

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

与儀 健人

河野 真治

琉球大学理工学研究科情報工学専攻

当研究室では Continuation based C という言語を提案しており、そのコンパイルにはこれまで Micro-C をベースにした独自のコンパイラを使用していた。また、GCC の Tail call optimization を用いた実装が可能である事を以前の論文で示した。ここでは GCC 上に実際に CbC 言語の実装し、その評価を行った。この実装はアーキテクチャに依存しないので、GCC が対応する全てのアーキテクチャ上で CbC が動く事になるはずであるが、若干の問題があり、その点に付いても考察を行う。

The implementation of Continuation based C Compiler on GCC

KENT yogi

Shinji KONO

Information Engineering, University Of the Ryukyus

We are approach Continuation based C Language, and have used Micro-C based Compiler that is developed by us to compile it. This research, We are implement a compiler for CbC into GCC and appraising it. This implementation is supposed to run on all architectures that is supported by GCC, but few problems are founded.

1 CbC について

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 1 は code segment 同士の関係を表したものである。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。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

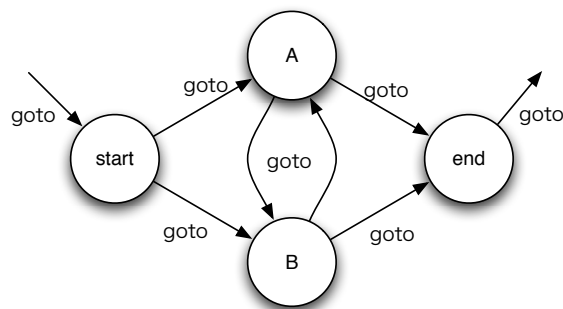


図 1: code segment 間の“継続”

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 segment 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。

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。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れることになる。

継続 (goto) は code segment 間の移動を表す。構文としては goto をつかっているが 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、return アドレスの push などを行わない。

2 GCC の構成

2.1 GCC の基本構造

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 パーサによってソースコードを解析する。

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。

gimplification Generic Tree をもとにこれを GIMPLE に変換する。

GIMPLE optimization GIMPLE に対して最適化を行う。

RTL generation GIMPLE をもとに RTL を生成する。

RTL optimization RTL に対して最適化を行う。

Output assembly 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 2 のように表される。各 pass は

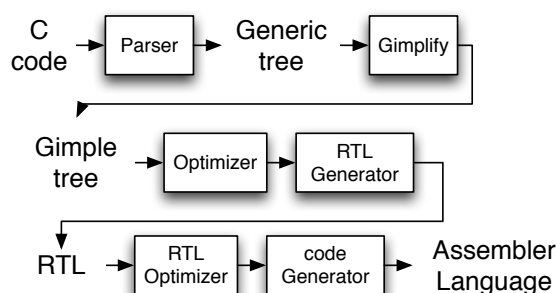


図 2: GCC の pass

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。

2.2 Tail call elimination

最適化のひとつである “Tail call elimination” 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。

2.2.1 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 に戻るようになる。“Tail call elimination” ではこの B から A に戻る無駄な処理を低減する。

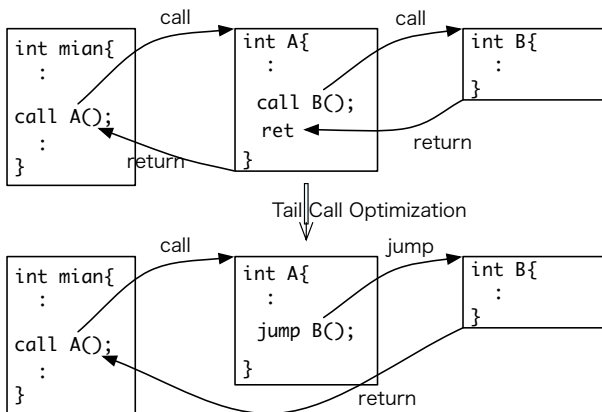


図 3: Tail call elimination の例

この様子を図 3 に示したので参考にしてください。

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いる i386 形式のアセンブラを使用している。

図 3 と同じように呼び出される関数 main, A, B をリスト 1 の様に定義する。

リスト 1: 関数 main A B の例

```
void B(int A, int A, int C){
    return ;
}
void A(int a, int b, int c, int d){
    return B(a, b, c+d);
}
int main(int argc, char **argv){
    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 2 のようなコードが出力される。(ここでは Tailcall 最適化が影響をあたえる関数 A のみをしめた)

リスト 2: 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
```

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リスト 3 に示す。

リスト 3: 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。

2.2.2 Tail call 時のスタック

このときのスタックの様子を図 4 に表した。図 4

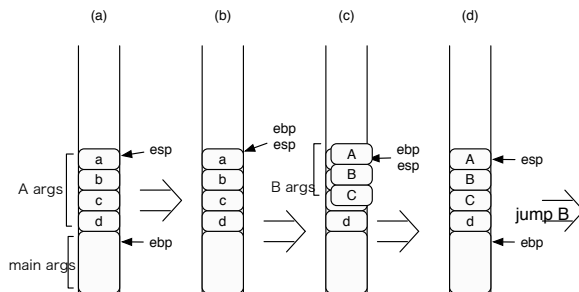


図 4: 関数 A から B を呼び出す時のスタックの様子

の各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いるが、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。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。

