

Continuation based C の GCC 4.6 上の実装について

大城 信康[†] 河野 真治[†]

GCC-4.6 をベースとした CbC コンパイラの実装を行った。CbC のコンパイラは GCC-4.2 ベースのコンパイラが 2008 年に開発されており、以来 GCC のアップデートにあわせて CbC のコンパイラもアップデートが行われてきた。本論文では GCC-4.6 への CbC の具体的な実装について述べる。

The implementation of Continuation based C Compiler on GCC 4.6

NOBUYASU OSHIRO[†] and SHINJI KONO[†]

We implemented Continuation based C Compiler on GCC-4.6. CbC Compiler on GCC-4.2 was developed on 2008. Since then we kept to update it. In this paper, we introduce implemented Continuation based C Compiler on GCC-4.6.

1. 歴史的経緯

当研究室ではコードセグメント (Code Segment) 単位で記述するプログラミング言語 Continuation based C (以下 CbC) を開発している。コードセグメントは並列実行の単位として使うことができ、プログラムの正しさを示す単位としても使用することができる。これにより、Many Core での並列実行を高い性能と高い信頼性で実現することができると考えている。

CbC のコンパイルには元々 Micoro-C 版の独自のコンパイラを用いていたが、2008 年の研究において GCC-4.2 ベースの CbC コンパイラが開発され、2010 年には GCC-4.4 へとアップデートが行われた。GCC への実装により、GCC の最適化やデバッガの機能を使うことができより実用的な CbC プログラミングが行えるようになった。だが、GCC をベースとした CbC のコンパイラ (以下 CbC-GCC) は、GCC のアップデートに合わせて変更する必要がある。本研究では、GCC-4.5.0 をベースとしていた CbC-GCC を GCC-4.6.0 へのアップデートを行い、Intel64 に対応するとともに、CbC の拡張を行う。

2. Continuation based C (CbC)

CbC のプログラムはコードセグメント毎に記述され、コード間を goto(軽量継続) により処理を移る。構

文は C と同じであるが、ループ制御や関数コールが取り除かれる。

2.1 継続 (goto)

コードセグメントの記述は C の関数の構文と同じで、型に “`__code`” を使うことで宣言できる。コードセグメントへの移動は “goto” の後にコードセグメント名と引数を並べて記述することで行える。図 1 はコードセグメント間の処理の流れを表している。

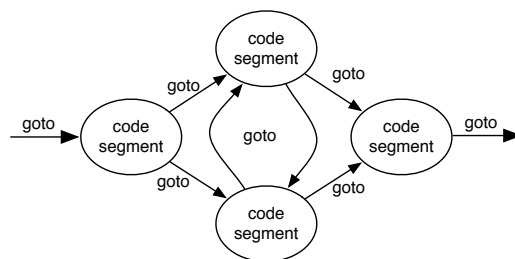


図 1 コードセグメント間の継続 (goto)

2.2 コードセグメント (code segment)

コードセグメントは C の関数と違って返り値を持たず、処理が終われば次のコードセグメントへと処理を移る。C において関数呼び出しを繰り返し行う場合、呼び出された関数の引数の数だけスタックに値が積まれていく。だが、返り値を持たないコードセグメントではスタックに値を積んでいく必要な無く、スタックは変更されない。

軽量継続により並列化、ループ制御、関数コールと

[†] 琉球大学

University of the Ryukyus

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。

図2はCbCで書いたプログラムの例である。与えられた数xの階上を計算して出力するプログラムとなっている。

```

_code print_factorial(int prod)
{
    printf("factorial = %d\n",prod);
    exit(0);
}

_code factorial0(int prod, int x)
{
    if ( x >= 1){
        goto factorial0(prod*x, x-1);
    }else{
        goto print_factorial(prod);
    }
}

_code factorial(int x)
{
    goto factorial0(1, x);
}

