Continuation based C コンパイラの GCC-4.2 による実装

045760E 与儀健人　　指導教員：河野真治

1 はじめに

当研究室では Continuation based C(以下 CbC) という言語を提案している。これまで CbC のコンパイルには Micro-C をベースとした当研究室独自のコンパイラを使用していた。

また、河野氏による以前の論文 [1] にて、Tail call optimization を用いることで GCC 上でも実装可能である事が示されている。

この様な背景から、CbC を GCC でコンパイルしたいという要望がでてきた。本研究ではこの言語を GCC へ移植することを目的とする。それにより GCC の最適化機構による CbC のコード性能の向上、また複数のアーキテクチャへの対応を目指す。

2 Continuation based C について

Continuation based C (以下 CbC) は当研究室が提案するアセンブラよりも上位で C よりも下位な記述言語である [2]。C の仕様からループ制御や関数コールを取り除き、継続 (goto) や code segment を導入している。これによりスタックの操作やループ、関数呼び出しなどのより低レベルでの最適化を、ソースコードレベルで行うことができる。図 1 にこの様子を表す。

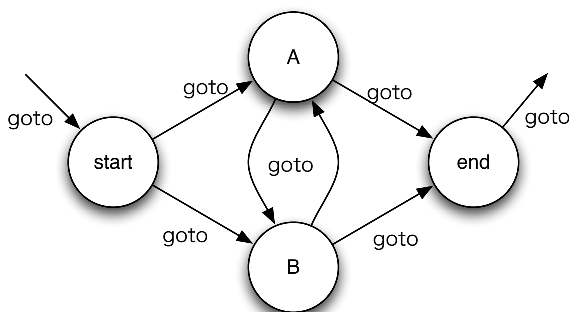

図 1: code segment 間の “継続”

3 The GNU Compiler Collection

3.1 GCC の基本構成

GCC は主に次のような手順でソースコードをコンパイルする。

parsing 一般的なコンパイラと同じく、GCC もまずはパーサによってソースコードを解析し、解析した結果は Generic Tree と呼ばれる tree 構造の構造体に格納される

gimplification この段階では Generic Tree をもとにこれを GIMPLE に変換していく。

GIMPLE optimization 前段階で変換した GIMPLE に対して最適化を行う。

RTL generation ここで、GIMPLE をもとに RTL を生成する。この段階でほぼ言語依存性がなくなる。

RTL optimization 前段階で生成された RTL に対して最適化を行う。

Output assembly 最後に RTL をもとにターゲットマシンのアセンブリに変換する。

これらの様子を図 2 に示す。

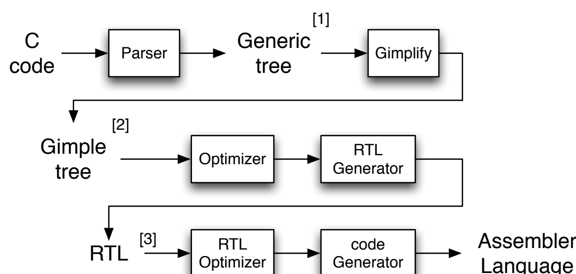

図 2: GCC の pass

3.2 Tail call elimination

GCC の最適化には “Tail call elimination” と呼ばれる、関数呼び出しを最適化するものがある。“Tail call elimination” は通常 call 命令を使用すべき関数呼び出しで、jump 命令に変更するというものである。この最適化の概要を図 3 にしめす。

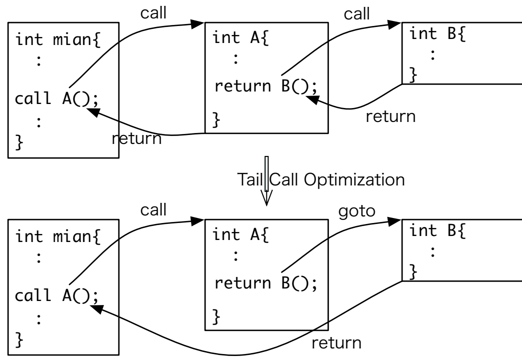

図 3: Tail call elimination の例

4 GCC への実装

河野氏の論文 "継続を基本とした言語 CbC の gcc 上の実装" [1] にて、Tail call optimization をもちいて CbC の goto が実装できる事が示されている。

実装の流れとしては次のようになる。

1. __code トークンの追加 (Tokenizer で読み込めるようにする)
2. code segment のパース及び tree 生成
3. CbC の goto ステートメントのパース及び tree 生成
4. goto ステートメント tree の RTL への変換
5. その他エラーメッセージ処理やコード改良

最も重要なところが RTL 生成である。ここでは tail call 可能なフラグのついた関数コールを RTL に変換することになる。これは通常は `expand_call` という巨大な関数にて生成されている。`expand_call` では tail call が可能かを詳しくチェックして、可能であれば tialcall 用の CALL_INSN RTL. 不可と判定されれば通常の CALL_INSN RTL を生成している。

しかし、goto 先が code segment であれば強制的に tailcall 用の CALL_INSN を生成する必要がある。そこで実装の際には `expand_cbc_goto` という新たな関数を作り、CbC の goto 処理はそこで全て行うようにした。

5 評価

本研究によって、GCC による CbC のコンパイルが可能になった。その評価としては両コンパイラによってコンパイルされたコードの実行速度を測れば良いだろう。評価にはこれまでも Micro-C の評価に用いてきた conv1 という簡単なプログラムを用いた。

測定結果は表 1 に示される。

	conv1 0	conv1 1	conv1 2	conv1 3
Micro-C	5.25	8.97	2.19	2.73
GCC	3.69	4.87	3.08	3.65
GCC (+omit)	2.74	4.20	2.25	2.76
GCC (+fastcall)	2.70	3.44	1.76	2.34
TCC	4.15	122.28	84.91	102.59

表 1: Micro-C, GCC の実行速度比較 (単位 秒)

この評価から本研究における目的の一つ、"CbC コードの高速化" を達成できたことが分かった。

6 今後の課題

本研究において、CbC を使う分にはほぼ問題はなくなったが、まだ対応していない構文や、バグが以下の通り見つかっている。

environment CbC にはもう一つ、environment 付きの継続という構文が存在する。これは関数から code segment に goto した場合に関数の呼び出し元に戻ることを可能にするものだが、今回この実装は間に合わなかった。

code segment ポインタの計算 今の実装では `goto cs->next(a, b);` のように呼び出し先 code segment を計算することができない。

-O2 オプションの強制 CbC は-O2 オプションをつけないとコンパイルできない。なのでファイル名が.cbc の場合はこれを強制させる必要がある。

fastcall code segment では fastcall を強制させることで高速化が期待できる。

この中から特に重要なのが environment と code segment ポインタの計算への対応だと考えている。この二つができればとりあえず CbC の現在の仕様を満たす。

これらに加えて、GCC にはすでに C++や Objective-C のコンパイルが可能である。これを活かし、CbC++, もしくは Objective-CbC といった既存の言語と CbC を組み合わせた言語に付いても考えてみる価値があるだろう。

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 の下位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] GNU Project - Free Software Foundation, GCC internal manual. "http://gcc.gnu.org/onlinedocs/gccint/".