない.

これらの条件のうち, コンパイラ側でサポートする必要のある条件について考える. 3 つめの条件にある tailcallopt は有効化することで tail call elimination を行うようになるオプションである. しかし tail call elimination を行うためには 関数呼び出しに対して tail フラグを立てる必要がある. これは Tail Call Elimination pass によって付与されるので, CbC コンパイラではこのパスをオプションに関わらず無条件で追加しなければならない. 4 つめの条件にある可変長引数について, CbC では可変長引数の機能は data segment を用いてサポートすることになるだろう. data segment は現在 CbC には未実装なので今の段階では可変長引数を持つ code segment を作ることは出来ない. 5 つめの条件について, LLVM を用いて PIC を生成する場合には -fPIC というオプションを付加する必要がある. 本論文の主旨から大きく離れるため詳しくは説明しないが, PIC は主に共有ライブラリなどに用いられる, 絶対アドレスにかかわらず正しく実行できるコードの形式, GOT は PIC がアクセスする全グローバル変数のアドレスを格納しているテーブルである. この条件の達成はユーザーに委ねられる.

これらのことから, code segment に対して tail call elimination を行うために達成しなければならない条件は以下のようになることがわかる.

- 呼び出し規約を fastcc, cc 10 (GHC calling convention), cc 11 (HiPE calling convention) のいずれかに指定.
- code segment を tail call にする.
- tailcallopt の有効化.
- 最適化レベルに関わらず Tail call elimination pass を追加.

3.6 omit leaf frame pointer

omit leaf frame pointer は関数呼び出しの際に行われるフレームポインタ操作に関わる最適化である. 通常関数呼び出しを行う際にはフレームポインタの値をスタックに積み, return 直前に戻すという作業を行う. omit leaf frame pointer はこの操作を leaf function を呼び出す際に行わないようにする最適化である.

leaf function とはコールツリーの終端の位置する関数のことを指し, この関数が関数を呼び出さないことを意味する.

以下のリスト 3.12, 3.13 はどちらもアセンブリコードから leaf function の部分を抜き出したものであり, 前者が omit leaf frame pointer 無し, 後者が omit leaf frame pointer ありのコードである. push, pop 命令によるフレームポインタの操作がリスト 3.13 ではなくなっていることがわかる.

ソースコード 3.12: 関数 caller (omit leaf frame pointer 無し)

```
1  _caller: ## @caller
2    .cfi_startproc
3  ## BB#0:
4    pushq^^c2^^bb %rbp
5  Ltmp0:
6    .cfi_def_cfa_offset 16
7  Ltmp1:
8    .cfi_offset %rbp, -16
9    movq^^c2^^bb %rsp, %rbp
10 Ltmp2:
11   .cfi_def_cfa_register %rbp
12   movl^^c2^^bb %edi, -4(%rbp)
13   movl^^c2^^bb %esi, -8(%rbp)
14   movl^^c2^^bb -4(%rbp), %esi
15   addl^^c2^^bb -8(%rbp), %esi
16   movl^^c2^^bb %esi, %eax
17   popq^^c2^^bb %rbp
18   retq
19   .cfi_endproc
```

ソースコード 3.13: 関数 caller (omit leaf frame pointer 有り)

```
1  _caller: ## @caller
2    .cfi_startproc
3  ## BB#0:
4    movl^^c2^^bb %edi, -4(%rbp)
5    movl^^c2^^bb %esi, -8(%rbp)
6    movl^^c2^^bb -4(%rbp), %esi
7    addl^^c2^^bb -8(%rbp), %esi
8    movl^^c2^^bb %esi, %eax
9    retq
10   .cfi_endproc
```

CbC の code segment は関数呼び出しでなく継続によって処理の遷移を行っていくため, 全ての code segment が leaf function ということになる. したがって CbC はこの最適化と非常に相性が良いと言え, 本研究ではこの最適化の強制も行った.

第4章　LLVM, clang 上での CbC の実装

第 2, 3 章をもとに LLVM, clang への CbC コンパイル機能の実装を行う．コードセグメントの解釈, 軽量継続などいくつかの機能は過去の研究によって実装されたが, 本研究での実装に繋がるので説明する．

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　code segment

code segment は __code 型の関数のように扱えるよう実装する．また, code segment は戻り値を返さないので void 型を参考に変更を加えていく．最初に関数が code segment であることを示す __code 型の追加を行う．これは clang, llvm 別々で行う必要がある．clang 側の作業は __code を予約語として定義する改変から始める．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, 次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し, この他には C++ の予約語であることを示す KEYCXX や C++11 以降で利用されることを示す KEYCXX11 等がある．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 を選択した．ここで環境付き継続に用いる予約語である __return, __environment の定義も同時に行う．

ソースコード 4.1: TokenKinds.def

```
1        :
2  KEYWORD(__func__ , KEYALL)
3  KEYWORD(__objc_yes , KEYALL)
4  KEYWORD(__objc_no , KEYALL)
5
6  #ifndef noCbC // CbC Keywords.
7  KEYWORD(__code , KEYALL)
8  KEYWORD(__return , KEYALL)
9  KEYWORD(__environment , KEYALL)
10 #endif
```

```
11          :
```

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 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, これを以下のように編集した.

ソースコード 4.2: Specifiers.h

```
1 enum TypeSpecifierType {
2    TST_unspecified,
3    TST_void,
4          :
5 #ifndef noCbC
6       TST___code,
7 #endif
8          :
```

これに加えてさらに 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, これを以下のように編集した. 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.