```

図2 階上を計算するCbCプログラムの例

3. GCCで扱われる内部表現

GCC-4.6への実装の前に、GCCで扱われる内部表現について触れておく。

GCCは内部でGeneric Tree, GIMPLE Tree, Tree SSA, RTLという4つの内部表現を扱う(GIMPLEとSSAは一緒に考えられることもある)。それぞれが読み込んだソースコードはGeneric Tree, GIMPLE Tree, Tree SSA, RTLの順に変換されていき、最後にアセンブラ言語へと出力される。図3はGCCがソースコードを読み込みアセンブラ言語出力までの流れを表した図である。

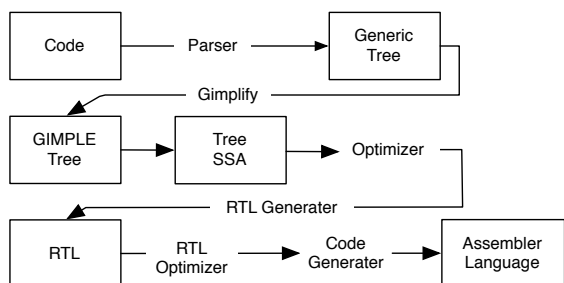


図3 GCCによるコンパイルの一連の流れ

3.1 Generic Tree

ソースコードより読み込んだ関数の情報を木構造で

表したものがGeneric Treeとなる。関数の戻値、引数、変数の型、条件式とプログラムの処理全てが木構造で表される。

CbCの実装ではparseの部分からこのGeneric Tree生成の部分に手が加わっている。

3.2 GIMPLE Tree

Generic Treeで表現されたデータはGIMPLE Treeに変換される。GIMPLE TreeはGeneric Treeより制約がかかった状態で作成された構文木となる。制約は「1つの枝につく子が3つ以下になるように分解」といったもので、GIMPLE Treeへと変換されたデータはGeneric Treeより簡単な命令で表されることになる。CbCの実装では特に修正は加えていない。

3.3 Tree SSA

Tree SSA (Static Single Assignment)は、プログラムの中で変数が一度しか代入されないように変換させた構文木になる。SSAに変換することで、様々な最適化が行いやすくなる。こちらもCbCの実装では特に修正は加えていない。

3.4 Register Transfer Language (RTL)

GIMPLE Treeは解析が行われた後RTLへと変換される。RTLはレジスタの割り当てといった低レベルの表現で、アセンブラとほぼ同じ表現を行うことができる。プログラム内部ではRTLも木構造で表される。

CbCにおける継続は、このRTLへの変換で行われる最適化の1つTail Call Eliminationが重要となってくる。

4. GCC-4.6への実装

前節まででCbCの基本仕様とGCCでのアセンブラ出力までの流れを確認した。ここからはGCC-4.6への実装について述べていく。

4.1 “__code”のパーズ

Cの予約後はgcc/c-family/c-common.cのc_common_reswords構造体で定義されている。ここに、図4のように__code型の登録を行う。

```

const struct c_common_resword c_common_reswords[] =
{
    {"_Bool",      RID_BOOL,   D_CONLY},
    {"_Complex",  RID_COMPLEX, 0},
    :
    /* CbC project */
    {"__code",    RID_CbC_CODE, 0},
    :
}

```

図4 __codeのパーズ

これで `__code` は `RID_CbC_CODE` として判定されるようになる。次に、`id` を用意する。Generic Tree が生成されるデータは一度 `c_declspecs` 構造体に保存される。そこに登録するコードセグメント判定用 `id` “`cts_CbC_code`” を用意する。これは `gcc/c-Tree.h` で定義される (図 5)。

```
enum c_typespec_keyword {
  cts_none,
  cts_void,
  :
  cts_CbC_code,
  :
};
```

図 5 cts_CbC_code の定義

後は `c_declspecs` 構造体にこの `id` を登録する。`id` の登録は `declspecs_add_type` 関数の中で行われる (図 6)。

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

図 6 id の登録 (declspecs_add_type 関数)

