2008－OS－108 (4)
2008／4／23

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 archtectures 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る。

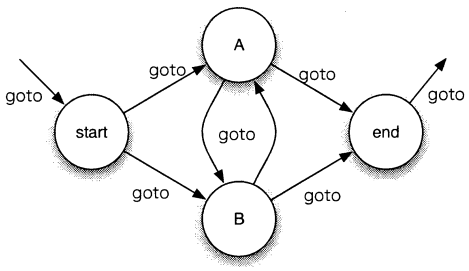

図 1: code segment 間の “継続”

これらの特徴から、CbCは自身でスケジューラの記述ができ、それにより並列処理や逐次処理をスムースに繋げることが出来る。また、OperatingSystemの記述やハードウェアの記述が容易になる。

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。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なども行わない。

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

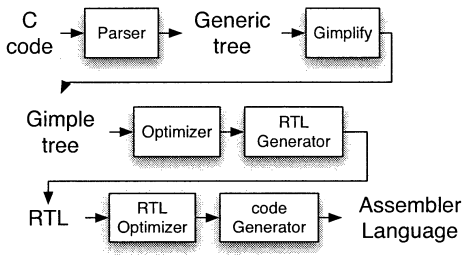

図2: GCCのpass

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。

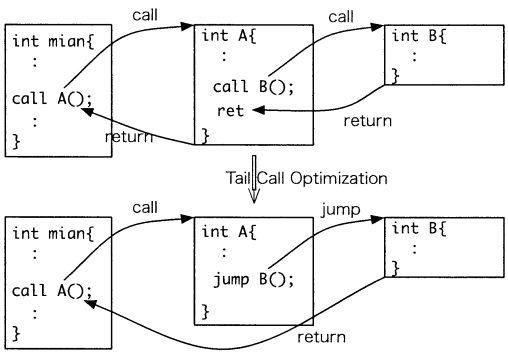

図 3: Tail call elimination の例

この様子を図 3 に示したので参考にしていただきたい。

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 のみをしめした)

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

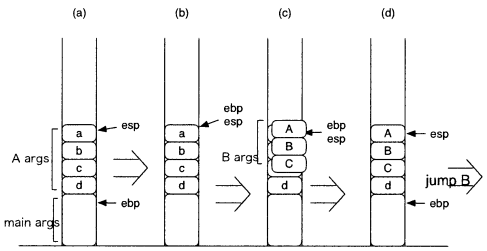

図 4: 関数 A から B を呼び出す時のスタックの様子

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。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` 配列に定義されており、ここに "__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、ここに cts_CbC_code を追加する。

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: 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_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 5: finish_declspecs 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 6: 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 7: 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 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。

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 8: 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(..);
    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

これらの処理はほぼ expand_call の内容をそのまま利用できる。ただし、4 のオーバーラップする引数がある場合のみ問題になる。goto の実装には 2.2 節で説明した Tail call を用いているため引数の書き込み領域と読み込み領域が重なる場合がある。本来この場合は Tail call 不能として通常の関数呼出が用いられるところであるが、CbC ではこれを強制しなければならない。そのため、このように重なる場合は変数の内容を一時退避する必要がある。次のリスト 9 がこの処理を書く引数に対して行っている。

リスト 9: push_overlaps 関数

```
push_overlaps(struct arg_data *args, int num_actu
  int i;
  for (i=0; i<num_actuals; i++)
  {
    int dst_offset;
    int src_offset;
    rtx temp;
    if (/*args[i].は    ス    タック    外
stack*/) continue;
    if (/*args[i].は    ス    タック    外
value*/) continue;
    temp =assign_temp(args[i].tree_value..);
    if ( args[i].mode==BLKmode )
      emit_block_move(temp,args[i].value..);
    else
      emit_move_insn(temp,args[i].value);
    args[i].value = temp;
  }
```

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のソースをプログラムでCbC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。一番結果の良い引数2の場合のcode segmentをリスト10に示す。

リスト 10: bench

```
__code  f2(int i,char *sp) {
    int k,j;
    k = 3+i;
    goto g2(i,k,i+3,sp);
}
__code g2(int i,int k,int j,char *sp){
    j = j+4;
    goto h2(i,k+4+j,sp);
}
__code h2_1(int i,int k,int j,char *sp){
    goto main_return2(i+j,sp);
}
__code h2(int i,int k,char *sp) {
    goto h2_1(i,k,i+4,sp);
}
```

