



```

--code print_factorial(int prod)
{
    printf("factorial of %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

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。

### 3. GCC の 3 つの内部表現

GCC-4.6 への実装の前に、GCC で扱われる 3 つの内部表現について触れておく。

#### 3.1 3 つの内部表現

GCC は内部で Generic Tree, GIMPLE, RTL という 3 つの内部表現を扱う。それぞれが読み込んだソースコードは Generic Tree, GIMPLE, 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。

##### 3.1.1 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。関数の戻り値、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。

##### 3.1.2 GIMPLE

Generic Tree で表現されたデータは GIMPLE というデータ構造に変換される。GIMPLE も Generic Tree と同じ構文木だが、より制約がかかった状態で

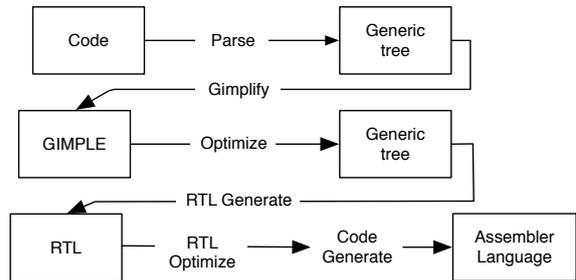


図 3 GCC によるコンパイルの一連の流れ

作成された構文木となる。制約は「1 つの枝に 4 つ以上の子を持たせない」等といったもので、GIMPLE 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り最適化がかけやすくなる。CbC の実装では特に修正は加えていない。

#### 3.1.3 Register Transfer Language (RTL)

構文木 GIMPLE 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 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 4 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、戻り値を funcB ではなく funcA へと返すことになる。

##### 4.1.1 expand\_call

ある関数が Tail Call Elimination を行えるかどうかは expand\_call 関数で判断される。expand\_call 関数内でチェックされる Tail Call Elimination が行える条件は以下になる。

- caller 側と callee 側の戻値の型が一致している。
- 関数呼び出しがリターン直前に行われている。

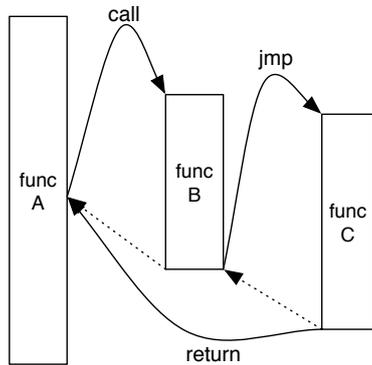


図 4 Tail Call Elimination

- 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。
- 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。

- コードセグメントは void 型で統一する。C の関数からコードセグメントに goto する場合は返す値の型チェックを行わない。
- goto の直後に return を置く。
- スタックサイズは関数宣言時に決まったサイズにする。
- 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。

スタックサイズを決め打ちで行うことで、ベース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 5 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示している。

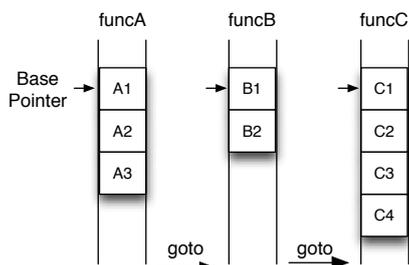


図 5 継続による引数のスタック格納の様子

GCC では、この他にも TCE を禁止するルールがあり、GCC-4.5, 4.6 でも Tail Call Elimination にかからないコードセグメントがある。この点を改善する

必要がある。

#### 4.2 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、以下のように使うと、main() は 1 を返す。

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

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

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 (Listing1)。

Listing 1 環境付き継続を行うコード

```
__label__ _cbc_exit0;
static int retval; // should be thread local
void _cbc_internal_return(int retval_, void *_envp){
    retval = retval_;
    goto _cbc_exit0;
}
if (0) {
    _cbc_exit0:
    return retval;
}
_cbc_internal_return;
```

#### 4.2.1 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、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。本来は、戻

値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。少し重いが setjmp を用いた実装方法もある。

#### 4.3 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を宣言することで、レジスタを用いた速度の向上を図る。

##### 4.3.1 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できではない。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 6 はコードセグメントに fastcall 属性を付与しているソースである。

```

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 |
17 |   c_parser_consume_token (parser);
18 |   break;

```

図 6 コードセグメントへの fastcall 属性付与

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、fastcall 属性を付けると warning を出すからである。

#### 4.4 typedefrec の実装の構想

C を基本とした CbC には型推論がない。

### 5. 評価

今回実装を行った GCC-4.6 ベースのコンパイラを GCC-4.4 ベース、Micro-C コンパイラとそれぞれ比較を行った。比較を行うのはクイックソートのプログラムである。クイックソートは再帰的にプログラムさ

Listing 2 typedefrec の例

```

typedefrec void *funcA(int, funcA);

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

```

れる為 CbC に向いているプログラムだと言える。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

また、比較を行うプログラムは最適化 (-O0 オプション) を行わないものと、速度最適化 (-O2 -fomit-frame-pointer) を行うものの 2 つ、それと -m32 オプションと -m64 オプションをつけたものそれぞれで行う。

表 1 が最適化無し、表 2 が速度最適化有りとなる。

CPU/OS	GCC-4.4	GCC-4.6	Micro-C
x86/Linux	7.378	0.829	2.890
x86_64/OS X(-m32)	3.890	0.382	2.288
x86_64/OS X	4.078	0.450	

表 1 アーキテクチャ毎の GCC と micro-c の速度比較 (単位: 秒)(最適化無し)

CPU/OS	GCC-4.4	GCC-4.6	Micro-C
x86/Linux	3.252	2.906	2.890
x86_64/OS X(-m32)	1.827	0.934	2.288
x86_64/OS X	1.101	2.896	

表 2 アーキテクチャ毎の GCC と micro-c の速度比較 (単位: 秒)(速度最適化)

### 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) 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.
- 4) GNU Compiler Collection (GCC) Internals. <http://gcc.gnu.org/onlinedocs/gccint/>.