図 6 のプログラムは `void` 型の `id` 登録を元に作られている。違うところは `cts_CbC_code` を登録するところだけである。

最後に、`finish_declspecs` 関数にて `id` 毎に Tree タイプの決定をする。コードセグメントは `void` 型として扱ってもらうために `void_type_node` を Tree のタイプとして登録している (図 7)。

4.2 goto シンタックスの追加

通常 `goto` のシンタックスは “`goto ラベル名;`” となっている。CbC では通常の `goto` に加え “`goto cs();`”

```
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;
```

図 7 declspecs_add_type 関数

の形でコードセグメントを呼び出すシンタックスを追加する必要がある。図 8 は、追加した `goto` のシンタックスである (通常のシンタックスは省いてある)。

```
case RID_GOTO:
c_parser_consume_token (parser);
if (c_parser_next_token_is (parser, CPP_NAME)
    && c_parser_peek_2nd_token (parser)->type == CPP_SEMICOLON)
{
  :
  else
  {
    if (c_parser_next_token_is (parser, CPP_NAME))
    {
      tree id = c_parser_peek_token (parser)->value;
      location_t loc = c_parser_peek_token (parser)->location;
      build_external_ref (loc, id, RID_CbC_CODE, &expr.original_type);
    }
    expr = c_parser_expr_no_commas (parser, NULL);
    if (TREE_CODE(expr.value) == CALL_EXPR)
    {
      location_t loc = c_parser_peek_token (parser)->location;
      cbc_replace_arguments (loc, expr.value);

      TREE_TYPE(expr.value) = void_type_node;
      /*tree env = NULL_TREE;*/
      CbC_IS_CbC_GOTO (expr.value) = 1;
      CALL_EXPR_TAILCALL (expr.value) = 1;
      add_stmt(expr.value);
      stmt = c_finish_return(loc, NULL_TREE, NULL_TREE);
    }
    else
      c_parser_error (parser, "expected code segment jump or %<*>");
  }
}
```

図 8 goto へのシンタックスの追加

具体的には `void` 型の Tree を作成している。加えてコードセグメントと判定するフラグと Tail Call のフラグを付けた関数とした Tree となる。

`cbc_replace_arguments` 関数と `c_finish_return` 関数の動作については CbC においても重要になるので後に詳しく説明する。

4.3 Tail Call Elimination

CbC の継続の実装には GCC の最適化の 1 つ、Tail Call Elimination (末尾除去) を強制することで実装する。これにより、コードセグメント間の移動を、`call` ではなく `jmp` 命令で実現する。図 9 は Tail Call Elimination が行われた際のプログラムの処理を表している。

`funcB` は `jmp` 命令で `funcC` を呼び出す。funcC は、

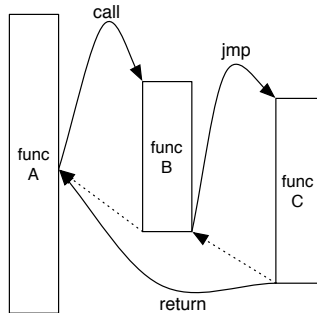


図 9 Tail Call Elimination

戻値を funcB ではなく funcA へと返すことになる。

4.3.1 expand_call

expand_call 関数は、関数を表す Tree から RTL を生成する関数である。Tail Call Elimination を行えるかどうかはこの関数で判断される。内部でチェックされる Tail Call Elimination の条件は以下になる。

- (1) caller 側と callee 側の戻値の型が一致している。
- (2) 関数呼び出しがリターンの直前に行われている。
- (3) 呼出先関数の引数に用いられるスタックサイズが呼出元関数のそれより少ない。
- (4) 引数の並びのコピーに上書きがない。

CbC の実装では上記の条件を、以下の様にして解決させている。

- (1) コードセグメントは void 型で統一する。最適化 (-O2) の強制付与。
- (2) goto の直後に return を置く。
- (3) スタックサイズは関数宣言時に決まったサイズにする。
- (4) 引数は一旦、一時変数にコピーして重なりがないようにする。

