LLVM, Clang 上の Continuation based C コンパイラの改良

Improvement of Continuation based C compiler on LLVM and Clang

平成27年度 学位論文(修士)

琉球大学大学院 理工学研究科
情報工学専攻

徳森 海斗

要 旨

Continuation based C (CbC) は本研究室で開発されている code segment, data segment を用いてプログラムを記述する言語である。CbC コンパイラは micro-c ベース、 GCC ベース、 LLVM, clang ベースのものが存在する。LLVM, clang 上に実装された CbC コンパイラはその最適化機能を有効に利用でき、GCC で正しくコンパイル出来ないことのあるコードもコンパイル可能という利点がある反面、実行速度で劣るという面があった。

本研究では LLVM, clang 上に実装された CbC コンパイラに最適化、機能の追加を行った。これにより、以前のバージョンのものよりも高速なアセンブリを出力できるようになる、CbC の記述が楽になる、という結果が得られた。

Abstract

目 次

図 目 次

表 目 次

第1章　研究目的

プログラミングに用いられる単位として関数, クラス, オブジェクト等が存在するが, これらは容易に分割, 結合することは出来ない. また, アセンブリ言語は分割, 結合を行うことは容易であるが, これのみでプログラムを記述することは困難である.

これらの問題を解決するべく, 設計された単位が code segment, data segment である. code segment, data segment は分割, 結合を容易に行うことのできる処理, データの単位として設計されたものであり, 並列プログラミングフレームワーク Cerium[1], 分散ネットワークフレームワーク Alice[2], プログラミング言語 Continuation based C (CbC)[3] はこれらの単位を用いている.

CbC のコンパイラは micro-c をベースにしたものと GCC をベースにしたものに加え, 2014 年の研究で LLVM, clang をベースにしたものが存在する. 本研究では, LLVM, clang をベースとした CbC コンパイラにさらなる最適化, 機能の追加, Gears OS の記述をサポートする機能の設計を行った.

第2章　Continuation based C (CbC)

CbC の構文は C と同じであるが, for 文, while 文といったループ制御構文や関数呼び出しを取り除き [注1], code segment と goto による軽量継続を導入している. 以下の図 2.1 は code segment 同士の関係を表したものであり, 図中の丸が code segment を, 矢印が goto による継続を表している.

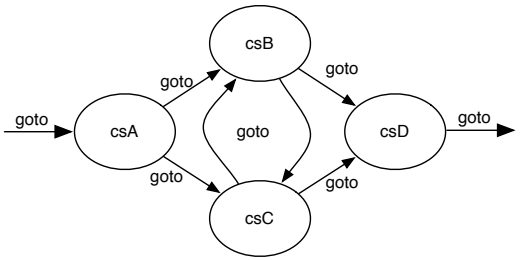

図 2.1: goto による code segment 間の継続

2.1　CbC における Code Segment

CbC では処理の単位として code segment を用いる。code segment は CbC における最も基本的な処理単位であり, C の関数と異なり戻り値を持たない. code segment の宣言は C の関数の構文と同じように行い, 型に __code を用いる. ただし, これは __code 型の戻り値を返すという意味ではない. 前述した通り, code segment は戻り値を持たないので, __code はそれが関数ではなく code segment であることを示すフラグのようなものである. code segment の処理内容の定義も C の関数同様に行うが, 前述した通り CbC にはループ制御構文が存在しないので, ループ処理は自分自身への再帰的な継続を行うことで実現する.

現在の code segment から次の code segment への処理の移動は goto の後に code segment 名と引数を並べて記述するという CbC 独自の構文を用いて行う. この goto による処理の遷移を継続と呼ぶ. 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が, 戻り値を持たない code segment ではスタックに値を積んでいく必要が無くスタックは変更されない. このようなスタックに値を積まない継続, つまり呼び出し元の環境を持たない継続を軽量継続と呼び, 軽量継続により並

[注1] 言語仕様としては存在しないが while や for を使用することは可能である.

列化, ループ制御, 関数コールと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.

以下に CbC を用いたコードの例として, 与えられた数値の階乗を算出するプログラムを示す. このコードの factorial0 という code segment に注目すると, 条件判別を行い, その結果に応じて自分自身への再帰的な継続を行うか別の code segment への継続を行うかという処理を行っていることがわかる. CbC ではこのようにしてループ処理を制御する.

ソースコード 2.1: 階乗を求める CbC プログラムの例

```
1  __code print_factorial(int prod)
2  {
3     printf("factorial␣=␣%d\n",prod);
4     exit(0);
5  }
6
7  __code factorial0(int prod, int x)
8  {
9     if ( x >= 1) {
10       goto factorial0(prod*x, x-1);
11    }else{
12       goto print_factorial(prod);
13    }
14
15 }
16
17 __code factorial(int x)
18 {
19    goto factorial0(1, x);
20 }
21
22 int main(int argc, char **argv)
23 {
24    int i;
25    i = atoi(argv[1]);
26
27    goto factorial(i);
28 }
```

2.2 環境付き継続

環境付き継続は C との互換性のために必要な機能である. CbC と C の記述を交える際, CbC の code segment から C の関数の呼び出しは問題なく行える. しかし, C の関数から CbC の code segment へと継続する場合, 呼び出し元の環境に戻るための特殊な継続が必要となる. これを環境付き継続と呼ぶ.

この環境付き継続を導入した言語は C with Continuation (CwC) と呼ばれ, C と CbC の両方の機能を持つ言語となる. また, C, CbC は CwC のサブセットと考えられるので, CwC のコンパイラを CbC に使用することができる. これまでに実装されてきた CbC コンパイラは実際には CwC のコンパイラとして実装されている.

