# 組み込み向け言語Continuation based CのGCC上の実装

平成21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

与儀 健人

# 要旨

本研究室では継続を基本としたプログラミング言語 Continuation based C(CbC) を開発している。この言語は C から関数や for ループ制御などを除き、同様の動作は全て継続を用いて実現する事で、C よりも細かい動作を可能にしている。

これまで CbC のコンパイラには micro-c をベースとしたコンパイラを用いてきた。また 2008 年の研究では GCC にて継続制御を実装し、GCC による CbC のコンパイルが可能となった。しかし GCC ベースコンパイラにはいくつかの制限があり、一部の機能が未実装であった。

本研究では GCC ベースのコンパイラに、未実装であった CbC の機能を実装した。

この改善により、GCC ベースコンパイラは CbC の機能を完全にサポートし、さらに以前のバージョンよりも高速化に成功した。加えて GCC ベースにしたことにより、これまで micro-c では対応していなかった多数のアーキテクチャへの対応が可能となった。実測評価においては micro-c ベースのコンパイラと比較し、良好な結果を得ることができた。

# 目 次


**第1章 序論** **1**

1.1 背景と目的 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 論文構成 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**第2章 Continuation based C (CbC)** **3**

2.1 CbC の要求仕様 . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 コードセグメントと継続 . . . . . . . . . . . . . . . . . . . . . . 4
2.2.1 call-return から継続制御へ . . . . . . . . . . . . . . . . . . 4
2.2.2 Scheme における継続制御 . . . . . . . . . . . . . . . . . . . 4
2.2.3 コードセグメント . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.4 軽量継続 (light-weight continuation) . . . . . . . . . . . . . 7
2.3 状態遷移に適した言語 . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 C with Continuation . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4.1 環境付き継続 . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5 CbC の用途・先行研究 . . . . . . . . . . . . . . . . . . . . . . . 11
2.5.1 プログラムの検証 . . . . . . . . . . . . . . . . . . . . . . . 11
2.5.2 ゲームプログラミングにおけるデモンストレーション . . . . . . . . 11
2.6 CbC コンパイラの現状と本研究における目標 . . . . . . . . . . . . 12
2.6.1 micro-c と GCC . . . . . . . . . . . . . . . . . . . . . . . . 12
2.6.2 本研究における目標 . . . . . . . . . . . . . . . . . . . . . . 12

**第3章 GNU コンパイラ コレクション** **14**

3.1 コンパイル、アセンブル、リンク . . . . . . . . . . . . . . . . . 14
3.2 cc1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2.1 フロントエンドと Generic, GIMPLE . . . . . . . . . . . . . . 14
3.2.2 ミドルエンドと RTL . . . . . . . . . . . . . . . . . . . . . . 15
3.2.3 バックエンドと Machine Description . . . . . . . . . . . . . 17
3.3 GCC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**第4章 GCC における実装・改善** **20**

4.1 過去の研究における実装部分 . . . . . . . . . . . . . . . . . . . . 20
4.1.1 コードセグメントの実装 . . . . . . . . . . . . . . . . . . . . 20
4.1.2 軽量継続の実装 . . . . . . . . . . . . . . . . . . . . . . . . 21

4.2 本研究における実装 . . . . . . . . . . 24
4.2.1 並列代入 . . . . . . . . . . 24
4.2.2 環境付き継続 . . . . . . . . . . 27
4.2.3 PowerPC における間接継続 . . . . . . . . . . 30
4.2.4 x86 における引数渡し . . . . . . . . . . 32
4.2.5 プロトタイプ自動生成 . . . . . . . . . . 33

**第 5 章 評価・考察** **35**
5.1 本研究での改善による成果 . . . . . . . . . . 35
5.2 GCC を使うことの利点・欠点 . . . . . . . . . . 36
5.3 性能評価 . . . . . . . . . . 37
5.3.1 評価項目、比較対象 . . . . . . . . . . 37
5.3.2 評価手法と環境 . . . . . . . . . . 38
5.3.3 評価結果 . . . . . . . . . . 38
5.3.4 評価結果考察 . . . . . . . . . . 38
5.4 メンテナンス性の向上に関する取り組み . . . . . . . . . . 42
5.4.1 二つのリポジトリ . . . . . . . . . . 43
5.4.2 このリポジトリ管理方法の評価 . . . . . . . . . . 44