ソースコード 4.3: BuiltinTypes.def

```
1          :
2 // 'bool' in C++, '_Bool' in C99
3 UNSIGNED_TYPE(Bool, BoolTy)
4
5 // 'char' for targets where it's unsigned
6 SHARED_SINGLETON_TYPE(UNSIGNED_TYPE(Char_U, CharTy))
7
8 // 'unsigned char', explicitly qualified
9 UNSIGNED_TYPE(UChar, UnsignedCharTy)
10
11 #ifndef noCbC
12 BUILTIN_TYPE(__Code, __CodeTy)
13 #endif
14          :
```

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. この関数のもつ巨大な 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 文内に以下のように記述を加えた. また, この関数の定義は $(CLANG)/lib/Parse/ParseDecl.cpp で行われている.

ソースコード 4.4: __code の parse

```
1  case tok::kw___code: {
2    LangOptions* LOP;
3    LOP = const_cast<LangOptions*>(&getLangOpts());
4    LOP->HasCodeSegment = 1;
5    isInvalid = DS.SetTypeSpecType(DeclSpec::TST___code, Loc, PrevSpec, DiagID);
6    break;
7  }
```

重要なのは 5 行目で, ここで __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あり, 後に QualType に変換される.

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. これを以下のリスト 4.5 のように変更して 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.

ソースコード 4.5: LangOptions の追加

```
1        :
2 LANGOPT(ApplePragmaPack, 1, 0, "Apple␣gcc-compatible␣#pragma␣pack␣handling")
3
4 BENIGN_LANGOPT(RetainCommentsFromSystemHeaders, 1, 0, "retain␣documentation␣
      comments␣from␣system␣headers␣in␣the␣AST")
5
6 #ifndef noCbC
7 LANGOPT(HasCodeSegment , 1, 0, "CbC")
8 #endif
9        :
```

ここまでの変更により clang が正しく __code を解釈し, code segment を AST に変換できるようになる. 次に LLVM 側の変更を行う.

LLVM でも clang と同様に __code 型の追加を行うが, 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う. そのため LLVM IR への変更は一切なく, 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. ここで変更されたは if 文の条件文のみなので, ソースコードの記載はしない.

ソースコード 4.6: LLVMContextImpl.h

```
1        :
2   // Basic type instances.
```

```
3    Type VoidTy, LabelTy, HalfTy, FloatTy, DoubleTy, MetadataTy;
4    Type X86_FP80Ty, FP128Ty, PPC_FP128Ty, X86_MMXTy;
5  #ifndef noCbC
6    Type __CodeTy;
7  #endif
8          :
```

ソースコード 4.7: Type.h

```
1    enum TypeID {
2          :
3      StructTyID, ///< 12: Structures
4      ArrayTyID, ///< 13: Arrays
5      PointerTyID, ///< 14: Pointers
6      VectorTyID ///< 15: SIMD 'packed' format, or other vector type
7  #ifndef noCbC
8      ,__CodeTyID /// for CbC
9  #endif
10         :
```

これらの編集により, LLVM 側でも code segment を認識できるようになった.

4.2　軽量継続

CbC で軽量継続は goto に code segment 冥を添えることで行う. この新しい goto syntax を追加する. 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. 具体的には以下のように変更した.

ソースコード 4.8: goto 文の構文解析

```
1          :
2  case tok::kw_goto: // C99 6.8.6.1: goto−statement
3  #ifndef noCbC
4    if (!(NextToken().is(tok::identifier) && PP.LookAhead(1).is(tok::semi)) && // C: 'goto' identifier
         ';'
5      NextToken().isNot(tok::star)) { // C: 'goto' '*' expression ';'
6        SemiError = "goto␣code␣segment";
7        return ParseCbCGotoStatement(Attrs, Stmts); // CbC: goto codesegment statement
8      }
9  #endif
10   Res = ParseGotoStatement();
11   SemiError = "goto";
12   break;
13         :
```

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, その内容を以下のリスト 4.9 に示す.

ソースコード 4.9: ParseCbCGotoStatement

```
1  StmtResult Parser::ParseCbCGotoStatement(ParsedAttributesWithRange &Attrs,StmtVector &
       Stmts) {
2    assert(Tok.is(tok::kw_goto) && "Not␣a␣goto␣stmt!");
3    ParseScope CompoundScope(this, Scope::DeclScope);
4    StmtVector CompoundedStmts;
5
6    SourceLocation gotoLoc = ConsumeToken(); // eat the 'goto'.
7    StmtResult gotoRes;
8    Token TokAfterGoto = Tok;
9    Stmtsp = &Stmts;
10
11   gotoRes = ParseStatementOrDeclaration(Stmts, false);
12   if (gotoRes.get() == NULL)
13     return StmtError();
14   else if (gotoRes.get()->getStmtClass() != Stmt::CallExprClass) { // if it is not function call
15     Diag(TokAfterGoto, diag::err_expected_ident_or_cs);
16     return StmtError();
17   }
18
19   assert((Attrs.empty() || gotoRes.isInvalid() || gotoRes.isUsable()) &&
20          "attributes␣on␣empty␣statement");
21   if (!(Attrs.empty() || gotoRes.isInvalid()))
22     gotoRes = Actions.ProcessStmtAttributes(gotoRes.get(), Attrs.getList(), Attrs.Range);
23   if (gotoRes.isUsable())
24     CompoundedStmts.push_back(gotoRes.release());
25
26   // add return; after goto codesegment();
27   if (Actions.getCurFunctionDecl()->getResultType().getTypePtr()->is__CodeType()) {
28     ExprResult retExpr;
29     StmtResult retRes;
30     retRes = Actions.ActOnReturnStmt(gotoLoc, retExpr.take());
31     if (retRes.isUsable())
32       CompoundedStmts.push_back(retRes.release());
33   }
34   return Actions.ActOnCompoundStmt(gotoLoc, Tok.getLocation(), CompoundedStmts, false);
35 }
```

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.

