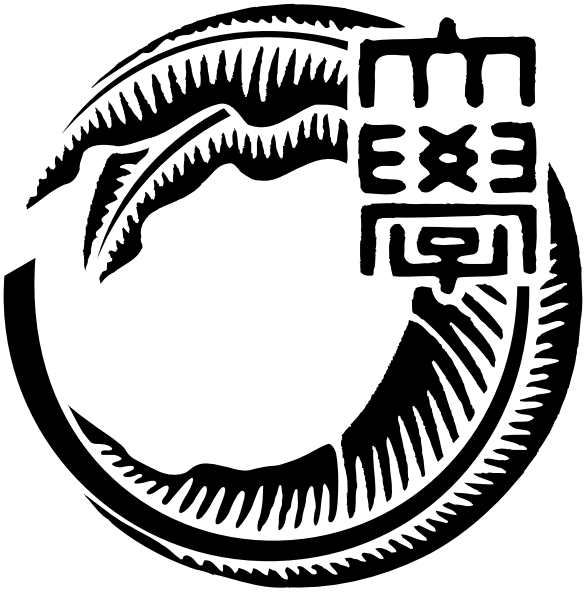
平成28年度 卒業論文

CbC 言語による OS 記述



琉球大学工学部情報工学科

135756F 宮城 光希

指導教員 河野 真治

目 次

図 目 次

リスト目次

第1章　メタ計算の重要性

プログラムを記述する際、通常の処理の他に、メモリ管理、スレッドの待ち合わせやネットワークの管理、エラーハンドリング等、記述しなければならない処理が存在する。これらの計算を Meta Computation と呼ぶ。

Meta Computation を通常の計算から切り離して記述するためには処理を細かく分割する必要がある。しかし、関数やクラスなどの単位は容易に分割できない。

そこで当研究室では Meta Computation を柔軟に記述するためのプログラミング言語の単位として Code Gear、Data Gear という単位を提案している。

Code Gear は関数に比べて細かく分割されているので Meta Computation をより柔軟に記述できる。Code Gear、Data Gear にはそれぞれメタレベルの単位である Meta Code Gear、Meta Data Gear が存在し、これらを用いて Meta Computation を実現する。

Continuation based C (CbC) はこの Code Gear 単位を用いたプログラミング言語として開発している。

CbC は軽量継続による遷移を行うので、継続前の Code Gear に戻ることはなく、状態遷移ベースのプログラミングに適している。

また、当研究室で開発している Gears OS は Code Gear、 Data Gear の単位を用いて開発されており、CbC で記述されている。

本研究では CbC を用いての Gears OS の実装と CbC における ユーザーの関知しない Meta Computation の自動生成を行なう。

第2章　Continuation based C (CbC)

2.1　Continuation based C (CbC)

CbC は 処理を Code Gear とした単位を用いて記述するプログラミング言語である。Code Gear から次の Code Gear へと goto による継続で遷移をし処理を行う。図 2.1 は Code Gear 間の処理の流れを表している。

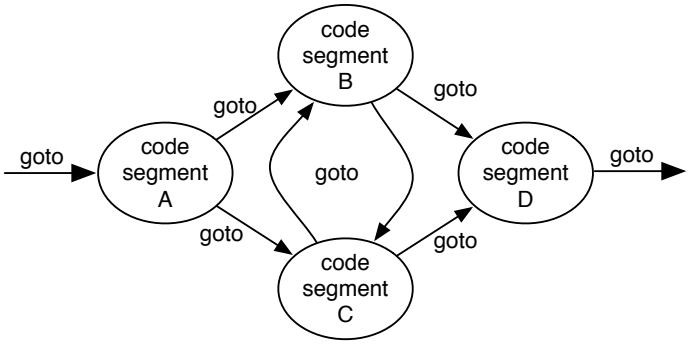


図 2.1: goto による code gear 間の継続

2.2　Code Gear

Code Gear は CbC における最も基本的な処理単位である。リスト 2.1 は最も基本的な CbC のコードの一例で、図 2.2 はそれを図示したものである。CbC では Code Gear は __code という型を持つ関数の構文で定義される。Code Gear は戻り値を持たないので、関数とは異なり return 文は存在しない。goto の後に Code Gear 名と引数を並べて、次の Code Gear の遷移を記述する。この goto の行き先を継続と呼ぶ。Scheme の継続と異なり CbC には呼び出し元の環境がないので、この継続は単なる行き先である。したがってこれを軽量継続と呼ぶこともある。軽量継続により、並列化、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する。