戻値を持たない為、コードセグメントを void 型で統一するのは自明だろう。最適化の強制付与及び 2, 3 と 4 については以下で詳しく説明を行う。

4.3.2 末尾最適化の強制付与

Tail Call Elimination は C のプログラムにおいて末尾最適化を有効にすることで行われる。以前の CbC-GCC の実装では expand_call 関数を元にした expand_cbc_call 関数を作成して、条件をクリアするようにしていた。しかし、その方法では expand_call 関数が改良される度に expand_cbc_call 関数にも変更を加える必要があり、手間となっていた。そこで、最適化フラグを強制的に付与させることで expand_cbc_call 関数を取り除くことに成功した(図 10:2 行目)。

```

1 | if (currently_expanding_call++ != 0
2 |   || ((!fndecl || !CbC_IS_CODE_SEGMENT (TREE_TYPE (fndecl)))
      && !flag_optimize_sibling_calls)
3 |   || args_size.var
4 |   || dbg_cnt (tail_call) == false)
5 |   try_tail_call = 0;

```

図 10 コードセグメントの末尾最適化の付与

expand_call 関数内では、Tail Call Elimination にかけるためのフラグ、try_tail_call 変数があり、コードセグメントはこのフラグには初め 1 がセットされている。コードセグメントの時はこの try_tail_call 変数に 0 を代入させないように実装を行った。また、万が一 try_tail_call 変数に 0 を代入された時の為にフラグに 1 を代入するコードの挿入も行った。これにより末尾最適化の強制付与がなされた。

4.3.3 goto の直後に return の配置

図 2 のコードセグメント factorial0 を listing11 の様に、goto の直後に return を置く必要がある。だがそれをプログラマが記述することは実用的でない。

```

_code factorial0(int prod, int x)
{
  if ( x >= 1 ) {
    goto factorial0(prod*x, x-1);
    return;
  }else{
    goto print_factorial(prod);
    return;
  }
}

```

図 11 goto の直後に return を置く

CbC では Generic Tree の生成時に継続の直後に return を自動で組み込むことで解決している。図 8 の c_finish_return 関数がそれに当たる。

4.3.4 スタックサイズの固定化

CbC では継続によりスタックに値が積まれていくということはない為サイズを固定することができる。また、サイズが固定な為、スタックポインタを変えずにスタックを扱うことができる。これも CbC の 1 つの特徴である。図 12 はコードセグメントの継続の際にスタックに積まれる引数を表している。

4.3.5 引数の並びの上書きコピー

CbC の継続では、引数渡しでスタックを入れ替える為値が書き換えられる可能性がでてくる。例えば listlisting??のような継続である。

この時のスタックの様子を表したのが図 14 となる。

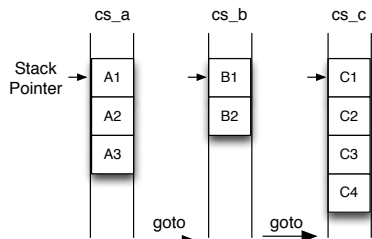


図 12 継続による引数のスタック格納の様子

```

__code cs_a(int a, int b) {
  goto cs_b(b, a)
}

```

図 13

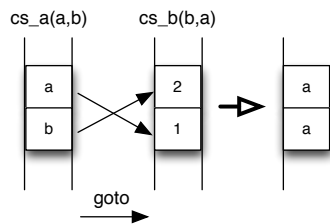


図 14 スタック書き換えの問題

数字の 1 と 2 は `cs_b` の引数をスタックに乗せる順を表している。CbC ではこの問題を一時変数に引数の値を代入することで問題を解決している。

4.3.6 一時変数へのコピー

一時変数へのコピーは、`goto` が行われるコードセグメントの Generic Tree 生成時に行われる。

図 15 に示す `cbc_replace_arguments` 関数が実際のコードとなる。

具体的には、内部で以下の事が行われている。

- 引数と同じ型の一時的変数を作成
- 一時変数に引数の値を代入
- 関数に渡す引数のポインタを一時的変数に変更

Tree call は継続を行うコードセグメントを指す。コードセグメントに渡された引数の情報を抜き出す部分が `FOR_EACH_CALL_EXPR_ARG` である。 `tmp_decl` という一時変数を作り、最後に `CALL_EXPR_ARG` を使って引数の情報が入っていたところへ一時変数を入れている。