4.3 Tail call elimination pass の条件の達成

tail call elimination を強制するためにその条件を満たす処理の追加を行う. 3.5 節で述べた条件のうち, code segment を tail call にする (呼び出した直後に return 文がくる) という条件は前節で達成した. したがって残りの条件は, 呼び出し規約を fastcc, cc 10, cc 11 のいずれかにする, tailcallopt の有効化, tail call elimination pass の追加の三つである.

まず初めに, tail call elimination pass の追加を行う. 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. この関数を以下のリスト 4.10 のように変更した. 変数 MPM は PassManagerBase という pass の管理を行うクラスのインスタンスで, MPM の持つ add 関数を用いて pass の登録を行う. add 関数の引数に createTailCallEliminationPass 関数を指定することで tail call elimination pass が追加され, リスト 4.10 の 5, 11, 13 行目がその処理を行っている. この中で書き加えられたのは 5 行目と 11 行目のものである. 二箇所に記述しているのはこの関数が最適化レベルが 0 かどうかによって追加する pass を変えるためである. CbC コンパイラを実装するためにはどちらの場合でも tail call elimination pass が必要となるので二箇所に記述している. 5 行目 最適化レベルが 0 の場合, 11 行目がそれ以外の場合ののための記述である. このとき, createTailCallEliminationPass 関数の引数は, tail call elimination を code segment のみに適用するかどうかを表す. 最適化レベルが 0 の場合に code segment 以外の関数に触れないようにするためにこのような引数を設けた.

ソースコード 4.10: tail call elimnation pass の追加

```
1         :
2   if (OptLevel == 0) {
3         :
4 #ifndef noCbC
5     MPM.add(createTailCallEliminationPass(true)); // Eliminate tail calls
6 #endif
7         :
8   }
9         :
10 #ifndef noCbC
11   MPM.add(createTailCallEliminationPass(false)); // Eliminate tail calls
12 #else
13   MPM.add(createTailCallEliminationPass()); // Eliminate tail calls
14 #endif
15        :
```

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 の三種類があり, それぞれ簡単に説明すると以下の様な特徴がある.

fastcc

この規約を指定すると, 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. この呼び出し規約は可変引数をサポートせず, 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.

cc 10

Glasgow Haskell Compiler のために実装された呼び出し規約. X86 でのみサポートされており, 引数に関するいくつかの制限をもつ. レジスタ内の値を全て渡し, 呼び出された関数はレジスタの値を保存できない. この呼び出し規約は関数型プログラミング言語を実装する際に使用される register pinning という技術の代わりとして特定の状況でしか使用してはならない.

cc 11

High-Performance Erlang のために実装された呼び出し規約. 通常の C の呼び出し規約以上に引数を渡すためにレジスタを利用する. X86 でのみサポートされており, cc 10 同様に register pinning を用いる.

これらのうち, cc 10, cc 11 は関数型プログラミング言語の実装に用いられる register pinning という技術の代わりに用いられ, これを積極的に利用することは好ましくないとある. 対して fastcc には, 可変引数をサポートしない, プロトタイプは正確でなければならないといった条件があるが, 前者は 3.5 節で説明したとおり tail call elimination の条件に含まれており, 後者は特別厳しい条件ではない上, プロトタイプの不正な関数は clang がエラーを出してくれるので問題ないだろう. よって code segment の呼び出し規約には fastcc を用いることにした.

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. この関数内に以下のリスト 4.11 に示されるコードを加えた. 5 行目が fastcc を設定している箇所である. 関数が __code 型の場合で, かつ可変長引数を持たない場合に呼び出し規約を fastcc に設定する. 可変長引数に関する条件を加えているのは, 可変長引数が使用されている場合には呼び出し規約を変えず tail call elimination を行わないことで通常の関数呼び出しを生成するためである.

ソースコード 4.11: fastcc の追加

```
1         :
2 #ifndef noCbC
3   if(resultType.getTypePtr()->is__CodeType()){
4     if(!required.allowsOptionalArgs())
5       CC = llvm::CallingConv::Fast;
6   }
7 #endif
8         :
```

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リスト 4.12 のように変更する. 6 行目が tailcallopt を有効にしている箇所である. 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. 右辺の HasCodeSegment が code segment を持つか否かを表し, これは予約語 __code を解析する際に有効化される. また, 5 行目からわかるように LLVM 側でも HasCodeSegment というオプションを追加している. これは 4.3 節で述べた codeGenPrepare pass を追加する際に利用する.

ソースコード 4.12: オプションの引き継ぎ

```
1         :
2   Options.PositionIndependentExecutable = LangOpts.PIELevel != 0;
3   Options.EnableSegmentedStacks = CodeGenOpts.EnableSegmentedStacks;
4 #ifndef noCbC
5   Options.HasCodeSegment = LangOpts.HasCodeSegment;
6   Options.GuaranteedTailCallOpt = LangOpts.HasCodeSegment;
7 #endif
8         :
```

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.

