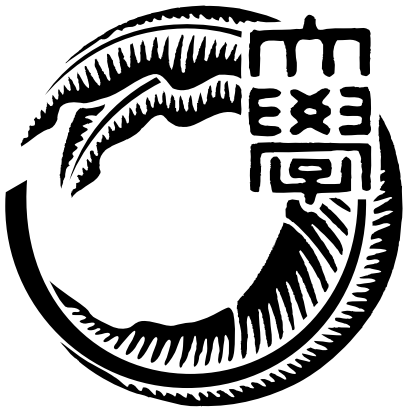
Continuation based CコンパイラのGCC-4.2による実装

The implementation of Continuation based C Compiler on GCC

平成19年度 卒業論文



琉球大学 工学部 情報工学科

045760E 与儀 健人

指導教官 河野 真治

目 次

図 目 次

プログラムコード目次

第1章　序論

1.1　研究の背景と目的

当研究室では Continuation based C(以下 CbC) という言語を提案している。この CbC は C と同じ様な構文であるが、よりアセンブリに近い形で実現されている。そのため、C 言語より細かく、アセンブリよりは高級なプログラミングを可能にする。

これまで当研究室では CbC のコンパイルに、太田昌孝氏の Micro-C をベースにした独自のコンパイラを使用していた。このコンパイラは i386, ppc, mips, spu などのアーキテクチャに対応おり、出力されるアセンブリも GCC のものと比較して、速度に大きな差は見られない。

しかし UNIX 環境に置けるコンパイラの標準は GCC であることは明らかであり、また GCC には最適化という機能が存在し、その機能を on にすれば GCC の出力コードの性能は格段に上昇する(注1)。さらに GCC の対応しているアーキテクチャは数十種類に及ぶ。

この様な背景から、CbC を GCC でコンパイルしたいという要望がでてきた。本研究ではこの言語を GCC へ移植することを目的とする。それにより GCC の最適化機能による CbC のコード性能の向上、また複数のアーキテクチャへの対応を目指す。

1.2　論文構成

本論文は以下の様な構成になっている。

第 2 章 CbC の概要について述べる。

第 3 章 CbC コンパイル機能の実装に関わる GCC の内部構造を述べる。

第 4 章 本論文の主旨である CbC コンパイル機能の GCC への実装について説明する。

第 5 章 この実装によってどの程度の性能向上ができたかを評価する。

第 6 章 実装によって明らかになった問題点、また今後の課題、改良点を論じる。

また、本研究では GCC の version-4.2.2 (注2)を使用した。GCC のソースコードは Free Software Foundation の GCC ページ (http://www.gnu.org/software/gcc/mirrors.html) からダウンロード可能である。論文内で示される C のソースファイルはこのソースコードを展開したディレクトリをカレントディレクトリとする相対パスで示されている。

(注1) もちろんこれはアーキテクチャにも依存する。i386 アーキテクチャでの実行速度の測定結果は第 5 章を参照の事。
(注2) 実装を始めた当初は 4.2.1. 2008/02/01 には 4.2.3 がリリースされている。

第2章　Continuation based C

2.1　CbC とは

Continuation based C (以下 CbC) は当研究室が提案するアセンブラよりも上位で C よりも下位な記述言語である [2]。C の仕様からループ制御や関数コールを取り除き、継続 (goto) や code segment を導入している。これによりスタックの操作やループ、関数呼び出しなどのより低レベルでの最適化を、ソースコードレベルで行うことができる。

図 2.1 は code segment 同士の関係を表したものである。図中の丸は code segment, 矢印は継続による code

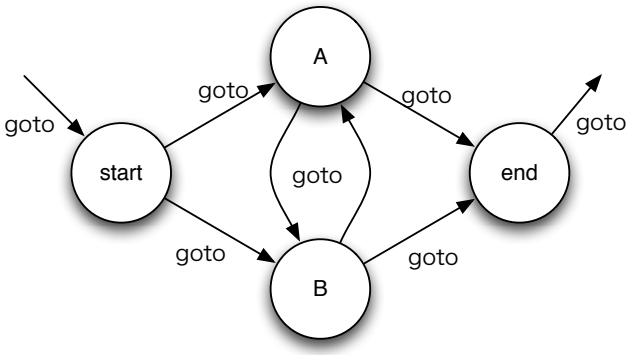


図 2.1: code segment 間の “継続”

segment 間の接続を表している。code segment`start` は実行を終えると goto によって別の code segment `A` もしくは `B` に実行を継続する(注1)。また、`A` から `B`, 再び `A` の用に継続を繰り返すことも可能だ。このように、code segment から goto を用いて別の code segment へ飛ぶ構成はオートマトンと似た構造になっていることがわかる。ただし CbC の継続では interface と呼ばれる code segment への引数が存在し、継続して実行された code segmnt は前の code segment からの状態を引数によって受け取ることができる。

以下では実装に必要な CbC の構文、code segment の定義と継続 (goto) について説明する。

2.1.1　code segment

code segment は CbC における最も基本的な処理単位である。構文としては通常の関数と同じであるが、型は “__code” となる。ただし、code segment は関数のようにリターンすることはないので、これは code segment であることを示すフラグの様なものである。

code segment の処理内容も通常の関数と同じように定義されるが、C と違い code segment では for や while, return などの構文は存在しない(注2)。ループ等の制御は自分自身への再帰的な継続によって実現さ

(注1) どちらにするかはもちろん if 文で決定する。

(注2) 言語仕様としては存在しないが、Micro-C version では while や for を使用することは可能である。この場合は CbC でなく、CwC(Continuation with C) とよばれる。

れることになる。

2.1.2　継続 (goto)

code segment は処理の基本単位となるが、その code segment 間の移動は CbC 独自の構文 “goto” を使って実現される。これを “継続” という。この goto は C における label への goto とは違い、goto の後ろに関数呼び出しの様な形をとる。例として、ある code segment `cs` への継続は `goto cs(10, "test");` となる。これにより、cs に対して引数 `10` と`"test"`を渡すことができる。ただし関数コールとは違い、継続ではコールスタックの拡張を行わない。代わりに goto を発行した code segment の持つスタック自体に次の code segment の引数を書き込むことになる。また、必要のない return アドレスの push なども行わない。

2.1.3　コード例

簡単な実例として、code segment による loop の実装を以下に示す。図 2.2 はこのコードをオートマ

リスト 2.1: CbC コード例

```
__code while_process(int total, int count){
    printf("while_process: do something.\n");
    total += count;
    count++;
    goto while_cond(total, count);
}

__code while_cond(int total, int count){
    printf("while_cond: check condition.\n");
    if ( count <= 100 ){
        goto while_process(total, count);
    }else{
        goto while_end(total);
    }
}

__code while_end(int total){
    printf("while_end: the loop ended.\n");
    printf("
: total = %d.\n", total);
    goto cs_exit(0);
}
```

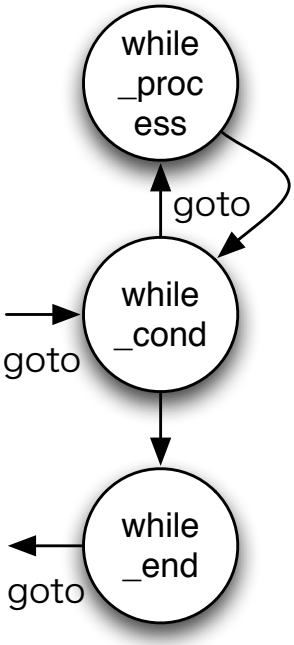


図 2.2: CbC コード例

トンのように表したものだ。このコードでは `while_cond` が条件判定し真を得れば `while_process` へ。`while_process` は処理が終わると `while_cond` に戻るので、ここでループが出来上がる。