4.4 環境付き継続

CbC には通常の C の関数からコードセグメントに継続する際、その関数から値を戻す処理への継続を得ることができる。これを環境付き継続という。これら

```

static tree
cbc_replace_arguments (location_t loc, tree call)
{
  tree arg;
  tree fn;
  tree tmp_decl;
  int i=0;
  call_expr_arg_iterator iter;

  fn = CALL_EXPR_FN (call);
  if ( TREE_CODE (fn)==PARAM_DECL || ITREE_CONSTANT (fn) )
    {
      tmp_decl = build_decl (loc, VAR_DECL, NULL_TREE, TREE_TYPE(fn));
      pushdecl (tmp_decl);

      add_stmt (build_modify_expr (loc, tmp_decl, NULL_TREE, NOP_EXPR,
loc, fn, NULL_TREE));
      CALL_EXPR_FN (call) = tmp_decl;
    }

  FOR_EACH_CALL_EXPR_ARG (arg, iter, call)
    {
      if ( TREE_CODE (arg)==PARAM_DECL || ITREE_CONSTANT (arg) )
        {
          tmp_decl = build_decl (loc, VAR_DECL, NULL_TREE,
TREE_TYPE(arg));
          pushdecl (tmp_decl);

          add_stmt (build_modify_expr (loc, tmp_decl, NULL_TREE,
NOP_EXPR, loc, arg, NULL_TREE));
          CALL_EXPR_ARG (call, i) = tmp_decl;
        }
      i++;
    }

  return call;
}

```

図 15 引数の一時変数へのコピー

は、以下の二種類の CbC で定義した特殊変数である。 `__environment` は、環境を表す情報である。 `__return` は、これを環境付き継続の行き先であり、関数の戻値と `__environment` の二つの引数を持つコードセグメントである。例えば、図 16 のように使うと、 `main()` は 1 を返す。

```

__code c1(__code ret(int,void *),void *env) {
  goto ret(1,env);
}

int main() {
  goto c1(__return, __environment);
}

```

図 16

GCC 内部では、 `__return` は、関数内で定義された `_cbc_internal_return` 関数へのポインタを返す。戻値は、 `cbc_internal_return` 関数内で定義された変数 `retval` を通して返される (図??)。

4.4.1 環境付き継続の問題

現在環境付き継続はこのコードを GCC 内部で生成することで実現している。これは正しく動作しているが、 `retval` に `static` を指定してしまうと、スレッドセーフな実装でなくなる。これを通常の変数にすると、

```

_label __cbc_exit0;
static int retval;
void __cbc_internal_return(int retval_,
                           void *_envp){
    retval = retval_;
    goto __cbc_exit0;
}
if (0) {
    __cbc_exit0:
    return retval;
}
__cbc_internal_return;

```

図 17 環境付き継続を行うコード

関数内の関数は closure として実装される。しかし、GCC 4.6 と Lion の組合せでは closure は正しく動作してないことがわかった。Thread local 変数を用いると、やはり closure が出力されてしまう。本来、戻用のレジスタが使用されれば問題ないが、戻値の型は整数やポインタとは限らず、浮動小数点や構造体自体である可能性があり複雑である。一つの解決策はレジスタ渡しと考えているが、他の方法もありえる。少し重いですが setjmp を用いた実装方法もある。

4.5 引数渡し

通常コードセグメントの継続において、引数は C の関数と同じスタックを用いて渡される。GCC には引数渡しをスタックではなくレジスタを用いて行う機能として fastcall がある。fastcall を用いてコードセグメント間を継続することで、速度の向上を図る。

4.5.1 fastcall

C において fastcall を用いる場合は関数にキーワード “__attribute__ ((fastcall))” をつけて行う。だが、コードセグメントを全てこのキーワードをつけて宣言することは実用的ではない。そこで、コードセグメントで宣言された場合、fastcall が自動で付くように実装を行う。図 18 はコードセグメントの生成を行なっているコードである。