**第 6 章 結論** **45**
6.1 まとめ . . . . . . . . . . 45
6.2 今後の課題 . . . . . . . . . . 45

**謝辞** **47**

**参考文献** **48**

**発表文献** **50**

**付 録 A 付録** **51**
A-1 測定環境 . . . . . . . . . . 51
A-2 `__return` 擬似変数の実装 . . . . . . . . . . 52
A-3 プロトタイプ生成スクリプト . . . . . . . . . . 54
A-4 quicksort 例題 . . . . . . . . . . 56

# 図 目 次

2.1 call-return 制御 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 継続制御 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 C with Continuation とそのサブセット . . . . . . . . . . . . . . . . . . . 9
2.4 __return の例 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1 cc1 でのデータフロー (Generic, GIMPLE, RTL) . . . . . . . . . . . . . . . 15
3.2 コード 3.1 の構文木の例 . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3 md からソースコードを生成、さらに cc1 をコンパイルする様子 . . . . . . 19

4.1 末尾呼び出し最適化が可能な関数 funcY の例 . . . . . . . . . . . . . . . . . 21
4.2 継続制御と関数呼び出しでのスタックの違い . . . . . . . . . . . . . . . . 23

5.1 CbC コンパイラ開発でのリポジトリ管理（左が本家のリリースタイムライン、中央が GCC-copy、右が CbC の開発用リポジトリのタイムライン） . 43

# 表 目 次

5.1 アーキテクチャ毎の GCC と micro-c の速度比較（単位: 秒） . . . . . . . . . 39
5.2 実行ファイルのファイルサイズ比較 stripped（単位: bytes） . . . . . . . 39
5.3 GCC-4.2.3 ベースと GCC-4.4.2 ベースの速度比較（単位: 秒） . . . . . . 39

# プログラムコード目次


2.1 Schemeでの継続制御の例 . . . 6
2.2 CbCプログラムの例（階乗計算） . . . 6
2.3 __returnの例 . . . 9
2.4 setjmp/longjmpの例 . . . 10
2.5 間接継続の例（2つめのgoto文） . . . 12
3.1 C言語の解析例（解析結果は図3.2） . . . 16
3.2 レジスタに20を乗算する命令のRTL表現例 . . . 17
3.3 ARMでのMachine Descriptionの例（コード3.2をアセンブラに変換） . . . 18
4.1 構文木生成(gcc/c-typeck.c) . . . 20
4.2 goto文の例 . . . 22
4.3 構文木での解釈 . . . 22
4.4 並列代入の必要な軽量継続の例 . . . 24
4.5 引数の退避 . . . 25
4.6 上記の処理を行う関数 . . . 26
4.7 __returnの例 . . . 28
4.8 コード2.3のfuncBに追加される処理 . . . 28
4.9 4.8のfuncBで出力されるアセンブラ(x86) . . . 29
4.10 PowerPCにおける間接継続のRTL . . . 30
4.11 4.10の変換規則 . . . 31
4.12 fastcallな関数fastfuncを宣言する例 . . . 32
4.13 c_parser_declspecsにおけるキーワード“code”の処理 . . . 33
4.14 プロトタイプ自動生成スクリプトの出力例 . . . 34
5.1 quicksortプログラムで使われているコードセグメント . . . 40
5.2 divider_eのGCCによる出力(ppc) . . . 41
5.3 divider_eのmicro-cによる出力(ppc) . . . 41
A.1 c_parser_postfix_expressionでの処理 . . . 52
A.2 c_parser_postfix_expressionでの処理 . . . 52
A.3 プロトタイプ生成スクリプト . . . 54
A.4 quicksort_cbc.cbc . . . 56
A.5 quicksort_test.cbc . . . 58