第3章　The GNU Compiler Collection

3.1　GCCの基本構造

GCC はコンパイラという名称を持っているがコンパイル、アセンブル、リンクまで、ソースコードを実行可能にするまでの変換を全て受け持っている。しかし CbC の拡張にはコンパイル以外は関わらないので、ここでは C のコンパイル処理を扱う cc1 というプログラムについて説明する。(以下、GCC は cc1 と同じ意味で使用する)

GCC は pass と呼ばれる一連の処理の中でソースコードをアセンブリに変換する。以下ではその pass の中でも重要なものをその実行順に説明する。

parsing 一般的なコンパイラと同じく、GCC もまずはパーサによってソースコードを解析し、解析した結果は Generic Tree と呼ばれる tree 構造の構造体に格納される (Generic Tree に関しては 3.1.1 で説明)。C のパーサは c_parse_* という関数名で統一されている。

gimplification この段階では Generic Tree をもとにこれを GIMPLE に変換していく (GIMPLE は 3.1.2 で説明)。

GIMPLE optimization 前段階で変換した GIMPLE に対して最適化を行う。この最適化には “Dead store elimination” や if 文等の条件判定を最適化する “Lower control flow” などが含まれる。

RTL generation ここで、GIMPLE をもとに RTL を生成する (3.1.3 で説明)。この段階でほぼ言語依存性がなくなる。GIMPLE を解析して RTL を生成する関数は expand_* という名前で統一されている。

RTL optimization 前段階で生成された RTL に対して最適化を行う。この最適化には必要のないコードを除去する “Cleanup control flow graph” や無駄に複数回行われる演算を減らす “Common subexpression elimination” などがある。

Output assembly 最後に RTL をもとにターゲットマシンのアセンブリに変換する。

これらの処理は図 3.1 のように表される。各 pass は通常は各々の関数毎に行われるものだが、inline 関数の処理や、関数間での最適化を行う場合には一つのソースファイル毎に行われる。また、ここでは説明してないが Tokenizer(字句解析器) ももちろん存在する。しかしこれは一般的なコンパイラと同じく parser の一部として同じ pass で行われているので割愛した。

以下の節では GCC において重要なデータ構造である Generi Tree, GIMPLE, RTL について簡単に説明する。なお、詳しくは Free Software Foundation の GCC の Web ページにある GCC Internals Manual[3] を参照していただきたい。

3.1.1　Generic Tree

この節では GCC の中でも最も重要なデータ構造である Generic tree について簡単に説明する。

Generic Tree はパースしたソースコードの内容を表したツリー構造のデータの集合である。例として、tree には FUNCTION_TYPE や CALL_EXPR, INTEGER_CST, PLUS_EXPR などがあり、それぞれ関

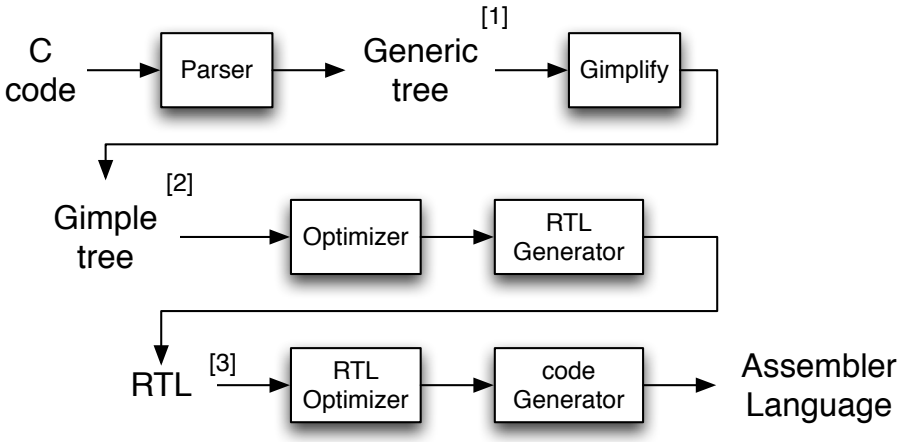


図 3.1: GCC の pass

数型、関数呼び出し、整数値定数、足し算を表している。これらはそれぞれ別の構造体であるが、tree 共用体がすべての構造体をメンバとして持っているので、tree ですべてを表すことができる。c_parse_* 関数で C のソースをパースし、その部分に合う tree が生成され、最終的に関数全体を表す一つのツリーとしての tree 構造体ができあがる。すべての tree の種類は gcc/tree.h が include する gcc/tree.def で定義されており、gcc/tree.c にはこの tree を生成、もしくは操作する様々な関数が定義されている。

具体的な例として、FUNCTION_TYPE と CALL_EXPR を次で説明する。この二つは本論文の主旨である GCC 拡張にも深く関わってくる tree である。

FUNCTION_TYPE

まずは関数の型を表す FUNCTION_TYPE を見てみる。関数の型を表現する為に必要な情報としては関数の返す型、関数に渡す引数列 (parameter) の型の二つが必要だろう。FUNCTION_TYPE もこの二つを主な要素に持つ。図 3.2 は関数 `int test(int *, double)` をパースした際の tree を表したものである。図の角丸の長方形は tree、矢印は tree へのポインタを表している。図の FUNCTION_TYPE から出ている “type” という矢印は関数の返す型である。これは int 型なので、int 型を表す INTEGER_TYPE へのポインタとなっている。ただし、tree 構造では char 型でも INTEGER_TYPE が使用される。この二つの型の違いを表すのは INTEGER_TYPE の bit 数を表す “size” から参照される INTEGER_CST である。 これは整数値定数を表し、ここでは 32 となっている。char 型の場合はこれが 8 となる。実数値を表す REAL_TYPE でも同じく、double なら 64, float なら 32 で表現される。

このように関数を表す FUNCTION_TYPE だけでなく、その返却値の型、ポインタ、引数、引数のリスト、int 型のサイズにいたるまで、全て tree で表現されている。図では省略したが、INTEGER_TYPE では bit size に加えて、最小値や最大値も INTEGER_CST へのポインタが存在する。

興味深いのは最後の引数型が VOID_TYPE になっていることである。これは引数の最後を明示的に指定しており、これがない場合は、GCC では可変長引数を持つ関数として扱うことになっている。