環境付き継続を用いる場合, C の関数から code segment へ継続する際に __return, __environment という変数で表される特殊変数を渡す. __return は環境付き継続先が元

の環境に戻る際に利用する code segment, __environment は元の関数の環境を表す. リスト 2.2 では関数 funcB から code segment cs に継続する際に環境付き継続を利用している. cs は funcB から渡された code segment へ継続することで元の C の環境に復帰することが可能となる. 但し復帰先は __return を渡した関数が終了する位置である. このプログラムの例では, 関数 funcA は戻り値として funcB の終わりにある -1 ではなく, 環境付き継続によって渡される 1 を受け取る. 図 2.2 にこの様子を表した.

ソースコード 2.2: 環境付き継続

```
1  __code cs(__code (*ret)(int, void*), void *env){
2    /* C0 */
3    goto ret(1, env);
4  }
5
6  int funcB(){
7    /* B0 */
8    goto cs(__return, __environment);
9    /* B1 (never reached). */
10   return -1;
11 }
12
13 int funcA(){
14   /* A0 */
15   int retval;
16   retval = funcB();
17   /* A1 */
18   printf("retval = %d\n", retval);
19   /* retval should not be -1 but be 1. */
20 }
```

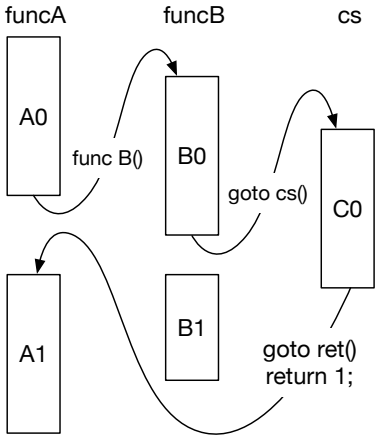

図 2.2: 環境付き継続

このような形にすることで, code segment 側では関数から呼ばれたか, code segment からの継続かを考慮する必要がなくなる. また, funcA から見た場合にも, 呼び出した関数の内部で code segment が使われているかどうかが隠蔽され, code segment の有無を考慮しなくて良い.

2.3 Gears OS サポート

Gears OS は当研究室で開発している並列フレームワークで, CbC で記述している. Gears では通常の CbC には存在しないメタレベルの処理を表す meta code segment, データの単位である data segment, data segment や code segment 等の情報を管理する context 等がある. これらを現在の CbC の機能のみを用いて記述するとリスト 2.3 のようになり, 多くの労力を要する. そのためこの記述を助ける機能が必要であり, 本研究ではこれらを利用するプログラミングをサポートするために以下の機能を提案した.

- code segment から meta code segment への自動接続
- 継続時に context から必要な情報を取得する stub の自動生成
- code segment 内での context の隠蔽

ソースコード 2.3: Gears OS コード例

```
1  __code meta(struct Context* context, enum Code next) {
2      goto (context->code[next])(context);
3  }
4
5  __code code1_stub(struct Context* context) {
6      goto code1(context, &context->data[Allocate]->allocate);
7  }
8
9  __code code1(struct Context* context, struct Allocate* allocate) {
10     allocate->size = sizeof(long);
11     allocator(context);
12     goto meta(context, Code2);
13 }
14
15
16 __code code2(struct Context* context, long* count) {
17     *count = 0;
18     goto meta(context, Code3);
19 }
20
21 __code code2_stub(struct Context* context) {
22     goto code2(context, &context->data[Count]->count);
23 }
```

第3章　LLVM, clang

LLVM とはコンパイラ, ツールチェーン技術等を開発するプロジェクトの名称である. 単に LLVM といった場合は LLVM Core を指し, これはコンパイラの基板となるライブラリの集合である. 以降は本論文でも, 単に LLVM といった場合は LLVM Core を指す. LLVM IR や LLVM BitCode と呼ばれる独自の言語を持ち, この言語で書かれたプログラムを実行することのできる仮想機械も持つ. また, LLVM IR を特定のターゲットの機械語に変換することが可能であり, その際に LLVM の持つ最適化機構を利用することができる.

clang は バックエンドに LLVM を利用する C/C++/Objective-C コンパイラである. 具体的には与えられたコードを解析し, LLVM IR に変換する部分までを自身で行い, それをターゲットマシンの機械語に変換する処理と最適化に LLVM を用いる. GCC と比較すると丁寧でわかりやすいエラーメッセージを出力する, コンパイル時間が短いといった特徴を持つ.

3.1　clang の基本構造

3.2　LLVM の基本構造

3.3　Tail call elimination

第4章　LLVM, clang 上での CbC の実装

4.1　コードセグメント

4.2　軽量継続

4.3　環境付き継続

4.4　プロトタイプ宣言の自動化

4.5　フレームポインタ操作最適化

第5章　Gears OS サポート

5.1　meta Code Segment の接続

5.2　stub の接続

第6章　評価・考察

6.1　本研究での改善による成果

6.2　アセンブリコードの評価

6.3　性能評価

6.4　LLVM, clang の利点

第7章　結論

7.1　今後の課題

謝辞

参考文献

[1] 宮國渡, 河野真治, 神里晃, 杉山千秋. Cell 用の fine-grain task manager の実装. 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.

[2] 赤嶺一樹, 河野真治. Datasegment api を用いた分散フレームワークの設計. 日本ソフトウェア科学会第 28 大会論文集, Sep 2011.

[3] 河野真治, 島袋仁. C with continuation と、その playstation への応用. 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), May 2000.