リスト 2.1: code segment の軽量継続

```
1 __code cs0(int a, int b){
2   goto cs1(a+b);
3 }
```

```
4
5  __code cs1(int c){
6    goto cs2(c);
7  }
```

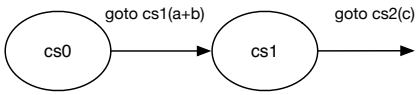


図 2.2: code segment の軽量継続

2.3 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 Gear 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 Gear 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。

環境付き継続を用いる場合、C の関数から Code Gear へ継続する際に __ return、__environment という変数を渡す。__return は __code (*)(return_type, void*) 型の変数で環境付き継続先が元の環境に戻る際に利用する Code Gear を表す。__environment は void** 型の変数で元の関数の環境を表す。リスト 3.1 では関数 funcB から Code Gear 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 Gear 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 2.3 にこの様子を表した。

リスト 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   return 0;
21 }
```

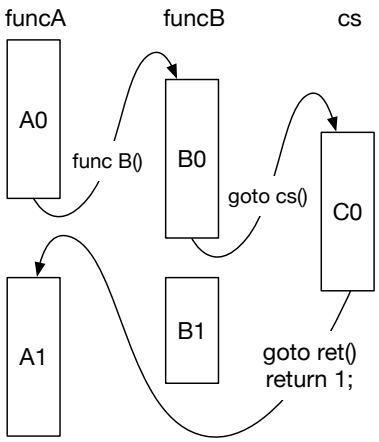


図 2.3: 環境付き継続

このように、環境付き継続を用いることで C、CbC 間の処理の移動が可能になる。

第3章　Gears OS

3.1　Gears OS

Gears OS では並列実行するための Task を、実行する Code Gear 、実行に必要な Input Data Gear 、Output Data Gear の組で表現する。Gears OS は Input/Output Data Gear の依存関係が解決された Task を並列実行する。Data Gear はデータの単位であり、int や文字列などの Primitive Type を持っている。Code Gear は 任意の数の Input Data Gear を参照して処理を行い、Output Data Gear を出力し処理を終える。また、接続された Data Gear 以外には参照を行わない。Gears OS は Input/Output Data Gear の依存関係が解決された Task を並列実行する。処理やデータの構造が Code Gear、Data Gear に閉じているため、これにより実行時間、メモリ使用量などを予測可能なものにすることが可能になる。

Gears OS では Meta Computation を Meta Code Gear、Meta Data Gear で表現する。Meta Code Gear は通常の Code Gear の直後に遷移され、Meta Computation を実行する。

CbC は Code Gear を処理の単位として用いたプログラミング言語であるため、Gears OS の Code Gear を記述するのに適している。

3.2　CbC による Gears OS の構文サポート

Gears OS では Context という 接続可能な Code/Data Gear のリスト、TaskQueue へのポインタ、Persistent Data Tree へのポインタ、Temporal Data Gear のためのメモリ空間等を持っている Meta Data Gear がある。Gears OS は必要な Code/Data Gear に参照したい場合、この Context を通す必要がある。

しかし、Context を直接扱うのはセキュリティ上好ましくない。そこで Context から必要なデータを取り出して Code Gear に接続する Meta Code Gear である stub を定義し、これを介して間接的に必要な Data Gear にアクセスする。

現在 CbC で Gears OS を記述すると通常の Computation に加えて Meta Computation である stub を記述する必要がある。Meta Computation

Context や stub は Meta Computation であるため。

3.3 interface の記述

interface は
interface を記述することで

リスト 3.1: stack の interface