4.4　環境付き継続

環境付き継続を行うためには __return, __environment という特殊変数を用いる. これら二つの変数の実装方法には様々な方法が考えられる. 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. しかし, 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. 過去の研究においてこの機能は実装されたが, その際には 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, リスト 4.13 のようなコードをコンパイルした場合にリスト 4.14 のように解釈することで実現される. この置き換えられるコードについて, 以前の実装では setjmp, longjmp を用いていたが, 今回 llvm builtin の setjmp, longjmp を用いるように変更を加えた.

ソースコード 4.13: 環境付き継続の例

```
1 __code cs(int retval,__code(*ret)(int,void *),void *env){
2   goto ret(n, env);
3 }
4
5 int func (){
6   goto cs(30, __return, __environment);
7   return 0;
```

```
8 }
```

ソースコード 4.14: 内部での解釈

```
1  #include <setjmp.h>
2
3  struct CbC_env {
4    void *ret_p,*env;
5  };
6
7  __code cs(int retval,__code(*ret)(int,void *),void *env){
8    goto ret(n, env);
9  }
10
11 __code return1 (int retval, void* env){
12   *(int*)((struct CbC_env *)(env))->ret_p = retcal;
13   __builtin_longjmp((int*)(((struct CbC_env *)env)->env),1);
14 }
15
16 int func (){
17   __code (*__return)();
18   struct CbC_env __environment;
19   jmp_buf env;
20   int retval;
21   __environment.ret_p = &retval;
22   __environment.env = &env;
23   __return = return1;
24     if (__builtin_setjmp(__environment.env)){
25       return retval;
26     }
27   goto code1(30, __return, &__environment);
28   return 0;
29 }
```

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, そして関数 func 内の 17 行目から 26 行目までの処理であり, 27 行目の継続の第三引数も参照渡しに変更される. 追加された処理の大まかな流れを説明する. 環境付き継続を用いる関数は継続前に __builtin_setjmp を用いて戻り先の環境を保存し, 継続を行う. このとき, 引数として渡される __return には特殊 code segment return1 のアドレス, __environment の持つメンバには戻り値として返される値を持つ変数のアドレスと, __builtin_setjmp により保存された継続前の環境が保持されている. 元の環境に戻るための code segment は自動生成されるので, 継続先の code segment は返す戻り値と __environemnt を引数として __return の指す code segment を呼び出すだけで元の環境に戻れる. return1 は 構造体の持つ戻り値へのポインタを利用して戻り値を更新した後, __builtin_longjmp を用いて元の環境に戻る. __builtin_longjmp によって再度 __builtin_setjmp の呼び出しに戻るが今回は if 文内に入り, 戻り値を返す.

自動生成される文をまとめるといかのようになる.

- __return
  - 元の環境に戻るための特殊な code segment
  - 各変数の宣言及び代入文

- __environment
  - 環境を保持する構造体 CbC_env の定義
  - 各変数の宣言及び代入文
  - __builtin_setjmp を条件文にとり, 真の場合に関数の戻り値を返す if 文
- どちらにも依存しない
  - setjmp.h のインクルード文

setjmp.h のインクルード文がどちらにも属さないのは, 無条件でこのヘッダをインクルードするようにしてもよいからである. 必要のないコードは最適化によって除去されるので, 使用されない場合でもインクルードして構わない. また, これらの文を生成する際, 元の関数の型の戻り値を返せるようにしなければならないが, 3.1 節で説明したように, clang では型情報を QualType が管理しているので, 生成する関数や変数に対して元の関数の返り値に対応する QualType を与えるだけでこの問題は解決する. 尚, 二つの予約語を登録する処理は既に説明したので省く.

まずはじめに setjmp.h のインクルード文の生成について説明する. clang では include マクロによって他のファイルのインクルードを行う場合, 字句解析対象となるのファイルの変更を行う. 本研究では, これらの処理に必要なものを IncludeHeader という一つの関数にまとめ, それを呼び出すことで任意のファイルをインクルードできるようにした. その関数のうち, 重要な部分を抜き出したものを以下のリスト 4.15 に示す. この関数は引数に現在の token とインクルードするファイルの名前を求める. まずこの関数は, 現在の token の保存処理を行う. これは, この関数が呼ばれた時点で取得された token はすでに字句解析対象のバッファから取り除かれており, これを保存しておかないとインクルード処理終了後の構文解析に影響が出るために行っている. 次に, 指定されたファイルの検索を行う. その処理を行っているのは 11 行目であり, このときファイルが見つからなかった場合には NULL が返るので, これを利用してエラーの生成も行っている. 解析対象の変更を行っているのが 16 行目以降の処理で, 実際にファイルを移っているのは 18 行目の EnterSourceFile 関数である. この関数によって字句解析対象が現在のファイルから指定したファイルへと変更される. また, この関数の戻り値はファイルのインクルード処理を行ったかどうかを判断するために用いる.