# 第1章　序論

## 1.1　背景と目的

企業システムの多様化、IT 導入の加速により、ソフトウェアは大規模化・複雑化する傾向にある。また家電製品のデジタル化も進み、組み込みシステムの需要も増大している。
それにともないハードウェアは驚異的な進歩を遂げ、近年は CPU のマルチコア化が進み、また新たな段階を築こうとしている。
ハードウェアの進歩に対し、ソフトウェアの開発に用いられる記述言語は、オブジェクト指向プログラミングの発明・導入やデザインパターンに見られる技術の集約などが行われ、注目されてきた。オブジェクト指向を主とした Java はその有用性が認められ多くのシステム開発に取り入られている。しかしその反面、Java ではガベージコレクタや実行時コンパイルにより、余分な処理が必要となる。そのため軽量かつ高速な応答が要求される Real-time 処理や組込み用途には適さない。
また PlayStation3 には Cell Broadband Engine という特殊な CPU が採用され注目されている。しかしこの様な複雑なアーキテクチャを持つマシンではプログラミング自体も複雑になる。C で記述されたプログラムからアーキテクチャに直接関わる命令（DMA やシグナル）を使用するのでは、高級言語の設計思想と矛盾するともみられる。
大規模システムにおけるバグの存在も深刻な問題である。テストファーストな開発スタイルなどで工学的なアプローチからバグの抑制が試みられているが、完全な排除は難しい。数学的なアプローチから無矛盾を証明する技術の研究も進んでいるが、現在のスタックベースのプログラミングは状態数が膨大になり、実用化された例は少ない。しかしマルチコアの台頭により並列プログラミングの必要性も高まっており、今後はより検証の必要性が増すと考えられる。
ハードウェアの進化や数学的検証にソフトウェアが対応するためには、これまでとは違う新たな視点を持ったプログラミング言語が望ましい。しかし既存のソフトウェアやシステムは膨大な数に上り、これらを新しい言語に書き換えるのは無理がある。新しい言語は古い言語との互換性が必須である。
我々はこれらの問題に取り組むため、Continuation based C（以下 CbC）という言語を提案している。Continuation とはプログラムの次の実行処理を表現する制御構造で、継続とも呼ばれている。CbC では C からサブルーチンやループ制御を除き、代わりに継続をベースとした実行制御を行う。この特徴から、CbC は C の下位言語と考えることができ、ハードウェアの記述や記述したプログラムの検証などを目的として設計されている。
これまで CbC のコンパイルには、micro-c をベースとしたコンパイラが用いられてき

た。加えて 2008 年の研究において GCC をベースとした CbC コンパイラが開発され、継続処理の実装が行われた。

本論文では GCC ベースのコンパイラにおいて残る CbC の機能の実装を行い、実用的な CbC プログラムの動作を目指す。

また、実装した GCC ベースコンパイラの評価として micro-c ベースコンパイラとの速度比較を行い、GCC の開発リリースに合わせるためのメンテナンス手法についても考察する。

## 1.2　論文構成

次章以降、本論文では本研究での成果を報告する。

2 章ではまず CbC についてその言語仕様から説明し、また現時点での CbC コンパイラの問題点を挙げる。

3 章では本研究での移植対象である GNU コンパイラコレクションの構造について、簡単に説明する。

4 章は実際の実装方法、改善の手法を説明し、これらの結果の評価を 5 章で行う。

最後に、本研究での成果と今後の課題を 6 章で述べ、本論文のまとめとする。

# 第2章　Continuation based C (CbC)

Continuation based C（以下 CbC）は当研究室の提案する、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。我々は様々な視点からこの CbC を用いた研究を行っている。本章ではその CbC の仕様と現在の状況について説明し、また CbC を用いた研究例についても紹介する。

## 2.1　CbC の要求仕様

90 年代以降、ハードウェアの進歩がプログラミング言語よりも早く進みつつあり、70 年代、80 年代に設計された言語は矛盾を抱えて来ている。

オブジェクト指向技術とそれに基づいた Java などの言語が注目されテイルが、Java ではガベージコレクタや実行時コンパイルにより、余分な処理が必要となる。そのため軽量かつ高速な応答が要求される Real-time 処理や組込み用途には適さない。この用途にはハードウェアに近い記述が要求される。

ハードウェアに一番近い言語はアセンブラであるがマクロアセンブラなどの記述はあまりにも低レベルであり、依存性が強く汎用的ではない。さらに使用可能なゲート数が増えるにつれ、RISC 的な対称性の高い少数の命令よりも、複雑な SIMD 命令やソフトウェアパイプライン命令を持つ CPU が増えてきている。そのために既存の言語に対するコンパイラをその都度設計し直すことが必要になってきている。

VHDL, Verilog などのハードウェア記述言語は有限状態遷移の中に閉じており、オブジェクト指向などの抽象化とはまったく別なものとなってしまっている。

このようにハードウェア記述言語、アセンブラ、プログラミング言語の 3 つは全く異なる方向を向いている。コンパイラの自動生成などが重要な研究テーマとなると考えられるが、この 3 つが全く独立したものであれば困難なものになると考えられる。