```
1  Stack* createSingleLinkedStack(struct Context* context) {
2      struct Stack* stack = new Stack();
3      struct SingleLinkedStack* singleLinkedStack = new SingleLinkedStack();
4      stack->stack = (union Data*)singleLinkedStack;
5      singleLinkedStack->top = NULL;
6      stack->push = C_pushSingleLinkedStack;
7      stack->pop = C_popSingleLinkedStack;
8      stack->pop2 = C_pop2SingleLinkedStack;
9      stack->get = C_getSingleLinkedStack;
10     stack->get2 = C_get2SingleLinkedStack;
11     stack->isEmpty = C_isEmptySingleLinkedStack;
12     stack->clear = C_clearSingleLinkedStack;
13     return stack;
14 }
```

第4章　LLVM/clang による CbC の実装

4.1　LLVM clang

LLVM とは、モジュラー構成および再利用可能なコンパイラと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。LLVM IR や LLVM BitCode と呼ばれる独自の中間言語を持ち、それを機械語に変換することができる。また、この言語で書かれたプログラムを実行するための仮想機械としても動作する。

clang は LLVM をバックエンドとして利用する C/C++/Ob jective-C のコンパイラである。

4.2　LLVM の基本構造

LLVM は LLVM IR をターゲットのアセンブリ言語に直接的に変換を行うわけではない。LLVM では、最適化や中間表現の変換を何段階か行う。この変換を行う処理は全て pass が行う。多くの pass は最適化のために存在し、そのなかから任意のものを利用することができる。pass には以下のようなものがある。

SelectionDAG Instruction Selection (SelectionDAGISel)
LLVM IR を SelectionDAG (DAG は Directed Acycric Graph の意) に変換し、最適化を行う。その後 Machine Code を生成する。

SSA-based Machine Code Optimizations
SSA-based Machine Code に対する最適化を行う。各最適化はそれぞれ独立した pass になっている。

Register Allocation
仮装レジスタから物理レジスタへの割り当てを行う。ここで PHI 命令が削除され、SSA-based でなくなる.

Prolog/Epilog Code Insertion
Prolog/Epilog Code の挿入を行う、どちらも関数呼び出しに関わるものであり、Prolog は関数を呼び出す際に呼び出す関数のためのスタックフレームを準備する処理、Epilog は呼び出し元の関数に戻る際に行う処理である。

Late Machine Code Optimizations

Machine Code に対してさらに最適化を行う。

Code Emission

Machine Code を MC Layer での表現に変換する。その後さらにターゲットのアセンブリ言語へ変換し、その出力を行う。

これらの処理の流れを図示したものが以下の図 4.1 である。前述した通りこれらの処理は全て pass によって行われる。pass にはいくつかの種類があり、関数単位で処理を行うもの、ファイル単位で処理を行うもの、ループ単位で処理を行うもの等がある。

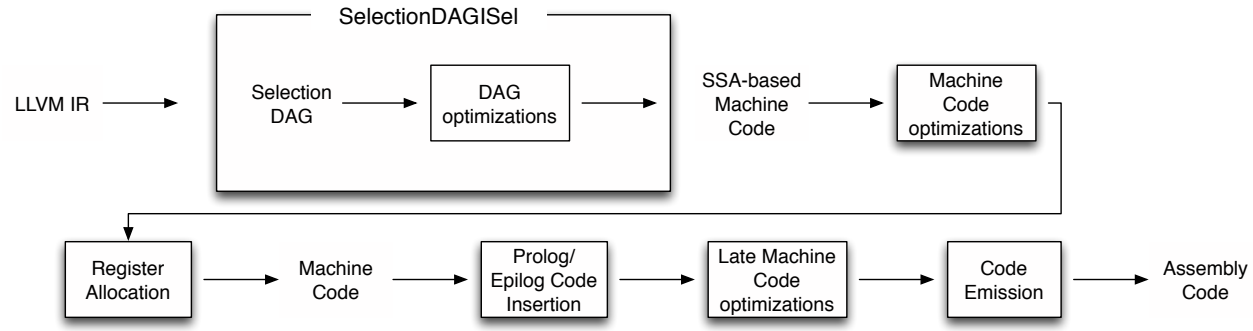


図 4.1: LLVM の 処理過程

4.3 LLVM/clang のデバッグ

LLVM/clang で CbC をコンパイルした際 Code Gear 内 の局所変数でポインタを参照すると tail call されないという不具合があることがわかった。

局所変数でポインタを参照していると clang は生成する LLVM IR にオブジェクトの寿命を示す lifetime.start と lifetime.end を書き出す。

ここでリスト 4.2 のようにオブジェクトの lifetime の終わりを示す lifetime.end が tail call の後に書き出されてしまうことにより、tail call の際に局所変数が解放されておらず lifetime が残っているので tail call が無視されてしまう。

しかし CbC では継続を行った後、継続前の Code Segment に戻ることはないので局所変数の解放は継続前に行っても良い。そこで lifetime.end を tail call の直前で生成を行うことで tail call を出すようにした。

修正後に生成された LLVM IR コード (リスト 4.1) では tail call の直前に生成された。

リスト 4.1: LLVM IR コード 修正前