ソースコード 4.15: IncludeHeader 関数

```
1  bool Preprocessor::IncludeHeader(Token Tok, const char* Name) {
2    if (SavedTokenFlag) // If the lexer has already entered a header file, we have to leave this function.
3      return false;
4    SourceLocation Loc = Tok.getLocation();
5    SavedToken = Tok;
6    SavedDepth = IncludeMacroStack.size();
7    SavedTokenFlag = true;
8
9          :
10
11   const FileEntry *File = LookupFile(Loc, Filename, isAngled, LookupFrom, CurDir, NULL, NULL
         ,
```

```
12                                          HeaderInfo.getHeaderSearchOpts().ModuleMaps ? &
                                                SuggestedModule : 0);
13
14          :
15
16    FileID FID = SourceMgr.createFileID(File, Loc, FileCharacter);
17    EnterSourceFile(FID, CurDir, FilenameTok.getLocation(), static_cast<bool>(BuildingModule));
18    return true;
19  }
```

次に, 元の環境に戻るための特殊な code segment の定義生成処理の説明をする. これを行うためには clang の持つ AST に関数の定義を加えてやれば良い. 今回の実装でその処理を担うのが以下のリスト 4.16 に示される CreateRetCS 関数である. ただしここでは重要でない箇所を省いている. この関数は引数として生成する関数の名前を受け取る. 2, 3 行目が元の関数の QualType を取得する処理である. 5 行目から 14 行目までの処理は関数宣言のためにスコープをグローバルスコープに変更するための処理で, すべての処理を終えた後, 59 行目から 62 行目の処理によって元のスコープに戻す. 16 行目から 42 行目までが関数のプロトタイプ宣言に関する設定処理を行っており, 29 行目で 取得しておいた QualType を利用していることがわかる. 44 行目からが関数の持つ処理の定義を指定する部分にあたり, 52 行目にある StmtVector クラスのインスタンスに対して任意の文に対応する StmtResult を追加し, 54 行目の関数を呼び出すと戻り値としてこの関数の中身を表す StmtResult を得ることができる. その後これを定義や宣言を管理するクラスである Decl へと変換し, 58 行目の関数を呼び出すことで生成した関数の定義が AST に追加される.

ソースコード 4.16: CreateRetCS 関数

```
1  bool Parser::CreateRetCS(IdentifierInfo *csName){
2    FunctionDecl *CurFunctionDecl = Actions.getCurFunctionDecl();
3    QualType CurFuncResQT = CurFunctionDecl->getResultType();
4
5    Scope *SavedScope = getCurScope();
6    DeclContext *SavedContext = Actions.CurContext;
7    TypeSourceInfo *CurFuncTI = Actions.Context.CreateTypeSourceInfo(CurFuncResQT);
8    sema::FunctionScopeInfo *SavedFSI = Actions.FunctionScopes.pop_back_val();
9
10   Actions.CurContext = static_cast<DeclContext *>(Actions.Context.getTranslationUnitDecl());
11   Scope *TopScope = getCurScope();
12   while(TopScope->getParent() != NULL)
13     TopScope = TopScope->getParent();
14   Actions.CurScope = TopScope;
15
16   DeclGroupPtrTy returnDecl = DeclGroupPtrTy();
17   ParsingDeclSpec PDS(*this);
18   setTST(&PDS, DeclSpec::TST___code);
19   ParsingDeclarator D(*this, PDS, static_cast<Declarator::TheContext>(Declarator::FileContext));
20   D.SetIdentifier(csName, Loc);
21   ParseScope PrototypeScope(this,Scope::FunctionPrototypeScope|Scope::DeclScope|Scope::
         FunctionDeclarationScope);
22   SmallVector<DeclaratorChunk::ParamInfo, 16> ParamInfo;
23   DeclSpec FDS(AttrFactory);
24   ParmVarDecl *Param;
25
26   IdentifierInfo *retvalII = CreateIdentifierInfo(__CBC_RETVAL_NAME, Loc);
27   Param = CreateParam(retvalII);
```

```
28    Param->setTypeSourceInfo(CurFuncTI);
29    Param->setType(CurFuncResQT);
30
31    ParamInfo.push_back(DeclaratorChunk::ParamInfo(retvalII, Loc, Param, 0));
32    IdentifierInfo *envII = CreateIdentifierInfo(__CBC_STRUCT_ENV_NAME, Loc);
33    Param = CreateParam(envII, 1, DeclSpec::TST_void);
34    ParamInfo.push_back(DeclaratorChunk::ParamInfo(envII, Loc, Param, 0));
35
36    D.AddTypeInfo(DeclaratorChunk::getFunction(HasProto, IsAmbiguous, Loc, ParamInfo.data(),
          ParamInfo.size(), EllipsisLoc, Loc,
37                                                FDS.getTypeQualifiers(), RefQualifierIsLValueRef,
                                                      RefQualifierLoc, ConstQualifierLoc,
38                                                VolatileQualifierLoc, SourceLocation(), ESpecType,
                                                      ESpecRange.getBegin(),
39                                                DynamicExceptions.data(), DynamicExceptionRanges.
                                                      data(), DynamicExceptions.size(),
40                                                NoexceptExpr.isUsable() ? NoexceptExpr.get() : 0,
41                                                Loc, Loc, D, TrailingReturnType), FnAttrs, Loc);
42    PrototypeScope.Exit();
43
44    Decl *TheDecl;
45    ParseScope BodyScope(this, Scope::FnScope|Scope::DeclScope);
46    Decl *BodyRes = Actions.ActOnStartOfFunctionDef(getCurScope(), D);
47
48    D.complete(BodyRes);
49    D.getMutableDeclSpec().abort();
50    Actions.ActOnDefaultCtorInitializers(BodyRes);
51    StmtResult FnBody;
52    StmtVector FnStmts;
53    /* add function body statements */
54    FnBody = Actions.ActOnCompoundStmt(Loc, Loc, FnStmts, false);
55    BodyScope.Exit();
56    TheDecl = Actions.ActOnFinishFunctionBody(BodyRes, FnBody.take());
57    returnDecl = Actions.ConvertDeclToDeclGroup(TheDecl);
58    (&Actions.getASTConsumer())->HandleTopLevelDecl(returnDecl.get());
59    Actions.CurScope = SavedScope;
60    Actions.CurContext = SavedContext;
61    Actions.FunctionScopes.push_back(SavedFSI);
62    return false;
63  }
```

