Continuation based C コンパイラの LLVM 3.4 上での実装

105711B 氏名 徳森 海斗 指導教員 : 河野 真治

平成 25 年 11 月 3 日

1 背景及び研究目的

現代社会においてプログラムは様々な場所で用いられ, その用途に応じて求められる要素も様々である. 例えば Real-time な制御が重要視される場合には速度が, 組込みシステムなどのメモリが制限される環境では軽量なプログラムが求められる. 軽量で高速であるという条件に最も適しているのはアセンブリ言語であるが, 低レベル過ぎてわかりづらい上に, アーキテクチャへの依存性が高いため汎用的でない. 当研究室ではそういった要求に応える言語として Continuation based C (以下 CbC) を提案している. CbC のコンパイラは 2001 年に micro-c をベースとしたものが開発され, 続いて 2008 年の研究により GCC ベースのものが開発された. これにより GCC の持つ最適化の恩恵を受けることに成功している. しかし、 Mac OS X の最新版である Mavericks が C コンパイラとして GCC ではなく clang を用いるようになったことや, LLVM にも GCC 同様に強力な最適化機能が存在し, さらにエラーメッセージが GCC に比べ丁寧であるという特徴を持つことから, GCC から clang に移り変わる人々も少なくない. そこで本研究では, micro-c , GCC に続く LLVM/clang ベースのコンパイラの開発を行う.

2 Continuation based C (CbC)

CbC のでは処理をコードセグメントという単位で記述し, コード間の移動に goto (軽量継続) を用いる. 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. この goto による処理の遷移を継続と呼ぶ. 図 1 はコードセグメント間の処理の流れを表している.

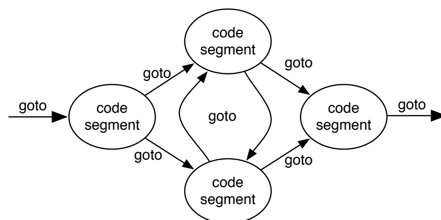

図 1: コードセグメント間の継続 (goto)

2.2 コードセグメント (code segment)

コードセグメントは C の関数と異なり戻り値を持たず, 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. しかし, 戻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. 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言い, 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なる.

3 LLVM/clang 3.4 上での実装

ここでは LLVM/clang 3.4 上でどのように CbC コンパイラを実装するかについて述べる.

3.1 “__code”のパース

予約語は clang/include/clang/Basic/TokenKinds.def で登録する. 登録する予約語が C, あるいは C++のどの規格で使用されるものかもここで記述し, これにより clang のパーサーが “__code”を kw___code として認識するようになる.

3.2 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 ではこれに加え,“goto codeSegment();” 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.

3.3 軽量継続の実装

軽量継続の実装は Tail Call Elimination (末尾除去) の強制によって実現する. これによりコードセグメント間の移動に call ではなく jmp 命令が用いられるようになる. 図 2 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.

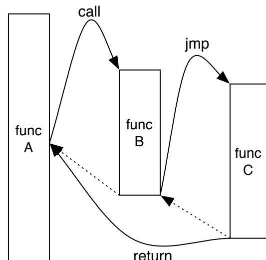

図 2: Tail Call Elimination

3.4 Tail Call Elimination の強制

LLVM で Tail Call Elimination を行う場合, 対象となる関数の isTailCall というフラグを true にした上で, アーキテクチャ によって異なるいくつかの条件を満たす必要がある. 現在は x86/x86-64 と PowerPC がこの最適化を利用できる. これらのアーキテクチャでの条件を以下に示す.

- caller と callee の calling convention が fastcc, cc10, cc11 のいずれかである.
- 関数呼び出しが return の直前に行われており, void 型である.
- 可変長引数リストを使用していない.
- tailcallopt が有効になっている.
- byval parameter を使用していない.
- PIC/GOT を用いる場合,visibility が hidden か protected である必要がある.

3.5 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, その関数から値を返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い,C との互換性を持たせるためにこの機能が必要となる. 環境付き継続は, CbC で定義した以下の二種類特殊変数によって実現される. `__environment` は, 環境を表す情報である. `__return` は, 環境付き継続によって継続した後, 環境付き継続を行った関数の呼び出し元に戻り値を返すためのものであり, 関数の戻り値と `__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 3 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 3: __return, __environment 変数の使用例

環境付き継続の実装案には複数あり, 例えば GCC 版の CbC コンパイラでは内部関数を用いることで環境付き継続を実現している. この他には setjmp , longjmp を用いた実装案, Exception を用いた実装案,LLVM IR を用いた実装案がある.

4 現状及び今後の課題

今後は本稿でも述べたとおり、LLVM/clang 上での CbC コンパイラの実装を行っていく. 実装後は GCC 版と LLVM 版の CbC コンパイラそれぞれでコンパイルしたプログラムの性能比較を行う予定である.

参考文献

[1] 与儀 健人. “組み込み向け言語 Continuation based C の GCC 上の実装”. 琉球大学 情報工学科 学位論文 (修士), 2008

[2] 大城 信康, 河野 真治. “Continuation based C の GCC 4.6 上の実装について”. 第 53 回プログラミング・シンポジウム, 2011

[3] LLVM 3.4 documentation. “http://llvm.org/docs/index.html”

[4] “Clang” CFE Internals Manual - Clang 3.4 documentation. “http://clang.llvm.org/docs/InternalsManual.html”

[5] LLVM API Documentation. “http://llvm.org/docs/doxygen/html/index.html”

[6] Clang API Documentation. “http://clang.llvm.org/doxygen/”