そこで CbC はこの 3 つを埋めるべく以下のような要求仕様に従って設計された。

- ハードウェアとスタックマシンの中間言語

  インタプリタ記述やコンパイラターゲットとして優れる。アーキテクチャ依存性が少ない。また、アーキテクチャ依存性をモデル化できる。

- C 言語よりも下位の言語

  アセンブラよりも汎用性と記述性に優れ C と互換である。C を CbC にコンパイルでき、ハンドコンパイルの結果を同値なコードに変換できる。

- 明確な実行モデル

  C++や Prolog のような複雑な実行モデルは好ましくなく、ハードウェアに実行順序の変更を許す範囲を広くする。

- 状態遷移を直接記述できる

  Yacc のような表駆動や C のような巨大な switch 文ではなく直接に状態遷移ができ実行できる。

- Thread を実行モデルに内蔵できる

  状態遷移記述と CbC 上のスケジューラ実装によりスレッドを実現可能にする。

- クリティカルパスの最適化

  全体を散漫に最適化するのではなく、実行ルーチンから重要な箇所を抜き出し、アセンブラに近い最適化をソースコードレベルで実現する。

これらの仕様はハードウェア記述とソフトウェア記述の両方を同時に行いつつ、C よりも精密な実行記述を可能にするためのものである。また、CbC はプログラム変換やコンパイラターゲットとしての使用を意識している。状態遷移記述のみでは制御機構は静的なものになってしまう。CbC では状態遷移記述に適した言語を作ることを考え、スタックマシンを避けて Continuation（継続）が導入されている。

## 2.2 コードセグメントと継続

### 2.2.1 call-return から継続制御へ

C などの一般的な手続き型言語では、呼び出した手続きの処理のあと、呼出し元の環境に復帰する。そのためプログラム全体においてスタックが用意され、呼出し元はスタックに復帰先アドレス及び環境を保持しておく事で呼出し先からの復帰を可能とする。これは call-return 制御と呼ばれるものである（図 2.1）。しかし復帰先が決まっていて環境を受け継ぐことができれば、この call-return 制御は図 2.2 の様に手続き呼び出しの前後で分割する事ができ、スタック操作を伴わないシーケンシャルな呼び出しに変換する事ができる。これは継続制御構造と呼ばれている。scheme の call-with-continuation の実装や、Java,C++の例外処理、C の setjmp()/longjmp() による大域脱出もこの継続制御の一種である。

### 2.2.2 Scheme における継続制御

継続とは一般的には「現在の処理を続行するための情報」と解釈されている。継続制御はその情報をプログラム記述で操作するための構文である。例として Scheme での継続の使用をコード 2.1 に挙げる。

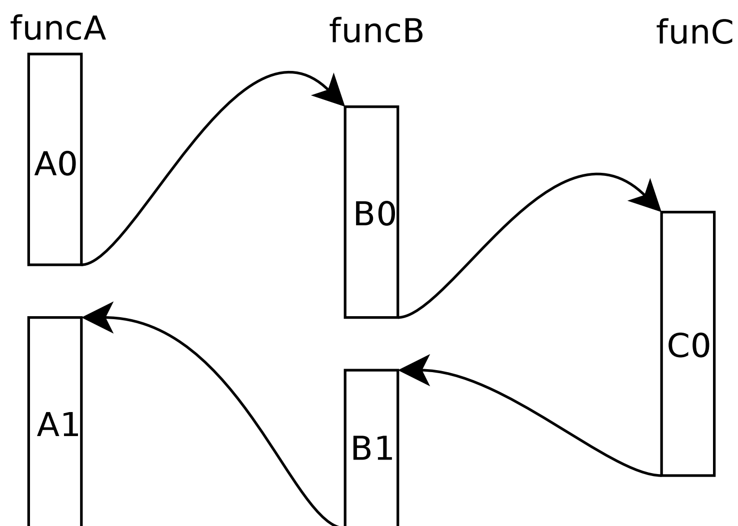


図 2.1: call-return 制御

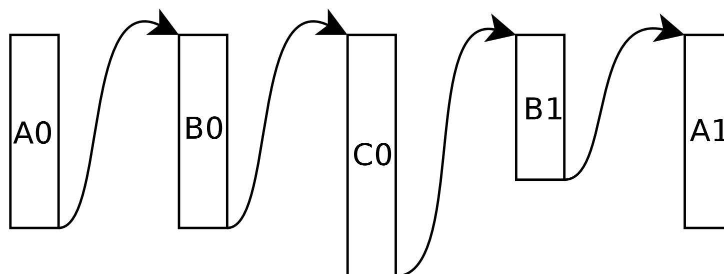


図 2.2: 継続制御

コード 2.1: Scheme での継続制御の例

```
gosh> (define (cont-test i)
  (print "before")
  (call/cc (lambda (k) (set! cont k)))
  (print "after")
  (set! i (+ 1 i))
  i)
cont-test
gosh> (cont-test 10)
before
after
11
gosh> (cont)
after
12
gosh> (cont)
after
13
gosh> (cont-test 2222)
before
after
2223
gosh> (cont)
after
2224
```