つづいて構造体 CbC_env の定義を行う. この構造体は __builtin_setjmp, __builtin_longjmp が環境の保持に使う jmp_buf へのポインタと元の関数の戻り値となる変数へのポインタを持つ. 構造体の宣言は通常, 構文解析器によって Decl というクラスのインスタンスが生成され, それを利用して行う. 今回の実装では Decl の生成を手書きのコードで行うことで, 自動で構造体の定義を行う処理を実現した. 今回の実装ではそのコード群を Create__CbC_envStruct という一つの関数にまとめた. その関数のうち, 重要な部分を抜き出したものを以下のリスト 4.17 に示す. この関数はソースコードの位置と C++ で用いられるアクセス修飾子を引数に取る. この関数もスコープを変更する処理を持つが, それについては先ほど説明したので省いている. 12 行目から 22 行目までが, この構造体の Decl を生成する処理である. メンバの設定を行っているのが 27 行目から 30 行目で, ここで使用している Create__CbC_envBody 関数はメンバの宣言を行うために独自に作成した関数である. 構造体のメンバとして宣言するために ActOnField という Sema クラスの関数を呼び出す以外は後述する CreateDeclStmt 関数と処理内容にあまり差が無いため, この関数については本論文では説明を省く. 32 行目で呼び出している ActOnTagFinishDefinision

が, 生成した Decl を元に構造体の宣言の意味解析を行ってくれる関数である. これにより, 生成した構造体に対応する Type も生成される.

ソースコード 4.17: Create__CbC_envStruct 関数

```
1  void Parser::Create__CbC_envStruct(SourceLocation Loc, AccessSpecifier AS) {
2
3              :
4
5    ParsingDeclSpec SDS(*this);
6    DeclSpec::TST TagType = DeclSpec::TST_struct;
7    DeclResult TagOrTempResult = true;
8    bool Owned = false;
9    bool IsDependent = false;
10   TagOrTempResult = Actions.ActOnTag(getCurScope(), TagType, Sema::TUK_Definition, Loc,
11                                      SDS.getTypeSpecScope(), Name, Loc, attrs.getList(), AS,
12                                      SDS.getModulePrivateSpecLoc(), TParams, Owned,
                                            IsDependent,
13                                      SourceLocation(), false, clang::TypeResult());
14
15   Decl *TagDecl = TagOrTempResult.get();
16   ParseScope StructScope(this, Scope::ClassScope|Scope::DeclScope);
17   Actions.ActOnTagStartDefinition(getCurScope(), TagDecl);
18   SmallVector<Decl *, 32> FieldDecls;
19
20   FieldDecls.push_back(Create__CbC_envBody(TagDecl, DeclSpec::TST_void, Loc,
         __CBC_STRUCT_POINTER_NAME));
21   FieldDecls.push_back(Create__CbC_envBody(TagDecl, DeclSpec::TST_void, Loc,
         __CBC_STRUCT_ENV_NAME));
22
23   Actions.ActOnFields(getCurScope(),Loc, TagDecl, FieldDecls,Loc, Loc,attrs.getList());
24   StructScope.Exit();
25   Actions.ActOnTagFinishDefinition(getCurScope(), TagDecl, Loc);
26
27             :
28 }
```

つづいて両予約語解析時に共通する処理の一つである変数の定義文に関して説明する. これも定義に関する文であるので Decl の生成がメインとなる. 今回の実装では以下のリスト 4.18 に示す CreateDeclStmt という関数を作り, その関数によってその処理を行う. ここでも重要でない処理は省いている. この関数は引数として変数名, code segment へのポインタかどうか, 呼び出し元の関数の型をコピーするかどうか, 変数の型, 構造体だった場合の構造体名を持つ. 4 行目の setTST 関数が指定された型修飾子を設定している箇所である. 変数名は 6 行目の SetIdentifier 関数によって設定され, その後は関数へのポインタだった場合や構造体だった場合のときにのみ行われる処理が来るがここでは省いた. その後型コピーを行うかどうかで処理が異なる. 型コピーの方法については既に述べたのでここでは省く. すべての処理を終えると, Decl の意味解析が行われ, その結果が StmtResult として返る.

ソースコード 4.18: CreateDeclStmt 関数

```
1  StmtResult Parser::CreateDeclStmt(IdentifierInfo *II, bool isRetCS, bool copyType, DeclSpec::TST
       valueType, IdentifierInfo* Name){
2    DeclGroupPtrTy DeclGPT;
3    ParsingDeclSpec DS(*this);
4    setTST(&DS, valueType, Name);
5    ParsingDeclarator D(*this, DS, static_cast<Declarator::TheContext>(Declarator::BlockContext));
```

```
6    D.SetIdentifier(II, Loc);
7
8            :
9
10   SmallVector<Decl *, 8> DeclsInGroup;
11   Decl *FirstDecl;
12
13   if (copyType)
14     FirstDecl = HandleDeclAndChangeDeclType(D);
15   else
16     FirstDecl = ParseDeclarationAfterDeclaratorAndAttributes(D);
17
18   D.complete(FirstDecl);
19   DeclsInGroup.push_back(FirstDecl);
20   DeclGPT = Actions.FinalizeDeclaratorGroup(getCurScope(), *DSp, DeclsInGroup);
21   return Actions.ActOnDeclStmt(DeclGPT, Loc, Loc);
22 }
```