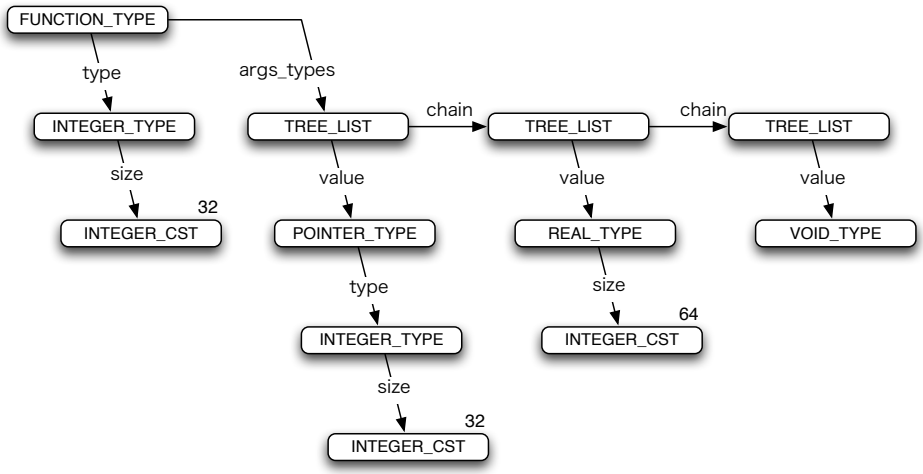


図 3.2: int test(int *, double) 関数の型 FUNCTION_TYPE

CALL_EXPR

次に CALL_EXPR の詳細を見てみよう。`int test01(int a, double b){...}` という関数に対して、`test01(c+d, 2.0);` のように呼び出したときのその文を表す tree は次の図 3.3 のようになる。CALL_EXPR

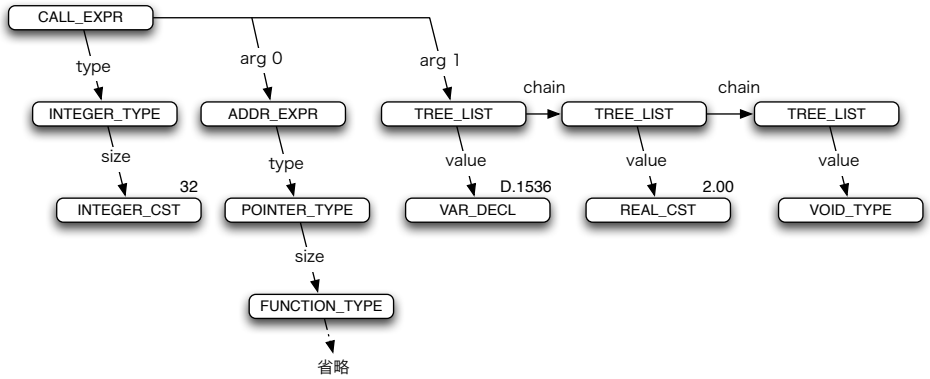


図 3.3: test01(c+d, 2.0); の構文木 CALL_EXPR

から arg 0 で参照されている ADDR_EXPR は呼び出す関数のアドレスを表す tree となっており、その type からさらに関数の型を参照できる。また、arg 1 から参照されるのは関数呼び出しの際の argument である。これは FUNCTION_TYPE で示される parameter とは違い、実際に関数に渡す値となる。これを RTL に落とす際には FUNCTION_TYPE から得られる parameter の型と、argument の型が一致するか?しなければキャストできるか? などの判定が行われる。また、最初の argument が D.1536 という VAR_DECL になっている。これは Gimplification pass によって生成されたローカル変数で、c+d の演算結果が格納されている。これについては次節で述べる。

3.2 節で解説する Tail call elimination はこの CALL_EXPR に対して行われる。

3.1.2　GIMPLE

GIMPLE とは GCC 内部でのみ使われる造語で、“Generic Tree の Simple なもの” という意味で、縮めて GIMPLE となっている。

GCC の gimplification によって Generic Tree が GIMPLE に変換されるが、データ構造としては Generic も GIMPLE も同じ tree 共用体を使っており、差はない。しかし GIMPLE では tree の様々な要素を制限しており、次に行われる最適化 pass を行いやすくしている。

具体的な gimplification の動作としては、複雑な式を 3 番地コード (3-address form) と呼ばれる簡単な形式に変換するものがある。この 3 番地コードとはどんな演算でも、`a = c # b` の形の複数の式に変換して、最適化を行いやすくするものである。また、if 文以外の制御構文 (while や for など) をすべて if と goto の形式に変換することも Gimplification の仕事である。3 番地コードの例と同じく、関数呼び出しでは全ての argument は式ではない変数となる。前節で `c+d` が `D.1563` となっていたのはこのためである。

一例として、関数を gimplification した後の状態でどのようになるかをリスト 3.1, 3.2 に示す。

リスト 3.1: 元の関数 test

```
int test(int a, int b){
    int i,sum=0;
    i = a;
    while ( i <= b ) {
        sum += i;
        i++;
    }
    return sum - a*b;
}
```

リスト 3.2: gimplification 後の関数 test

```
test (a, b){
    int D.1556;
    int D.1557;
    int i;
    int sum;
    sum = 0;
    i = a;
    goto <D1554>;
    <D1553>:;
    sum = sum + i;
    i = i + 1;
    <D1554>:;
    if (i <= b) {
        goto <D1553>;
    } else {
        goto <D1555>;
    }  <D1555>:;
    D.1557 = a * b;
    D.1556 = sum - D.1557;
    return D.1556;
}
```

3.1.3　RTL

RTL は Register Transfer Language の略称である。この言語は一般的に言う中間言語のことで、アーキテクチャに依存せずにアセンブリ言語を表現する汎用的な言語となっている。例えば、“PLUS” という RTL は足し算 (ia32 なら add 命令)、“CALL” という RTL は関数呼びだし命令など、アセンブラと 1 対 1 の関係で記述することができる。ただし、アセンブラとの大きな違いとして無数のレジスタを持つことが挙げられる。これによりコード生成直前の最適化を行いやすくしている。またアセンブリに近いことから、この RTL で表現されたコードを実際のアーキテクチャのアセンブリに変換することが容易になることもその特徴の一つだと言える。

GCC ではこの RTL 表現するために rtx という構造体を用いている。この rtx は命令だけでなくレジスタや数値を表すことができ、それぞれ “mode” と呼ばれる bit サイズを指定することができる。RTL を生成

する際は命令を表す rtx のオペランドに数値やレジスタを表す rtx を付加することで一つの命令として確保されることになる。

3.1.4　最適化機構

GCC において、最適化はとても重要な要素である。ソースコードの大半は最適化のために存在し、先に述べた GIMPLE や RTL も最適化のために最適なデータ構造を成している。また、新しい最適化を追加する環境も整備されており、`struct tree_opt_pass` 型の `all_passes` という変数に独自の関数を登録するだけでその関数による最適化が行われる。

最適化は本研究においてはほとんど関係しないが、“Tail call elimination” に関しては例外である。この最適化については次節で詳しく説明する。

3.2　Tail call elimination

“Tail call elimination” は通常 call 命令を使用すべき関数呼び出しで、jump 命令に変更するというものである。この最適化は本研究における CbC コンパイラの実装に深く関わってくる。本節ではこの最適化機構について詳しく説明する。

3.2.1　Tail call の概要

具体的に説明する。まず main 関数から関数 A を呼び出していて、関数 A の最後の処理 (return 直前) では次に関数 B を呼び出している状況を考える。このあと関数 B の処理が終了すると、ret 命令により一旦関数 A に戻ってきて、そこで再び ret 命令をつかって main に戻ることになる。“Tail call elimination” ではこの B から A に戻る無駄な処理を低減する。この様子を図 3.4 に示したので参考にしていただきたい。