この例では関数 `cont-test` 内にて `call/cc` を呼ぶことで、現在の計算処理の “継続” を関数として変数 `cont` に保持している。

その後、`(cont)` という命令でその関数を実行すると、cont が代入された位置に処理が復帰する。そのため、直前の “before” は出力されずに “after” が出力されていることが分かる。`cont` では関数の継続処理だけでなく、引数などの環境も一緒に保持しているので、この `cont` は呼ばれる度に `i` カウントアップし、その値を返すことになる。

CbC はこの継続制御を基本として設計されており、その実現のためにコードセグメントと軽量継続という概念を用いている。以下ではその二つについて説明する。

### 2.2.3 コードセグメント

CbC は図 2.2 の様に分割された手続きのそれぞれを一つの処理単位として用いる。これを “コードセグメント (code-segment)” と呼ぶ。

コードセグメントはキーワード “code” を用いて C の関数の様に定義される。引数部分はインタフェイスと呼ばれ、継続前のコードセグメントからの出力にあたる。例として、引数で与えられた数 x の階乗を求めるプログラムをコード 2.2 に示した。

コード 2.2: CbC プログラムの例（階乗計算）

```
code print_factorial(int prod) {
  printf("factorial = %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);
}

int main(int argc, char **argv) {
  int i;
  i = atoi(argv[1]);

  goto factorial(i);
}
```

### 2.2.4　軽量継続 (light-weight continuation)

コードセグメントは C における関数とは違い、呼出し元への復帰は存在しない。そのためコードセグメントの処理の末尾で別のコードセグメントへ継続することになる。CbC ではこの継続制御を “軽量継続 (light-weight continuation)” と呼ぶ。

軽量継続はキーワード “goto” のあとにコードセグメント名とそのコードセグメントのインタフェイスに渡す引数列を並べて記述する。(同じく軽量継続の例がコード 2.2 にみられる。)

この例の様に、プログラムは for や while などのループ制御構造を含んでいない。代わりに、コードセグメント `factorial0` の様に自分自身への軽量継続を用いることで繰り返し処理を実現している。C では再帰関数を使うことで同じことを行えるが、そこにはスタックの拡張という処理が入る。しかし CbC ではスタックの拡張は行われず、元の環境に戻ることはない。

## 2.3　状態遷移に適した言語

Continuation based C は値を返すプログラムよりも、状態遷移記述に適している。

従来の言語での状態遷移記述は

- 表を使った状態遷移インタプリタ
- 巨大な switch 文

などが用いられてきた。しかしこれらは記述性が悪く、効率も良くない。

表を使った状態遷移インタプリタはコンパイラ言語とは考えられない。また、それをハードウェア記述に落とすことは難しい。

巨大な switch 文は、コンパイルが複雑になり、適切な最適化を行うことが難しい。また、人間が読む場合にも読みやすいとは言えない。

CbC は元々状態遷移を直接記述することを目的として設計されており、手続きの様に環境の保持を伴わないため、その時々に実行中のコードセグメントとその引数を直接プログラムの状態とみなす事ができる。

特にゲームや GUI を用いたプログラムなどでは状態遷移記述が多用されており、そのようなプログラムでは CbC を状態記述言語として使うことにより、直接実行による実行の高速化と既存の言語と状態遷移記述の整合性の向上をはかることができる。

## 2.4　C with Continuation

数学的検証や組み込み用途を目的として提案された CbC であるが、既存のソフトウェアやシステムは膨大な数にのぼり、これらを CbC に置き換えるのは無理がある。そのため、少なくともソースコードのレベルで C との互換性を持つことが望ましい。Continuation based C の名のとおり、CbC から C の関数の呼び出しは問題なく行える。しかし CbC を C と相互に利用するためには、C の関数から継続を行った場合に元の環境に戻るための、特殊な継続を導入する必要がある。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 の両方の機能をもつ言語となる。また、 C、CbC は CwC のサブセットと考えられるので（図 2.3)、CwC のコンパイラを CbC に使用する事ができる。これまでに実装されてきた CbC の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。