つづいて両予約語解析時に共通するもう一つの処理, 代入文の生成について説明する. 今回の実装ではこの処理を以下のリスト 4.19 に示す CreateAssginmentStmt 関数にまとめた. ここでも重要でない処理は省いている. この関数は引数として左辺, 右辺の変数名と, 左辺が構造体のメンバかどうか, 右辺が演算子 & を持つかどうか, 左辺が構造体だった場合の構造体名を求める. 8 行目から 14 行目までは左辺に対応する ExprResult を取得する処理で, ExprResult は一つの式に対応するクラスである. 同様に, 18 行目が右辺に対応する ExprResult を取得する処理で, 代入式の生成は 20 行目に当たる. ここで生成した ExprResult を ActOnExprStmt に渡すことで, 式として StmtResult に変換される.

ソースコード 4.19: CreateAssignmentStmt 関数

```
1  StmtResult Parser::CreateAssignmentStmt(IdentifierInfo* LHSII, IdentifierInfo* RHSII, bool
       LHSisMemberAccess, bool RHShasAmp,
2                                          IdentifierInfo* extraLHSII){
3    ExprResult Expr,LHS,RHS;
4    Token Next,LHSToken;
5
6            :
7
8    ExternalSpace::StatementFilterCCC Validator(Next);
9    Sema::NameClassification Classification = Actions.ClassifyName(getCurScope(), SS, LHSII, Loc,
         Next, false, SS.isEmpty() ? &Validator : 0);
10   setExprAnnotation(LHSToken, Classification.getExpression());
11   LHSToken.setAnnotationEndLoc(Loc);
12   PP.AnnotateCachedTokens(LHSToken);
13
14   LHS = getExprAnnotation(LHSToken);
15
16           :
17
18   RHS = LookupNameAndBuildExpr(RHSII);
19
20   Expr = Actions.ActOnBinOp(getCurScope(), Loc,tok::equal,LHS.take(),RHS.take());
21
22   return Actions.ActOnExprStmt(Expr);
23 }
```

さて, 残る if 文の自動生成についてであるが, これは if 文のためのスコープを生成した後条件文を作り, その後 if 文内のスコープを作りそこに文を生成していくことで行える. この処理は前述した code segment の生成に含まれる処理に酷似する. したがってここで

はその説明を省略する.

4.5　プロトタイプ宣言の自動化

プロトタイプ宣言の自動化は本研究で追加された機能である. CbC の code segment の処理単位は小さく, そのままでは大量のプロトタイプ宣言を書く必要があり, 好ましくない. また, tail call elimination 強制のために付加した fastcc は正確にプロトタイプ宣言を書くことを要求する. つまりプロトタイプ宣言を自動的に行うようにすることで fastcc の条件を安定して満たすことができ, さらにプログラマは大量のプロトタイプ宣言を書く必要がなくなるのである.

プロトタイプ宣言の自動化は, パーサーが code segment への継続の解析を行った際にプロトタイプ宣言の有無を確認し, 存在しない場合に継続先の code segment のプロトタイプ宣言を生成するというようにして行う. リスト 4.20 はプロトタイプ宣言の有無を確認する関数である. この関数に code segment 名を渡すとプロトタイプ宣言の有無を返す. LookupParsedName で指定された識別子が解析されているかどうかを知ることが出来るのでそれを用いて有無の判別を行う.

ソースコード 4.20: プロトタイプ宣言の有無を確認する関数

```
1 bool Parser::NeedPrototypeDeclaration(Token IITok){
2   LookupResult LR(Actions, IITok.getIdentifierInfo(), IITok.getLocation(), Actions.
        LookupOrdinaryName);
3   CXXScopeSpec SS;
4   Actions.LookupParsedName(LR, getCurScope(), &SS, !(Actions.getCurMethodDecl()));
5
6   return (LR.getResultKind() == LookupResult::NotFound);
7 }
```

プロトタイプ宣言の有無を確認し, 存在しない場合には以下のリスト 4.21 に示した CreatePrototypeDeclaration 関数を用いて対象となる code segment のプロトタイプ宣言を生成する. この関数では対象 code segment の定義を探しだし, そこからプロトタイプ宣言部分を抜き出す. 定義を探す処理は 4.22 に示す SearchCodeSegmentDeclaration 関数で行う. プロトタイプ宣言の自動生成を行う際まず scope を top レベルに移さなければならない. 有無を確認した時点では scope は関数の内部であるためこのまま生成すると不具合が生じるためである. リスト 4.21 の 10 行目までがその処理である. top の scope が親を持たないことを利用して top に変更している.

その次のブロックでは Token を保存している. この関数の処理によって code segment 名等の token が解析されたという扱いになってしまうが, プロトタイプ宣言の生成が済んだあとは呼び出しの処理に戻る必要がある. そのため token を保存しておき解析を再度行う際に利用できるようにしているのである.

その後, SearchCodeSegmentDeclaration 関数を使用して対象 code segment の定義を探すが, SkipAnyUntil という関数を用いる. この関数は指定された token が来るまで token を飛ばすというものである. このままこの関数を使用すると現在のバッファが破壊されてしまい, 以後正しく解析を行えなくなってしまう. それを回避するために, 現在のファイ

ルを新しいものとして別に読み込みを行い, この処理に入る. これにより現在のバッファを破壊せずに code segment の探索が可能になる.