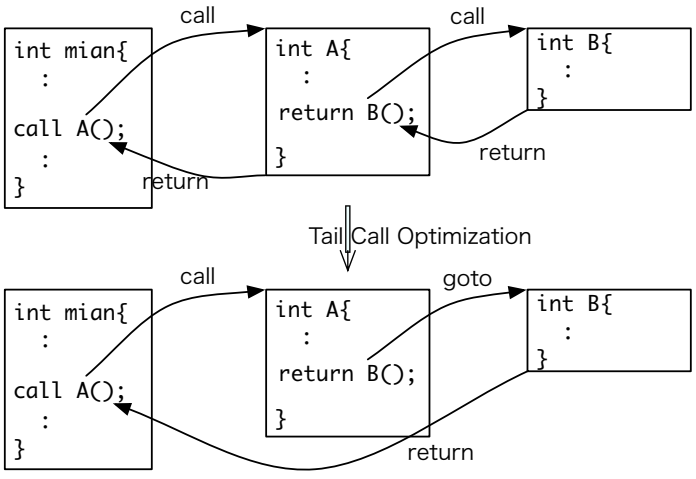


図 3.4: Tail call elimination の例

次に “Tail call elimination” によって、アセンブリレベルでどのようにコードが変わるのか、スタックの変化も交えて見てみる。この例では最も一般的に使われており、また Micro-C version の CbC も対応している i386 形式のアセンブラを使用している。

図 3.4 と同じように呼び出される関数 main, A, B をリスト 3.3 の様に定義する。

リスト 3.3: 関数 main A B の例

```
void B(int A, int A, int C){
    //printf("B: a=%d, b=%d, c=%d\n", A, B, C);
    return ;
}
void A(int a, int b, int c, int d){
    //printf("A: a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
    return B(a, b, c+d);
}
int main(int argc, char **argv){
    //printf("main: \n");
    A(10, 20, 30, 40);
    return 0;
}
```

これを通常通り、“Tail call elimination” を使用せずにコンパイルすると次のリスト 3.4,3.5 のようなコードが出力される。(ただし関数 B は最適化に関係しないのでここでは省いた。)

リスト 3.4: 関数 A のコンパイル結果 (Tail call なし)

```
A:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    movl    20(%ebp), %eax
    addl    16(%ebp), %eax
    movl    %eax, 8(%esp)
    movl    12(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    8(%ebp), %eax
    movl    %eax, (%esp)
    call    B
    leave
    ret
    .size   A, .-A
```

リスト 3.5: 関数 main のコンパイル結果

```
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    subl    $20, %esp
    movl    $40, 12(%esp)
    movl    $30, 8(%esp)
    movl    $20, 4(%esp)
    movl    $10, (%esp)
    call    A
    movl    $0, %eax
    addl    $20, %esp
    popl    %ecx
    popl    %ebp
    leal    -4(%ecx), %esp
    ret
    .size   main, .-main
```

Tail call をしない場合は A のスタック領域の上に B のスタック領域が確保され、B が終了するとそれが破棄される形になる。

次に Tail call elimination が行われた場合のコンパイル結果を図 3.6 に示す。

リスト 3.6: Tail call elimination の行われた関数 A

```
A:
    pushl   %ebp
    movl    %esp, %ebp
    movl    20(%ebp), %eax
    addl    %eax, 16(%ebp)
    popl    %ebp
    jmp     B
    .size   A, .-A
```

20(%ebp) は変数 d、16(%ebp) は変数 c を表している。ここでは B のためにスタック領域は確保せず、かわりに A のスタック領域に B のための引数を書き込んでいることが分かる。ただし、変数 a と b は書き込む位置も値も変わらないので触れられていない。このときのスタックの様子を図 3.5 に表した。図 3.5 の各

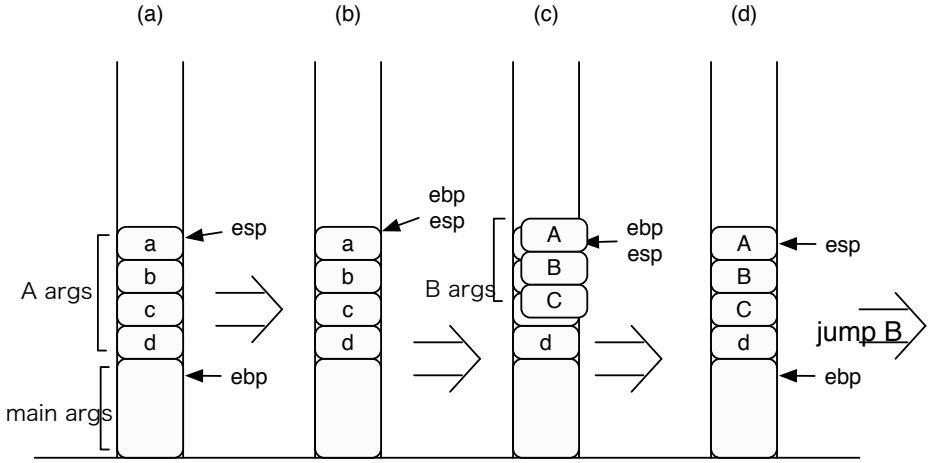


図 3.5: 関数 A から B を呼び出す時のスタックの様子 (Tail call)

ステップは次のような状態を表している。

(a) main から A が呼ばれた直後の状態。esp は引数のトップをさしてるが、ebp は main の引数をさしたまま

(b) ebp を esp に合わせる。通常は ebp のオフセットから引数のアドレスを指定する。

(c) A 自身のスタックフレームに B 用の引数をつめる。

(d) ebp を元に戻す。その後関数 B に jump。

(a),(b) は関数 A の初期化処理、 (c),(d) は関数 B の呼び出し処理である。普通の関数では (b) と (c) の間にいくつかの処理があると思われるが、ここでは省略した。通常は関数呼び出しの際は A のスタックフレームの上に新たに作るはずである。しかし、関数 A の Tail call elimination 後のコードを見ても分かる通り、無駄な処理が少なくなっていることが分かる。これが Tail call elimination における最適化の主な効果である。最大の効果が得られるのは、caller 関数が持っている引数を callee 関数に直接渡す場合である。この時はスタック操作は全く必要なく、単に jump 命令のみになる。

3.2.2　Tail call の条件

Tail call が可能かどうかの条件についてここで考察する。必要に応じて前節の図 3.5 と、図 3.3 を説明に用いるので参考にしていただきたい。

まず最初の条件として、“関数コールが return の直前にある” ということは自明だろう。void でなく何らかの型を返す関数なら return 文の値自体に関数呼び出しがあることになる。これは関数 B が戻る際に A を経由せず、main に直接戻ることから必ず必要な条件である。また、これに関連して “関数の返す型が caller と callee で一致している” ことが必要となる。(注 1)

(注 1) int 型の B と char 型の A なら自明なキャストが可能かという問題もあるが、これは本論文では省略する。なぜなら code segment は全部 void 型だからである。

また、図の (c) にて callee 関数 B のための引数をスタックに上書きしているが、この領域は A のためのスタックフレームであることは説明した。ここでもし B の引数が 5 つ以上あったらどうなるだろうか? 図を見て分かる通り、main のスタックまで書きつぶすことになってしまう。このことから “caller 側の引数サイズが callee 側の引数サイズより大きいもしくは等しい” という条件が必要だと分かる。