### 2.4.1　環境付き継続

環境付き継続を用いる場合、C の関数からコードセグメントへ継続する際に `__return` という変数で表される特殊なコードセグメントポインタを渡す。コード 2.3 では関数 `funcB` からコードセグメント `cs` に継続する際に `__return` を渡している。継続先のコードセグメントでは渡されたコードセグメントポインタへ継続する事で元の C の環境に復帰することが可能となる。ただし復帰先は `__return` を参照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` からは `funcB` が正常に終了したように見える。図 2.4 にこの様子を表した。

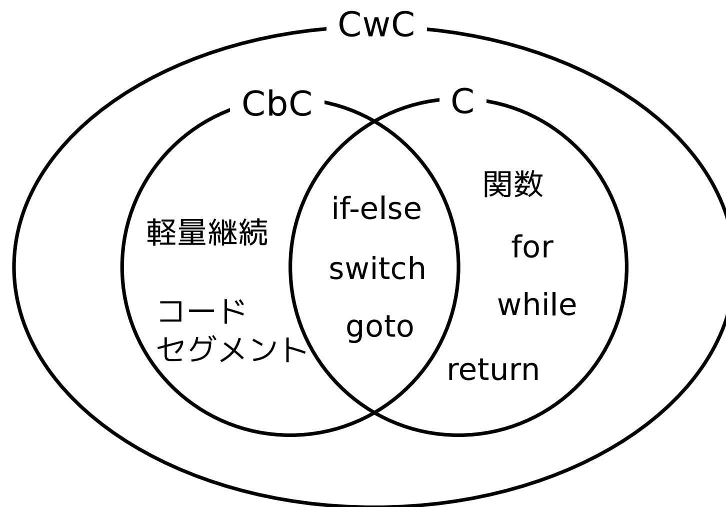


図 2.3: C with Continuation とそのサブセット

コード 2.3: __return の例

```
code cs(code (*ret)(int))
{
    goto ret(2);
}

int funcB()
{
    code (*ret)(int);
    ret = __return;
    /* do something.  */
    goto cs(ret);

    /* never reached.  */
    return -1;
}

void funcA()
{
    int t;

    t = funcB();

    printf("t=%d\n", t);
    /* t should not be -1 but 2.  */
}
```

この様な形にすることで code segment 側では関数から呼ばれたか、コードセグメントからの継続かを考慮する必要がない。また、funcA からもその内部でコードセグメントが使われていることを隠蔽できる。

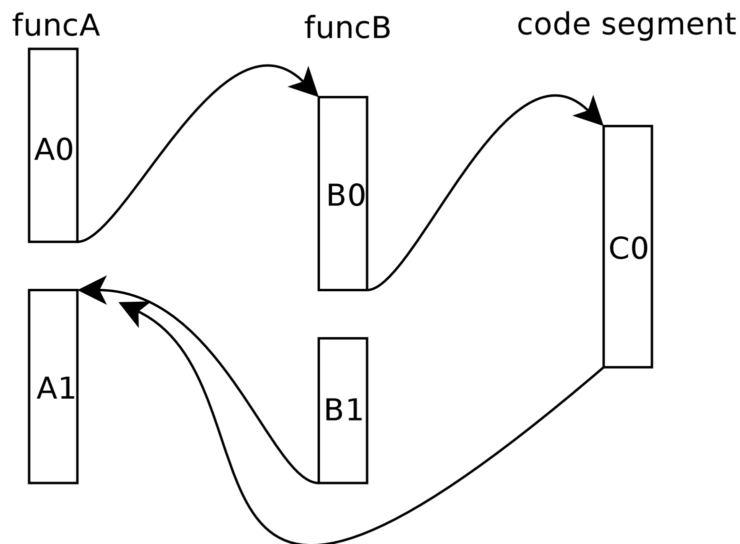


図 2.4: __return の例

環境付き継続は実際には C における setjmp()/longjmp() とほぼ同じ処理である。この二つの関数は C で継続を実現するために用いられる。

例としてコード 2.4 を挙げる。このコードでは setfunc 内で setjmp を使用している。setjmp は通常は 0 を返すため、if 文の内部は実行されないが、その後 longjmp が実行されると、関連する setjmp が呼び出された環境に "継続" し、非零を返すため if 文の中が実行されることになる。この時、longjmp の呼出側（この例では jmpfunc）の環境は失われる。