対象 code segment の定義を見つけたらそれを元にプロトタイプ宣言を行う. その後前のファイルに戻り, 保存した token を元に戻すことで正しく元の解析位置に戻ることが出来るのである.

ソースコード 4.21: プロトタイプ宣言の生成を行う関数

```
1  void Parser::CreatePrototypeDeclaration(){
2    // move to the top level scope
3    Scope *SavedScope = getCurScope();
4    DeclContext *SavedContext = Actions.CurContext;
5    sema::FunctionScopeInfo *SavedFSI = Actions.FunctionScopes.pop_back_val();
6    Actions.CurContext = static_cast<DeclContext *>(Actions.Context.getTranslationUnitDecl());
7    Scope *TopScope = getCurScope();
8    while(TopScope->getParent() != NULL)
9      TopScope = TopScope->getParent();
10   Actions.CurScope = TopScope;
11
12   Token Next = NextToken();
13   Token CachedTokens[3] = {Next, PP.LookAhead(1)};
14   Token SavedToken = Tok;
15   Token IITok = Tok.is(tok::identifier) ? Tok : Next;
16   PP.ClearCache();
17   PP.ProtoParsing = true;
18   ProtoParsing = true;
19
20   const DirectoryLookup *CurDir = nullptr;
21   FileID FID = PP.getSourceManager().createFileID(PP.getCurrentFileLexer()->getFileEntry(),
         IITok.getLocation(), SrcMgr::C_User);
22   PP.EnterSourceFile(FID,CurDir,IITok.getLocation());
23   ConsumeToken();
24
25   if(SearchCodeSegmentDeclaration(IITok.getIdentifierInfo()->getName().str())){
26     DeclGroupPtrTy ProtoDecl;
27     ParseTopLevelDecl(ProtoDecl);
28     // add declaration to AST.
29     if(ProtoDecl)
30       (&Actions.getASTConsumer())->HandleTopLevelDecl(ProtoDecl.get());
31     // File Closing
32     Token T;
33     PP.HandleEndOfFile(T, false);
34
35     // recover tokens.
36     Tok = SavedToken;
37     PP.RestoreTokens(CachedTokens, 2);
38
39   }
40   else {
41     // recover tokens.
42     CachedTokens[2] = Tok;
43     Tok = SavedToken;
44     PP.RestoreTokens(CachedTokens, 3);
45   }
46
47   // move to the previous scope.
48   Actions.CurScope = SavedScope;
49   Actions.CurContext = SavedContext;
50   Actions.FunctionScopes.push_back(SavedFSI);
51
52   ProtoParsing = false;
```

```
53    PP.ProtoParsing = false;
54  }
```

ソースコード 4.22: code segment を探す関数

```
1  bool Parser::SearchCodeSegmentDeclaration(std::string Name){
2    while(SkipAnyUntil(tok::kw__code, StopBeforeMatch)){
3      if(NextToken().is(tok::identifier) && NextToken().getIdentifierInfo()->getName().str() ==
           Name)
4        return true;
5      ConsumeToken();
6    }
7    return false;
8  }
```

4.6　フレームポインタ操作最適化

フレームポインタ操作の最適化は omit leaf frame pointer を強制することで行う．この最適化は 3.6 節でしたとおり leaf function に対して行われるが, 継続を続ける code segment は何もせずとも leaf function の条件を満たしているので内部で有効化するだけで良い.

omit leaf frame pointer は clang 内部では CodeGenOpts.OmitLeafFramePointer として表現されている. clang は CodeGen で LLVM IR の function を生成する際にこのフラグが立っているかどうかを確認し, 立っている場合には関数の no-frame-pointer-elim という attribute を false にする．それにより最適化が当該関数に作用するようになる．この, LLVM IR function への attribute 設定を行っているのは $(CLANG)/lib/CodeGen/CGCall.cpp である．ここをリスト 4.23 のように変更した．通常は OmitLeafFramePointer のみを確認する部分を HasCodeSegment によって code segment の有無も確認するように変更している．これにより, code segment を含むコードのコンパイル, 即ち CbC のコードのコンパイルの際には自動的に omit leaf frame pointer が有効化されるようになった.

ソースコード 4.23: omit leaf frame pointer 有効化

```
1      } else if (CodeGenOpts.OmitLeafFramePointer || LangOpts.HasCodeSegment) {
2        FuncAttrs.addAttribute("no-frame-pointer-elim", "false");
3        FuncAttrs.addAttribute("no-frame-pointer-elim-non-leaf");
```

第5章　Gears OS サポート

5.1　meta Code Segment の接続

5.2　stub の接続

第6章　評価・考察

6.1　本研究での改善による成果

6.2　アセンブリコードの評価

6.3　性能評価

6.4　LLVM, clang の利点

第7章　結論

7.1　今後の課題

謝辞

参考文献

[1] 宮國渡, 河野真治, 神里晃, 杉山千秋. Cell 用の fine-grain task manager の実装. 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.

[2] 赤嶺一樹, 河野真治. Datasegment api を用いた分散フレームワークの設計. 日本ソフトウェア科学会第 28 大会論文集, Sep 2011.

[3] 河野真治, 島袋仁. C with continuation と、その playstation への応用. 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), May 2000.

[4] Llvm documentation.

[5] Llvm language reference manual.

[6] Clang 3.9 documentation.

[7] clang api documentation.