このベンチマークではCbCの継続と計算を交互に行っている。測定結果は表1に示される。

	./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

表 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、その他は次節で説明するオプションを付加したものである。見てのとおり、手動で最適化された引数2,3の場合はオプションを加えなければMicro-Cの速度に及ばなかった。次節ではこの点について考察する。

4.1 出力コード

先ほどのリスト10のcode segment g2のみをMicro-Cでコンパイルした結果をリスト11, GCCのオプション無しによるコンパイル結果をリスト12に示す。

リスト 11: Micro-Cによる出力コード

```
f2:
    lea -_44(%ebp),%esp
    movl $3,%eax
    addl %esi,%eax
    movl %eax,-28(%ebp)
    movl %edi,-32(%ebp)
    movl -28(%ebp),%edi
    movl %esi,%eax
    addl $3,%eax
    movl %eax,-28(%ebp)
    jmp g2
```

リスト 12: GCCによる出力コード

```
f2:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    movl    12(%ebp), %ecx
    leal    3(%eax), %edx
    movl    %edx, 12(%ebp)
    movl    %edx, 16(%ebp)
    movl    %ecx, 20(%ebp)
    popl    %ebp
    jmp     g2
```

このとおり出力コードは10命令と、行数にはあまり差が無い。(他のsegmentも同様である) しかしGCCの出力においては無駄なコードが混じっていることがわかるだろう。`pushl%ebp` と `popl %ebp` である。すべてのcode segmentにおいてこれと同じ命令が出てしまっているが、これはTailcallを無理矢理適用したために出てきたコードである。

このような関数の最初と最後にある無駄なフレームポインタのpushを抑制するオプションが-fomit-frame-pointerである。このオプションを付加するとリスト13

リスト 13: GCCによる出力コード

```
f2:
    movl    4(%esp), %eax
    movl    8(%esp), %ecx
    leal    3(%eax), %edx
    movl    %edx, 8(%esp)
    movl    %edx, 12(%esp)
    movl    %ecx, 16(%esp)
    jmp     g2
```

これによって一気に3命令減った。ベンチマークは表1の3行目、"GCC (+omit)" である。しかし、(code segment にもよるが)3/10 命令減ったにもかかわらず Micro-C との速度差がほとんど無い。

リスト 11 をみると Micro-C では引数の格納にレジスタ%edi と%esi を用いる分、高速なコードを生成出来ていることが分かる。この違いが命令数の差を埋めている。GCC でも引数をレジスタに詰めることができる fastcall 属性がある。-fomit-frame-pointer に加えて fastcall を付加した結果をリスト 14 に示す。

リスト 14: GCC による出力コード

```
f2:
    movl    %edx, %eax
    leal    3(%ecx), %edx
    movl    %edx, 4(%esp)
    movl    %eax, 8(%esp)
    jmp     g2
```

命令数はさらに2命令減り、またメモリへのアクセスが減ったためベンチマーク結果 (表 1GCC (+fast)) も大幅に改善した。

この評価結果から、GCC の最適化オプションを用いることで CbC コードのさらなる高速化が可能であることが示された。また、使用したベンチマークプログラムは C のコードをプログラムで CbC に変換したものだが、これを C のままコンパイルすると最適化をかけても約 3.3 秒かかる。このように不要なスタック操作を減らすことによって、C 言語のみでは不可能な手動による最適化が CbC の利点としてあげられる。

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。

しかしまだいくつかの問題が残っているので、今後の課題と併せて、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 fasecall 属性の付加などを強制させる必要がある。

SPU 対応と GCC の version 実装できた version は 4.2.3 である。しかし現在 SPU に対応した GCC は 4.1 までしかでていないうえに、GCC の version 間の差異によって移植が難しくなっている。

ここで、二つ目の PowerPC への対応が大きな問題となっている。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、PowerPC 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はずだが、今回その実装には至らなかった。

またこれらに加えて、GCC 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今後実装していく。

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] Simon Peyton Jones, Thomas Nordin, and Dino Oliva, "C–: a portable assembly language". Implementing Functional Languages, 1997.

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