最後に callee 用の引数を格納する順番が問題になる。通常、引数は関数定義の右から順にスタックにつめられる(注2)。例えば図 3.3 のコードにおいて、A から B の呼び出しが `B(c, b, c+d)` となっていたらどうだろうか? 最初に `c+d` の書き込みによって変数 c は上書きされてしまう。そのため、最後に書き込む引数 c は上書きされた c+d が使われ、実行結果はまったく違うものになってしまうだろう。よって、“書き込んだ引数が、その後書き込む引数を上書きしてはならない” という条件も必要となる。

他にも細かな条件はあるが、以上の考察より以下の 4 つの条件が明らかになった。

- 関数コールが return の直前にある
- 関数の返す型が caller と callee で一致している
- caller 側の引数サイズが callee 側の引数サイズより大きいもしくは等しい
- 書き込んだ引数が、その後書き込む引数を上書きしてはならない

CbC コンパイル機能の実装の際にはこれらの条件をパスさせる必要がある。

(注2) アーキテクチャによっては逆のものもあるかもしれないが、ここでは順序があるということが重要

第4章　実装

第 2, 3 章をもとに、GCC への CbC のコンパイル機能の実装を行う。実装における最大の問題は goto 文での code segment への jump の際のスタックフレームの状態をどう扱うかである。先に述べたように code segment へ飛ぶ時は Tail call を使用するのだが、その条件として caller 関数の引数サイズは callee 関数と同じかより大きくなければならない。

これを解決するために、この実装では code segment の引数サイズを一定にすることにした。どのような code segment を定義してもスタックは一定に保たれる。

実装の流れとしては次のようになる。

1. __code token の追加 (Tokenizer で読み込めるようにする)
2. code segment のパース及び tree 生成
3. CbC の goto ステートメントのパース及び tree 生成
4. goto ステートメント tree の RTL への変換
5. その他エラーメッセージ処理やコード改良

以下の節ではそれぞれの行程について詳しく説明する。

4.1　__code 基本型の追加とパース

まず最初に必要となるのが “__code” という予約語を定義することである。C の予約語は全て gcc/c-parser.c にて `reswords` 配列に定義されている。次のように修正した。

リスト 4.1: reswords 定義

```
static const struct resword reswords[] =
{
          :
    { "_Decimal128",      RID_DFLOAT128, D_EXT },
    { "__alignof",        RID_ALIGNOF,    0 },
    { "__attribute__",    RID_ATTRIBUTE,  0 },
    /* CbC project */
    { "__code",           RID_CbC_CODE,   0 },
          :
```

ここで “__code” を定義することで、Tokenizer からそれを予約語として認識できるようになる。

もう一つ必要なものが、__code 型であることを示す id である。これは GCC が関数や変数の宣言を表す tree を生成するまでの間にデータを保持する `c_declapecs` 構造体で使用される。void 型なら `cts_void`, int 型なら `cts_int` などで表されている。これは gcc/c-tree.h にて定義されており次のようになる。

リスト 4.2: c_typespec_keyword 定義

```
enum c_typespec_keyword {
    :
```

```
  cts_int,
  cts_float,
  cts_double,
  cts_CbC_code,
  cts_dfloat32,
    :
};
```

以上により、__code をパースする準備ができた。実際にはパース段階では関数の場合や変数の場合などで違う手続きが踏まれるが、c_declspecs 構造体に cts_CbC_code を格納する手続きは declspecs_add_type() 関数に統一されている。この関数の巨大な switch 文に対して case RID_CbC_CODE を追加すれば良い。以下のようになる。

リスト 4.3: declspecs_add_type 関数

```
case RID_CbC_CODE:
if (specs->long_p)
  error ("both %<long%> and %<void%> in "
      "declaration specifiers");
else if (specs->short_p)
  error ("both %<short%> and %<void%> in "
      "declaration specifiers");
else if (specs->signed_p)
  error ("both %<signed%> and %<void%> in "
      "declaration specifiers");
else if (specs->unsigned_p)
  error ("both %<unsigned%> and %<void%> in "
      "declaration specifiers");
else if (specs->complex_p)
  error ("both %<complex%> and %<void%> in "
      "declaration specifiers");
  else
  specs->typespec_word = cts_CbC_code;
  return specs;
```

これは実際には case RID_VOID とほぼ同じである。違うのは specs->typespec_word = cts_CbC_code のみとなる。同様に code segment の型はほぼ、void 型と同じように扱うことになる。

gcc/c_decl.c にある finish_declspecs 関数は c_declspecs をもとに、パースされた型を決定し、その分のちいさな tree を生成する関数である。tree にする際は code segment は全て void と見なされるようにすることになっている。よってここで生成する tree は void にしなければならない。

リスト 4.4: finish_declspecs 関数

```
  case cts_void:
  case cts_CbC_code:
    gcc_assert (!specs->long_p && !specs->short_p
                && !specs->signed_p && !specs->unsigned_p
                && !specs->complex_p);
    specs->type = void_type_node;
    break;
```

これで__code による型が void 型にマップされた。

4.2　code segment の tree 表現

次に、関数と同じようにパースされる code segment の tree を後の処理で識別するため、FUNCTION_TYPE tree にフラグをつける必要がある。この特殊な FUNCTION_TYPE を生成する関数を gcc/tree.c に作って

おく。具体的には以下の様な関数になる。

リスト 4.5: build_code_segment_type 関数

```
tree
build_code_segment_type (tree value_type, tree arg_types)
{
    tree t;

    /* Make a node of the sort we want.  */
    t = make_node (FUNCTION_TYPE);
    TREE_TYPE (t) = value_type;
    TYPE_ARG_TYPES (t) = arg_types;

    CbC_IS_CODE_SEGMENT (t) = 1;

    if (!COMPLETE_TYPE_P (t))
        layout_type (t);
    return t;
}
```

CbC_IS_CODE_SEGMENT というマクロが code segment を示すフラグである (これは gcc/cbc-tree.h で定義してある)。ユーザが定義できるように gcc/tree.h で用意されている TYPE_LANG_FLAG_5 を使用した。この関数は通常の FUNCTION_TYPE を作る build_function_type とほぼ同じ構造になっているが、この tree をハッシュ表に登録しないところだけが違っている。

つづいてこの build_code_segment_type を使用すべき関数、grokdeclarator を修正する。この関数は今までパースしてきた情報の入った構造体、c_declspecs と c_declarator をもとに、その変数や関数を表す tree を gcc/tree.c の関数を使って生成している。

この関数で build_function_type 関数を使用している箇所 3 番目の (巨大な)switch 文の case cdk_function: の部分である。これを、code segment の場合には build_code_segment_type を使うようにする。

リスト 4.6: grokdeclarator 関数

```
  if ( declspecs->typespec_word &=&  cts_CbC_code )
    {
      type = build_code_segment_type (type, arg_types);
    }
  else
    {
      type = build_function_type (type, arg_types);
    }
```

これで、__code 型がパースされた時には FUNCITON_TYPE にフラグが付くようになった。code segment かをチェックする時は tree type に対して CbC_IS_CODE_SEGMENT (type) として真偽値が返される。

4.3　goto のパース

つづいて goto 文のパースが必要になる。goto 文は通常の C の構文にも存在するが、CbC では goto トークンの後に関数呼び出しと同じ構文がくる。

C の関数定義をパースしているのは c_parser_statement_after_labels という関数である。この関数内の巨大な switch 文における case RID_GOTO: を修正することになる。具体的な修正は以下のようになった。