```
1 if.then: ; preds = %entry
2   %code_stack = getelementptr inbounds %struct.Context, %struct.Context* %context
        , i64 0, i32 8
3   %4 = load %struct.stack*, %struct.stack** %code_stack, align 8, !tbaa !7
```

```
4   %next = getelementptr inbounds %struct.Context, %struct.Context* %context, i64
        0, i32 0
5   %call1 = call i32 (%struct.stack*, i32*, ...) bitcast (i32 (...)* @stack_pop to
         i32 (%struct.stack*, i32*, ...)*)(%struct.stack* %4, i32* %next) #3
6   %5 = load i32, i32* %next, align 8, !tbaa !11
7   tail call fastcc void @meta(%struct.Context* nonnull %context, i32 %5) #3
8   br label %cleanup
9
10 if.end: ; preds = %entry
11  %6 = load %struct.stack*, %struct.stack** %node_stack, align 8, !tbaa !24
12  %call4 = call i32 (%struct.stack*, %struct.Node**, ...) bitcast (i32 (...)*
        @stack_push to i32 (%struct.stack*, %struct.Node**, ...)*)(%struct.stack*
        %6, %struct.Node** nonnull %parent) #3
13  tail call fastcc void @meta(%struct.Context* nonnull %context, i32 18) #3
14  br label %cleanup
15
16 cleanup: ; preds = %if.end, %if.then
17  call void @llvm.lifetime.end(i64 8, i8* %0) #3
18  ret void
```

リスト 4.2: LLVM IR コード 修正後

```
1  if.then: ; preds = %entry
2   %code_stack = getelementptr inbounds %struct.Context, %struct.Context* %context
        , i64 0, i32 8
3   %4 = load %struct.stack*, %struct.stack** %code_stack, align 8, !tbaa !7
4   %next = getelementptr inbounds %struct.Context, %struct.Context* %context, i64
        0, i32 0
5   %call1 = call i32 (%struct.stack*, i32*, ...) bitcast (i32 (...)* @stack_pop to
         i32 (%struct.stack*, i32*, ...)*) (%struct.stack* %4, i32* %next) #3
6   %5 = load i32, i32* %next, align 8, !tbaa !11
7   call void @llvm.lifetime.end(i64 8, i8* %0) #3
8   tail call fastcc void @meta(%struct.Context* nonnull %context, i32 %5) #3
9   br label %cleanup
10
11 if.end: ; preds = %entry
12  %6 = load %struct.stack*, %struct.stack** %node_stack, align 8, !tbaa !24
13  %call4 = call i32 (%struct.stack*, %struct.Node**, ...) bitcast (i32 (...)*
        @stack_push to i32 (%struct.stack*, % struct.Node**, ...)*)(%struct.stack*
        %6, %struct.Node** nonnull %parent) #3
14  call void @llvm.lifetime.end(i64 8, i8* %0) #3
15  tail call fastcc void @meta(%struct.Context* nonnull %context, i32 18) #3
16  br label %cleanup
17
18 cleanup: ; preds = %if.end, %if.then
19  ret void
```

第5章　Context、stub の自動生成

第6章　今後の課題

参考文献

謝辞