2.3 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 4 と、リスト 1 を説明に用いるので参考にさせていただきたい。

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。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。

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。例えば図 1 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。

3 実装

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。

以下の節ではそれぞれの行程について簡単に説明する。

3.1 __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: 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_declspecs 構造体で使用される。void 型なら cts_void, int 型なら cts_int など表されている。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 5: c.typespec.keyword 定義

```

enum c_typespec_keyword {
    :
    cts_int,
    cts_float,
    cts_double,
    cts_CbC_code,
    cts_dfloating32,
    :
};

```

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_declspececs 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 6: declspecs_add_type 関数

```

case RID_CbC_CODE:
if (specs->long_p)
    error ("both %<long%> and %<void ..>")
else if (specs->signed_p)
    error ("both %<signed%> and %<vo ..>")
    :
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_declspececs 関数は c_declspececs 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 7: finish_declspececs 関数

```

case cts_void:
case cts_CbC_code:
    specs->type = void_type_node;
break;

```

これで__code による型が void 型にマップされた。

3.2 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 8: build_code_segment_type 関数

```

tree
build_code_segment_type(value_type ...)
{
    tree t;
    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 9: `grokdeclarator` 関数

```
if (typespec_word == cts_CbC_code)
    type = build_code_segment_type (...);
else
    type = build_function_type (...);
```

これで、`__code` 型がパースされた時には `FUNCTION_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。

3.3 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 10: `goto` 文の構文解析

```
c_parser_consume_token (parser);
if (c_parser_next_token_is (parser, ...)
    && _parser_peek_2nd_token (pars...))
{
    stmt = c_finish_goto_label (...);
    c_parser_consume_token (parser);
} else {
    struct c_expr expr;
    expr = c_parser_postfix_expression (parser);
    if (TREE_CODE (expr) == CALL_EXPR) {
        CbC_IS_CbC_GOTO (expr) = 1;
        CALL_EXPR_TAILCALL (expr) = 1;
        stmt = c_finish_return (expr);
    } else
        c_parser_error (parser, ...);
}
```

`goto` トークンを読み込むと、次のトークンが識別子で、その次がセミコロンであれば通常の C における `goto` と判定できる。そうでなければ CbC の継続である。

3.4 expand_call の分割

ここで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。

3.5 expand_cbc_goto

簡単に説明すると、`expand_cbc_goto` は `expand_call` での `Tail call` の処理を可否判定無しで行うものとなる。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ついてソースコードを交えながら説明する。

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。

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。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。

引数の計算 引数の計算を行う。

リスト 11: 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] ..);
```

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。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出力するだけでよい。

オーバーラップ 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。

リスト 12: `push_overlaps` 関数

```
if (args[i].mode==BLKmode)
  emit_block_move(temp, args[i].value..)
else
  emit_move_insn(temp, args[i].value);
```

この関数は引数の読み込み元と書き込み先が重なる場合に、一部の引数を別の場所に退避する役割を持つ。この処理がリスト 12 である。 `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、対象引数の格納位置、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。

引数の格納 オーバー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 13: 引数の格納

```
emit_push_insn(arg->value, arg->mode ..);
```

CALL_INSN 最後に `CALL_INSN` を発行する処理が来る。

リスト 14: `CALL_INSN` の発行

```
funexp=rtx_for_function_call(fndecl..);
:
emit_call_1(funexp,exp,fndecl,fun ..);
```

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。

4 評価

今回実装できた GCC による CbC コンパイラでベンチマークを行い、Micro-C との比較を行った。

今回ベンチマーク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 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。測定結果は表 1 に示される。

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

	./conv1 1	./conv1 2	./conv1 3
Micro-C	8.97	2.19	2.73
GCC	4.87	3.08	3.65
GCC (+omit)	4.20	2.25	2.76
GCC (+fast)	3.44	1.76	2.34
TCC	122.28	84.91	102.59

表 1: Micro-C, GCC の実行速度比較 (単位 秒)

数に 1 を入れた場合、すなわち Micro-C 用に改良されていないコードでは 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ものである。また、“GCC (+fast)”はそれに加えてfastcallを用いたものである。その結果が表 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。

5 今後の課題と考察

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、未だに実装されていない 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 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。

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いないことが原因と思われる。

オプションの強制 -O2 オプションや、code segment への fseccall 属性の付加などを強制させる必要がある。

ここで、二つ目の PowerPC への対応がひとつ問題となっている。本来、このコンパイラはアーキテクチャに依存しない形で実装したのが、実装後、PPC は Tailcall elimination にたいして一部対応してないことがわかった。これは MachineDescription とよばれる RTL からアセンブラへの対応を表すファイルを記述することで対応させることができる。今回はこれには対応していない。

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] 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”. 琉球大学情報工科学位論文, Feb, 2006.
- [4] GNU Project - Free Software Foundation, GCC internal manual. “<http://gcc.gnu.org/onlinedocs/gccint/>”.