リスト 4.7: goto 文の構文解析

```
case RID_GOTO:
  c_parser_consume_token (parser);
```

```
if (c_parser_next_token_is (parser, CPP_NAME))
  {
    tree id = c_parser_peek_token (parser)->value;
    c_parser_consume_token (parser);
    if ( !c_parser_next_token_is (parser, CPP_OPEN_PAREN) )
      {
        stmt = c_finish_goto_label (id);
      }
    else //CbC goto statement
      {
        struct c_expr expr;
        tree exprlist;
        // from c_parser_postfix_expression
        expr.value = build_external_ref (id, 1, loc);
        expr.original_code = ERROR_MARK;

        c_parser_consume_token (parser);
        // from c_parser_postfix_expression_after_primary
        if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN))
            exprlist = NULL_TREE;
        else
            exprlist = c_parser_expr_list (parser, true);
        c_parser_skip_until_found (parser, CPP_CLOSE_PAREN,
                "expected %<)%>");
        expr.value = build_function_call (expr.value, exprlist);
        CbC_IS_CbC_GOTO (expr.value) = 1;
        CALL_EXPR_TAILCALL (expr.value) = 1;
        //expr.value->common.lang_flag_5 = 1;
        expr.original_code = ERROR_MARK;
        expr = default_function_array_conversion (expr);
        stmt = c_finish_return (expr.value);
        CbC_HAVE_CbC_GOTO (current_function_decl) = 1;
      }
  }
```

goto トークンを読み込むと次のトークンが何かによって処理が分かれる。CbC の goto では次のトークンは CPP_NAME となるなんらかの変数のはずである。この後 tree を生成する必要がある。ここでは `build_function_call` によって CALL_EXPR を生成している。また、それだけでなく、return 文の直前であるために、`c_finish_return` によって RETURN_EXPR も生成している。この様な処理は `c_parser_postfix_expressio` 関数における CALL_EXPR の生成とを参考にした。

4.4　expand_call の分割

前節まではパーサ段階の実装を説明した。ここからはパーサによって生成された tree を元に、RTL を生成する段階について説明する。

とはいうものの、実際に RTL をいじる必要があるのは code segment への jump のみである。これは tree 上では Tail call と認識されているので、その tree から RTL に変換する関数 `expand_call` を修正することになる。

関数 `expand_call` は CALL_EXPR tree をもとに関数呼び出しの際のスタック領域確保、引数の格納、関数への call 命令の発行などを行う RTL を生成している。しかしこの `expand_call` は約 1200 行も存在し、その大半は Tail call が可能かの判定をしているにすぎない。そこでこの実装では CbC の goto のための RTL を生成する関数 `expand_cbc_goto` を新たに作成した。

expand_call では、tree が code segment への goto だった場合にのみ expand_cbc_goto を呼び出す形になる。次節にて関数 expand_cbc_goto の詳細について説明する。

4.5　expand_cbc_goto

expand_cbc_goto の前にすこし expand_call について説明する。この関数は大きく 2 つの処理に分けられる。前半は call すべき関数のアドレスの算出や、与えられた引数を格納する場所を計算、またそれらのチェックなどを主に行う。後半には巨大な for 文ループが存在し、内部の処理が最大 2 回実行される。最初の 1 回は Tail call をする前提での処理、次は Tail call なしの処理である。最初の処理では途中で Tail call 不能と判断されると中断され、2 回目の処理が優先される。

expand_cbc_goto はこの巨大な for 文の直前に呼ばれる。簡単に説明すると for 文の最初の処理と同じことを Tail call 可否のチェックなしで実装することになる。大まかな処理内容は以下の通り

1. スタックフレームのベースアドレス RTL を取得
2. 各引数の格納されるアドレス RTL を取得
3. 各引数の式を計算 (一時的にレジスタに格納)
4. オーバーラップする引数を一時に変数に格納
5. 引数のスタックへの格納
6. call_insn RTL の生成

以下ではこれらの処理に付いてソースコードを交えながら説明する。

4.5.1　スタックフレームポインタ

引数を格納するスタックフレームのベースアドレスは、以下のコードで取得される。

```
argblock = virtual_incoming_args_rtx;
```

この virtual_incoming_args_rtx は現在実行中の caller 側の関数のフレームポインタを表す rtx である。ia32 アーキテクチャなら%ebp レジスタがこの rtx に値する。Tail call でない通常の call では virtual_incoming_args_rtx ではなく virtual_outgoing_args_rtx が使用される。こちらはこの関数が現在使用しているスタックのトップを表す rtx である。もちろん Tail call の場合には caller と同じフレームに callee 関数の引数を格納しなければならないので virtual_incoming_args_rtx が使用されている。

4.5.2　各引数の格納場所

次にそれぞれの引数を格納するためのアドレスを表す RTL を生成する。それぞれの引数がどの offset に格納されるかは expand_call の中ですでに決定し、args 変数に入っている。これと、先ほど生成した argblock rtx を元に計算する関数が compute_argument_addresses 関数である。こちらは gcc/calls.c にある関数をそのまま使用した。

4.5.3　引数の計算

引数の計算を行う。

リスト 4.8: 引数の計算

```
for (i = 0; i < num_actuals; i++)
  {
    if (args[i].reg == 0 || args[i].pass_on_stack)
      {
        preexpand_argument_expr (&args[i],
                 adjusted_args_size.var != 0);
      }
  }
```

この処理で一つ一つの引数に付いて、与えられた式を計算し、レジスタかもしくはスタックに格納しておく。preexpand_argument_expr 関数は gcc/calls.c にある store_one_arg を元に作った関数で、一つだけ引数の情報を受け取り、計算して args[i].value に計算の結果の格納されている rtx をおく。args[i].value には一時的なレジスタや、スタック、もしくは変数の場所を示す rtx が格納されることになる。よって、あとは args[i].value から args[i].stack にデータを移動する命令をするだけでよい。

4.5.4　オーバーラップ

3.2.2 節でも説明したように、2 つ以上の引数のもとのアドレスと格納先アドレスが相互に重なる場合、一時的な変数に格納する必要がある。expand_cbc_goto ではこの処理を push_overlaps 関数に任せている。それほど大きな関数ではないので以下にコードを示す。

リスト 4.9: push_overlaps 関数

```
push_overlaps(struct arg_data *args, int num_actuals){
    int i;
    for (i=0; i<num_actuals; i++)
    {
        int dst_offset;
        int src_offset;
        rtx temp;
        if ( (dst_offset=check_frame_offset(args[i].stack)) < 0 ) continue;
        if ( (src_offset=check_frame_offset(args[i].value)) < 0 ) continue;
        temp = assign_temp(args[i].tree_value, 1, 0, 0);
        if ( args[i].mode==BLKmode )
            emit_block_move ( temp, args[i].value, ARGS_SIZE_RTX(args[i].locate.size), 0 )
        else
            emit_move_insn ( temp, args[i].value );
        args[i].value = temp;
    }
    return;
}
```

重要なのは assign_temp である。この関数は指定されたサイズ分のメモリ領域をスタックに確保する。これにより、オーバラップする可能性のある引数を一時的な領域に格納できる。emit_block_move は構造体用の、emit_move_insn はその他の基本型用の move RTL を発行する関数である。また、ループの初期でこの引数の格納位置、読み込み位置がスタックフレーム内かどうかを確認し、両方が真の時のみ実行される。