環境付き継続もこの動作によく似ており、if 文内で return のみを記述することに相当する。

コード 2.4: setjmp/longjmp の例

```
int
setfunc()
{
    int a;
    jmp_buf env;

    if (a=setjmp(env)) {
        printf("it's continued! with value %d", a);
        return a;
    }

    jmpfunc(env);
```

```
    return 0;
}

void
jmpfunc(jmp_buf env)
{
    if (flag) {
        longjmp(env, 2);
    }
    return;
}
```

## 2.5 CbCの用途・先行研究

CbCによるプログラム記述の例として本研究室における研究例を紹介する。

### 2.5.1 プログラムの検証

計算機科学の進歩により、ソフトウェアは大規模かつ複雑なものになっている。しかしそれに応じて、設計段階において誤りが生じる可能性も高くなってきており、設計されたシステムに誤りがないことを保証するための論理設計や検証手法及びデバッグ手法の確立が重要な課題となっている。

どんなプログラムでも状態と状態遷移が存在し、その全てを網羅的に探索することでデッドロックなどの望ましくない状態を検出することができる。探索にはさまざまな手法が考えられるが、プログラムを直接状態遷移として記述できればこの探索に有利となる。

本研究室の下地らはこの特徴を持つCbCを用いて線形時相論理による検証を提案し、その有用性を示した。[8], [9]

### 2.5.2 ゲームプログラミングにおけるデモンストレーション

我々は家庭用ゲーム機で動作するゲームプログラムのオープンな開発フレームワークに関する研究も行ってきた。家庭用ゲーム機の多くは特殊なアーキテクチャをもち、そのためゲームプログラムには汎用性や冗長性が極めて小さく、移植が困難という問題がある。

その問題の解決に、ゲームプログラム全体を小規模なプログラムの集合である“デモンストレーション”に分割することで移植性を向上する手法を本研究室の金城らが提案した。[6],[10]

このデモンストレーション手法はプログラムを細かく分割するため、ゲーム機や組み込みなどの資源が制約された環境ではサブルーチンによるスタック操作がネックとなる。そのためこの手法ではプログラム分割の実現にCbCを用いており、CからCbCへの機械的な変換方法について述べている。

## 2.6　CbC コンパイラの現状と本研究における目標

### 2.6.1　micro-c と GCC

CbC のコンパイラには二つの実装が用意されている。一つは 2000 年に当研究室の河野らにより開発された、micro-c という C のコンパイラをベースとしたものである。こちらは現在安定して動作しており、アーキテクチャは PowerPC, x86, MIPS, ARM などに対応している。もう一つは 2008 年に開発された、GCC をベースとしたコンパイラである。[13]

GCC は元より多数のアーキテクチャに対応しており、高機能な最適化も備えている。これらを CbC でも活用したいという要望からコンパイラ環境の移植が行われた。

### 2.6.2　本研究における目標

この時の実装でコードセグメント、継続制御構造などは実装され、一通りの CbC プログラムのコンパイルが可能となった。

本研究ではこの GCC ベースのコンパイラをより実用的な CbC コンパイラとすべく以下の項目を目標とする。

- 環境付き継続

  C との互換性のための制御構造である環境付き継続を実装する。

- 並列代入

  これまで GCC ベースのコンパイラでは、実装方法の影響から継続制御に一部制限が存在した。これは実行中のコードセグメントの引数と継続制御に渡す引数の順序が入れ替わる場合等に継続が行えないという制限である。

  並列代入を行うことで引数順序の影響はなくなり、この制限を排除できる。

- PowerPC における間接継続 (indirect goto)

  C での関数ポインタを用いた間接呼び出し (indirect call) の様に、CbC で用いる継続制御においても、コードセグメントポインタを用いたメモリ参照による間接的な継続が可能である。これを “間接継続” と呼んでいる。コード 2.5 の codepointer への継続が間接継続に当たる。

コード 2.5: 間接継続の例（2 つめの goto 文）

```
code somesegment( . . . ) {
    code (*codepointer)();
    /*  do something  */
    if ( )
        goto nextsegment();
    else
```

```
        goto (*codepointer)();
}
```

しかし PowerPC アーキテクチャでは最適化の問題からこの間接継続がこれまで制限されていた。

間接継続は CbC でのプログラミングには必須であり、また本研究室の主要プロジェクトである Cerium は PS3（PowerPC をもつ）をメインターゲットとしているため、この対応は必須のものである。