13,14 行目が fastcall 属性を付与している部分になる。if 文で条件を決めているのは、64 bit の場合 fastcall が標準で行われていて、warning を出すからである。

4.6 typedefrec の実装の構想

C では関数や構造体の宣言の時に自分自身を引数にすることができない。そこで “typedefrec” という構文を作り、図??のような宣言を行えるようにしたい。

typedefrec によりコードセグメントは自分自身に戻る構成ができるようになる。より柔軟なプログラミングが行えるように typedefrec の実装を行う予定である。

```

1 | case RID_CbC_CODE:
2 | if (!typespec_ok)
3 |     goto out;
4 |     attrs_ok = true;
5 |     seen_type = true;
6 |     if (c_dialect_objc ())
7 |         parser->objc_need_raw_identifier = true;
8 |     t.kind = ctsk_resword;
9 |     t.spec = c_parser_peek_token (parser)->value;
10 |     declspecs_add_type (loc, specs, t);
11 |
12 | if (!TARGET_64BIT) {
13 |     attrs = build_tree_list (get_identifier("fastcall"), NULL_TREE);
14 |     declspecs_add_attrs(specs, attrs);
15 | }
16 | c_parser_consume_token (parser);
17 | break;

```

図 18 コードセグメントへの fastcall 属性付与

```

typedefrec void *funcA(int, funcA);

typedefrec struct {
    NODE left;
    NODE right;
} *NODE;

```

図 19 typedefrec の例

5. 評価

今回実装を行った GCC-4.6 ベース、GCC-4.4 ベース、Micro-C の CbC コンパイラでベンチマークを行った Micro-C のベンチマークで使用されていたプログラムを使う。また、通常の C のプログラムを CbC へと変換した。このプログラムは CbC の継続と計算を交互に行う。引数 1 はただ CbC へと変換したプログラム、引数 2 と 3 は Micro-C 用に手で最適化を行ったプログラムになる。

比較を行うのは以下のアーキテクチャと OS になる。

- x86/Linux
- x86/OS X

32 bit, 64 bit の動作も確認する。また、最適化無し (-O0) と速度最適化 (-O2 -fomit-frame-pointer) にかけたコードの比較を行う。比較の結果を図 20, 21 に示す。ただし GCC-4.6 の最適化無しコードは、コードセグメントに対して末尾最適化を強制したことが原因で segmentation fault を起こす為除外している。(また Micro-C の 64bit 版は Linux では動かなかった為 OS X だけとなっている。)

5.1 評価の考察

最適化無しの結果では Micro-C が早いですが、最適化有りでは GCC バージョンの方がどれも Micro-C に 2.5 倍程の差をつけている。最適化有りの GCC は 64bit

Linux		./conv1 1	./conv1 2	./conv1 2
MC(32bit)		6.70	4.52	3.97
gcc 4.4	-m32 -O0	22.98	10.04	12.59
	-m64 -O0	25.56	10.59	12.96
	-m32 -O2	4.71	1.98	2.41
	-m64 -O2	3.09	1.13	1.67
gcc 4.6	-m32 -O2	1.91	1.85	1.07
	-m64 -O2	1.58	1.17	1.07

図 20 それぞれのコンパイラにより生成されたコードの速度比較 (Linux)

OS X		./conv1 1	./conv1 2	./conv1 2
MC(32bit)		9.93	6.31	7.18
MC(64bit)		5.03	5.12	5.00
gcc 4.4	-m32 -O0	20.26	10.23	12.33
	-m64 -O0	20.49	10.84	12.75
	-m32 -O2	6.00	2.35	2.83
	-m64 -O2	3.05	1.16	1.37
gcc 4.6	-m32 -O2	2.52	2.34	1.53
	-m64 -O2	1.80	1.20	1.44

図 21 それぞれのコンパイラにより生成されたコードの速度比較 (OS X)