4.5.5　引数の格納

オーバラップの処理が終われば引数の格納である。この処理のために、引数の計算と同じく store_one_arg のコードを参考にした関数 expand_one_arg_push を作成した。この関数では渡された引数の情報を元に、args->value から args->stack へデータを移動する RTL を生成する。具体的には以下の様な関数 emit_push_insn を使っている。

リスト 4.10: 引数の格納

```
emit_push_insn (arg->value, arg->mode, TREE_TYPE (pval), size_rtx,
                parm_align, partial, reg, excess, argblock,
                ARGS_SIZE_RTX (arg->locate.offset), reg_parm_stack_space,
                ARGS_SIZE_RTX (arg->locate.alignment_pad));
```

4.5.6　CALL_INSN

最後に CALL_INSN を発行する処理が来る。

リスト 4.11: CALL_INSN の発行

```
funexp = rtx_for_function_call (fndecl, addr);
   :
emit_call_1 (funexp, exp, fndecl, funtype, unadjusted_args_size,
             adjusted_args_size.constant, struct_value_size,
             next_arg_reg, valreg, old_inhibit_defer_pop, call_fusage,
             flags, & args_so_far);
```

この rtx_for_function_call 関数により、funexp 変数に callee 関数のアドレスを示した rtx が格納され、それを引数に emit_call_1 関数を呼び出している。ここで、変数 flags は flags & ECF_SIBCALL != 0 を満たしている必要がある。これがこの CALL_INSN が tail call であることを示すフラグとなる。

第5章　評価

今回実装できた GCC による CbC コンパイラを評価してみる。評価の手法としてはある CbC プログラムを Micro-C と GCC でコンパイルして、その出力されたコードの実行速度を測れば良いだろう。

今回測定に使用したプログラムはこれまでも Micro-C の測定に使われていたテストルーチンで、普通の C のソースを CbC に機械的に変換したものである。引数に 0 を入れると変換される前の通常の関数のコード、引数に 1 を入れるとそれが変換された CbC のコード、引数 2,3 では変換されたコードを手動で Micro-C 用に最適化したコードが実行される。また、評価は ia32 アーキテクチャの Fedora 上で行った。実行結果は表 5.1 に示される。

	./conv1 0	./conv1 1	./conv1 2	./conv1 3
Micro-C	5.25	8.97	2.19	2.73
GCC	3.69	4.87	3.08	3.65
GCC (+omit)	2.74	4.20	2.25	2.76
GCC (+fastcall)	2.70	3.44	1.76	2.34
TCC	4.15	122.28	84.91	102.59

表 5.1: Micro-C, GCC の実行速度比較 (単位 秒)

通常の Tail call elimination のみを有効にした場合の結果が 2 行目のデータである。この結果では引数に 1 を入れた場合、すなわち Micro-C 用に改良されてないコードでは 2 倍以上の速度になっていることが分かる。しかし、Micro-C に特化したコードでは速度が負けている。

次の “GCC (+omit)” では最適化フラグ-fomit-frame-pointer を付加してコンパイルした。このフラグをたてた場合、関数の最初にフレームポインタを push, このフラグをたてた場合、フレームポインタの push や pop がなるべく少なくなるようにコンパイルされる。この場合では引数 2,3 の場合も大幅に改善され、Micro-C の結果に近づいていが、やはり少し速度は勝てない。この結果には様々な理由が考えられるが、最大の理由は Micro-C では fastcall が使われていることだと思われる。Micro-C のコードでは関数呼び出しの際、スタックに全ての引数をつめるのではなく、できる限り開いているレジスタを使うようになっている。これが fastcall である。

GCC は fastcall をサポートしているので、これも試してみた。fastcall を有効にするには code segment 定義の際に`__code __attribute__ ((fastcall)) test(){` として、型と関数名の間に挿入する。ここでは上記の-fomit-frame-pointer も有効にした。その測定結果が表 5.1 の “GCC (+fastcall)” の行である。ここまで最適化を行って、Micro-C の速度を超えることができた。

この評価から本研究における目的の一つ、“CbC コードの高速化” を達成できたことが分かった。しかし、fastcall というオプションをわざわざつけるというのは無駄が多いだろう。GCC と互換性のある CbC の処理系は他にないので、code segment の場合は fastcall を強制させることも今後の課題として考えられる。

ちなみに、表の TCC とは “Tiny C Compiler” のことである。このコンパイラの詳細に付いては割愛するが、C のソースコードをアセンブラを介さずに直接オブジェクトファイルに変換することができる。本研究の前に TCC にも CbC コンパイル機能の実装を行ったが、表の通り満足の行く結果ではなかった(注1)。

(注1) しかしこれは実装手段が悪かったと思われる。goto の際に毎回 strcpy するようなことを改良すれば大幅高速化できるだろう。

第6章　今後の課題

本研究の実装により、GCC を使って CbC のソースコードをコンパイルすることができるようになった。また、これまでの Micro-C ベースのコンパイラではできなかった最適化を GCC 上で実行できるようになった。

しかし、未だに実装されてない CbC の構文がまだ残っている。また、一部アーキテクチャではコンパイル不能に陥る現象も発生している。これらの問題とその他改良点を今後の課題として以下に簡単に説明する。

environment 第 2 章では説明を省いたが、CbC にはもう一つ、environment 付きの継続という構文が存在する。これは関数から code segment に goto した場合に関数の呼び出し元に戻ることを可能にするものだが、今回この実装は間に合わなかった。

code segment ポインタの計算 今の実装では `goto cs->next(a, b);` のように呼び出し先 code segment を計算することができない。第 5 章の conv1.c では一旦別の変数に格納することによって回避している。

PPC の RTL 変換不能 PowerPC アーキテクチャにおいて、code segment のポインタ参照へ goto することができない。これは RTL レベルで対応されてないことが原因と思われる。

push_overlaps リスト 4.9 において、余計な変数まで一時変数に格納することがある。これは引数をスタックにおく順番を変えることで改良可能だ。

-O2 オプションの強制 CbC は-O2 オプションをつけないとコンパイルできない。なのでファイル名が.cbc の場合はこれを強制させる必要がある。

fastcall 第 5 章でも述べたように、code segment では fastcall を強制させることで高速化が期待できる。

この中から特に重要なのが environment と code segment ポインタの計算への対応だと考えている。この二つができればとりあえず CbC の現在の仕様を満たす。

これらに加えて、GCC にはすでに C++や Objective-C のコンパイルが可能である。これを活かし、CbC++, もしくは Objective-CbC といった既存の言語と CbC を組み合わせた言語に付いても考えてみる価値があるだろう。

参考文献

[1] 河野真治. “継続を基本とした言語 CbC の gcc 上の実装”. 日本ソフトウェア科学会第 19 回大会論文集, Sep, 2002.

[2] 河野真治. “継続を持つ C の下位言語によるシステム記述”. 日本ソフトウェア科学会第 17 回大会論文集, Sep, 2000.

[3] GNU Project - Free Software Foundation, GCC internal manual. “http://gcc.gnu.org/onlinedocs/gccint/”.

[4] 金城拓実. “軽量継続を用いたゲームプログラムの分割と再構成の考察”. 琉球大学情報工学科 学位論文, Feb, 2006.

付 録A　conv1 プログラム

以下は第 5 章 評価で使用したプログラム conv1 である。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  static int loop;
5  #define CC_ONLY 0
6
7  /* classical function call case (0) */
8  int f0(int);
9  int g0(int);
10 int h0(int);
11
12 f0(int i) {
13     int k,j;
14     k = 3+i;
15     j = g0(i+3);
16     return k+4+j;
17 }
18
19 g0(int i) {
20     return h0(i+4)+i;
21 }
22
23 h0(int i) {
24     return i+4;
25 }
26
27 #if !CC_ONLY
28
29 /* straight conversion case (1) */
30
31 typedef char *stack;
32
33 struct cont_interface { // General Return Continuation
34     __code     (*ret)(int, stack);
35 };
36
37 __code     f_g0(int i,int k,stack sp);
38 __code     f_g1(int j,stack sp);
39
40 __code     f(int i,stack sp) {
41     int k,j;
42     k = 3+i;
43     goto f_g0(i,k,sp);
44 }
45
46 struct f_g0_interface {  // Specialized Return Continuation
47     __code      (*ret)(int, stack);
48     int i_,k_,j_;
49 };
50
51 __code     g(int i,stack sp);
52 __code     f_g1(int j,stack sp) ;
53
54 __code     f_g0(int i,int k,stack sp) { // Caller
55     struct f_g0_interface *c =
56         (struct f_g0_interface *)(sp -= sizeof(struct f_g0_interface));
57
58     c->ret = f_g1;
59     c->k_ = k;
60     c->i_ = i;
61
62     goto g(i+3,sp);
63 }
64
65 __code     f_g1(int j,stack sp) {  // Continuation
66     struct f_g0_interface *c = (void *)sp;
67     int k = c->k_;
68     __code      (*ret)(int, stack);
69
70     sp+=sizeof(struct f_g0_interface);
71     c = (struct f_g0_interface *)sp;
72     ret = c->ret;
73     goto ret(k+4+j,sp);
74 }
75
76 __code     g_h1(int j,stack sp);
77 __code     h(int i,stack sp);
78 __code     g_h1(int j,stack sp);
79
80 __code     g(int i,stack sp) { // Caller
81     struct f_g0_interface *c =
82         (struct f_g0_interface *)(sp -= sizeof(struct f_g0_interface));
83
```

```
84      c->ret = g_h1;
85      c->i_ = i;
86
87      goto h(i+3,sp);
88  }
89
90  __code      g_h1(int j,stack sp) {  //
        Continuation
91      struct f_g0_interface *c = (void
            *)sp;
92      __code      (*ret)(int, stack);
93      int i = c->i_;
94
95      sp+=sizeof(struct f_g0_interface)
            ;
96      c = (struct f_g0_interface *)sp;
97      ret = c->ret;
98      goto ret(j+i,sp);
99  }
100
101 __code      h(int i,stack sp) {
102     struct f_g0_interface *c = (void
            *)sp;
103     __code      (*ret)(int, stack);
104     ret = c->ret;
105     goto ret(i+4,sp);
106 }
107
108 struct main_continuation { // General
        Return Continuation
109     __code      (*ret)(int, stack);
110     __code      (*main_ret)();
111     void *env;
112 };
113
114 __code      main_return(int i,stack sp
        ) {
115     if (loop-->0)
116         goto f(233,sp);
117     printf("#0103:%d?n",i);
118     exit(0);
119 }
120
121 /* little optimzation without stack
        continuation (2) */
122
123 __code g2(int i,int k,int j,char *sp)
        ;
124 __code h2_1(int i,int k,int j,char *
        sp) ;
125 __code h2(int i,int k,char *sp) ;
126 __code main_return2(int i,stack sp) ;
127 __code f2(int i,char *sp) ;
128
129
130 __code f2(int i,char *sp)
131 {
132     int k,j;
133     k = 3+i;
134     goto g2(i,k,i+3,sp);
135 }
136
137 __code g2(int i,int k,int j,char *sp)
        {
138     j = j+4;
139     goto h2(i,k+4+j,sp);
140 }
141
142 __code h2_1(int i,int k,int j,char *
        sp) {
143     goto main_return2(i+j,sp);
144 }
145
146 __code h2(int i,int k,char *sp) {
147     goto h2_1(i,k,i+4,sp);
148 }
149
150 __code main_return2(int i,stack sp) {
151     if (loop-->0)
152         goto f2(233,sp);
153     printf("#0132:%d?n",i);
154     exit(0);
155 }
156
157 /* little optimizaed case (3) */
158 __code g2_1(int k,int i,char *sp) ;
159 __code f2_0_1(int k,int j,char *sp);
160 __code h2_1_1(int i,int k,int j,char
        *sp) ;
161 __code h2_11(int i,int k,char *sp) ;
162 __code f2_0_1(int k,int j,char *sp) ;
163
164 __code f2_1(int i,char *sp) {
165     int k,j;
166     k = 3+i;
167     goto g2_1(k,i+3,sp);
168 }
169
170 __code g2_1(int k,int i,char *sp) {
171     goto h2_11(k,i+4,sp);
172 }
173
174 __code h2_1_1(int i,int k,int j,char
        *sp) {
175     goto f2_0_1(k,i+j,sp);
176 }
177
178 __code h2_11(int i,int k,char *sp) {
179     goto h2_1_1(i,k,i+4,sp);
180 }
181
182 __code f2_0_1(int k,int j,char *sp) {
183     __code      (*ret)(int, stack);
184     ret = ( (struct cont_interface *)
            sp)->ret;
185     goto ret(k+4+j,sp);
186 }
187
188 __code      main_return2_1(int i,stack
        sp) {
```

```
189     if (loop-->0)
190         goto f2_1(233,sp);
191     printf("#0165:%d?n",i);
192     exit(0);
193 }
194
195 #define STACK_SIZE 2048
196 stack main_stack[STACK_SIZE];
197 #define stack_last (&main_stack[
        STACK_SIZE])
198
199 #endif
200
201 #define LOOP_COUNT 0x10000000
202
203 void go_codesegment(int sw);
204 main(int ac,char *av[])
205 {
206     int sw;
207     if (ac==2) sw = atoi(av[1]);
208     else sw=3;
209
210     go_codesegment(sw);
211     return 0;
212 }
213
214 void go_codesegment(int sw)
215 {
216 #if !CC_ONLY
217     struct main_continuation *cont;
218     stack sp = (void*)stack_last;
219 #endif
220
221     int j=0;
222     if (sw==0) {
223         for(loop=0;loop<LOOP_COUNT;
                loop++) {
224             j += f0(233+loop);
225         }
226         printf("#0193:%d?n",j);
227 #if !CC_ONLY
228     } else if (sw==1) {
229         loop = LOOP_COUNT;
230         sp -= sizeof(*cont);
231         cont = (struct
                main_continuation *)sp;
232         cont->ret = main_return;
233         //cont->main_ret = return;
234         //cont->env = environment;
235         goto f(233,sp);
236     } else if (sw==2) {
237         loop = LOOP_COUNT;
238         sp -= sizeof(*cont);
239         cont = (struct
                main_continuation *)sp;
240         cont->ret = main_return2;
241         //cont->main_ret = return;
242         //cont->env = environment;
243         goto f2(233,sp);
244     } else if (sw==3) {
245         loop = LOOP_COUNT;
246         sp -= sizeof(*cont);
247         cont = (struct
                main_continuation *)sp;
248         cont->ret = main_return2_1;
249         //cont->main_ret = return;
250         //cont->env = environment;
251         goto f2_1(233,sp);
252 #endif
253     }
254 }
255
256 /* end */
```