の方が 32bit より 2 倍程結果が優れているのも確認できる。4.4, 4.6 での違いは指数が 1 の時である。4.6 が 2 倍近く早くなっている。これは、4.6 の方が 4.4 よりも最適化が進んでいる為だと思われる。

6. CbC のアップデート手法

最後に、CbC のアップデート手法について述べる。

現在 GCC は年に数回アップデートが行われている。GCC に合わせて CbC のアップデートを行うのが好ましいが、その度新しいソースコードに合わせていくのは負担が大きい。GCC の正式な機能として CbC を組み込んで貰うことが最良の方法だが現時点ではまだそこまで至っていない。

そこで Mercurial を使ってアップデート方法を行っている。

6.1 Mercurial によるアップデート

Mercurial はバージョン管理システムである。当研究室では CbC のソースコードは Mercurial によって管理されている。Mercurial では本家 GCC のソースコードも管理されており、これら 2 つのリポジトリを使って CbC のアップデートは行われる (図??)。具体

的な方法は以下になる。

- GCC リポジトリ
 - (1) GCC リポジトリの中身を削除 (バージョン管理情報以外)
 - (2) 新しい GCC のソースを入れる
 - (3) hg status で追加ファイルと削除ファイルを確認
 - (4) 追加, 削除するファイルに対して hg add, hg remove を行う
 - (5) コミット
 - (6) gcc version タグを追加
- CbC リポジトリ
 - (1) GCC リポジトリから hg pull を行う
 - (2) hg merge でマージを行う
 - (3) 衝突が発生したファイルのマージを行う
 - (4) ビルドを行い動作確認
 - (5) コミット
 - (6) gcc version タグを追加



図 22 当研究室で管理している GCC リポジトリのグラフ

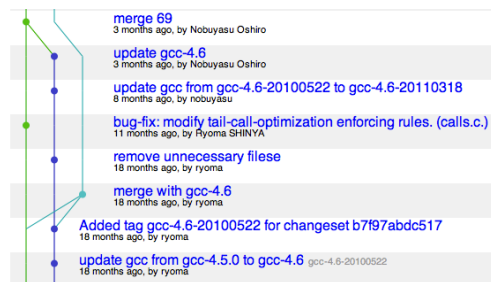


図 23 CbC-GCC リポジトリのグラフ

6.2 リポジトリ管理方法の評価

上記のリポジトリ管理方法を用いて GCC-4.5.0 から GCC-4.6.0 へのアップデートを行った。この手法を用いない場合は手動で diff を行い差分を探すことになる。だが、上記の手法ではほとんどの差分の適用を Mercurial 自身がおこなってくれた。手動で差分を直したのは CbC の実装を行ったファイルだけで済

んだ。CbC の為のコードが入り、かつ移動されたファイルがあり、それらだけは 2 つのバージョンを並べて手動で diff を行うことになったが、ファイル自体が少ない為すぐに差分の適応は行えた。

7. まとめと今後の課題

今回 CbC コンパイラを GCC-4.6 へとアップデートを行った。アップデートに伴い不具合の修正と Intel64 ビットへの対応を行った。だが、環境付き継続等未だ幾つかの問題を残している。また、typedefrec の様に新たに実装を行いたい機能もでてきている。今後はこの問題の解決と typedefrec の実装を行い、CbC-GCC の実装を突き詰めていく。また、CbC を Google Go 言語での実装等の研究も行う予定である。

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 の回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) 河野真治. Implementing continuation based language in gcc. *Continuation Festa*, April 2008.
 - 4) 与儀健人, 河野真治. Continuation based c コンパイラの gcc-4.2 による実装. 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.
 - 5) 与儀健人, 河野真治. Continuation based c コンパイラの gcc-4.2 による実装. 琉球大学 情報工学科 学位論文, Feb 2008.
 - 6) 楊挺, 河野真治. 継続を基本とする cbc による分散計算. 沖縄情報通信ワークショップ, Nov 2002.
 - 7) GNU Compiler Collection (GCC) Internals. <http://gcc.gnu.org/onlinedocs/gccint/>.
-