- プロトタイプ宣言の自動化

  C のプロトタイプ宣言はコンパイル時のエラー検出に役立っているが、CbC では返り値が存在しないなど、あまり重要な意味をなさない。また、micro-c ではこれを極力排除するよう設計されているため、既存の CbC プログラムとのソースコードレベルでの互換性が薄れてしまう。

  プロトタイプを自動生成することにより、この互換性を向上させる。

- x86 での継続制御の最適化

  x86 では、C の関数呼び出し全ての引数をメモリに格納する。コードセグメントは関数をベースに作られているため、この ABI に引きずられ実効速度に影響をもたらしている。引数の一部をレジスタに格納することで、x86 における継続処理の高速化を行う。

- メンテナンス性の向上

  GCC のソースコードは 200 万行にものぼる。CbC コンパイラで修正するソースコードはそのごく一部であるが、GCC のアップデートによる修正は CbC 用のソースコードにも大きな影響をもたらす。GCC の最新リリースに追従するためには、アップデートも考慮し、洗練されたメンテナンス方法が必要になる。

4 章ではこれらの項目の実装を行う。

# 第3章　GNU コンパイラ コレクション

GNU コンパイラ コレクション（以下 GCC）はフリーソフトウェア財団によって管理されているオープンソースのコンパイラ群である。C, C++, Java, FORTRAN などの様々な言語に対応しており、UNIX 系 OS の標準的なコンパイラとして用いられている。

本研究における CbC コンパイラの実装対象もこの GCC となる。本章では実装に当たる予備知識として GCC のプログラム構成について簡単に説明する。

## 3.1　コンパイル、アセンブル、リンク

GCC はコンパイルだけでなく、出力したアセンブラのアセンブル、リンクまで行い、最終的に実行ファイルを出力する。このコンパイル、アセンブル、リンクはそれぞれ cc1, as, collect2 というプログラムが行っており、GCC は実際にはそれらを統括しているだけである。

言語に関する処理は cc1 だけである。以降はこのプログラム cc1 がどのようにソースコードをアセンブラに変換するかを説明する。

## 3.2　cc1

GCC ではプログラムソースコードをアセンブラに変換する過程で Generic, GIMPLE, RTL という 3 つの内部表現を用いている。これらの内部表現とソースコード、アセンブラ間の変換はフロントエンド、ミドルエンド、バックエンドが担当している。図 3.1 にその様子を示す。

### 3.2.1　フロントエンドと Generic, GIMPLE

フロントエンドはコンパイラ中で直接ソースコードを解析するルーチンのことである。ソースコードの解析はコンパイルする言語毎に異なるため、フロントエンドの実装も C, C++, Java などで様々である。言語によって異なるソースコードを解析し、その結果を Generic という構文木（プログラムの構造を表すデータ構造）に変換するのがフロントエンドの役割となる。

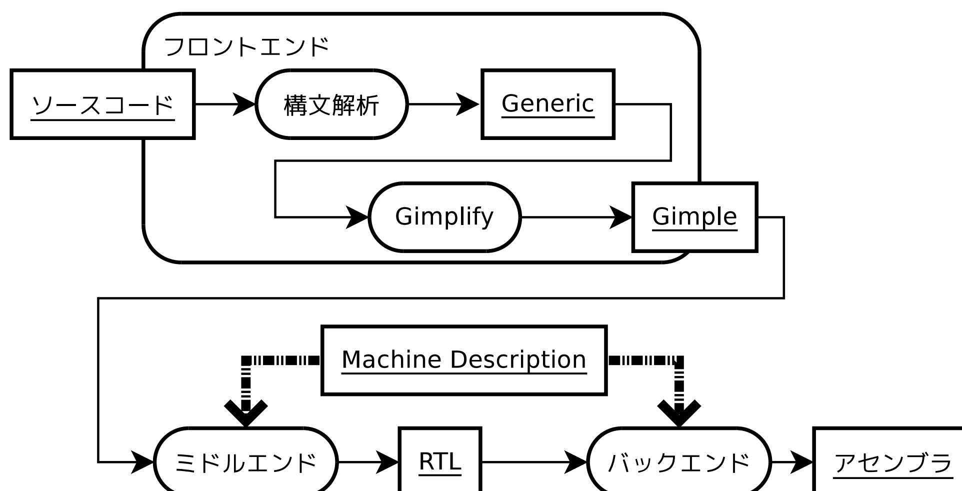


図 3.1: cc1 でのデータフロー (Generic, GIMPLE, RTL)

構文木 Generic は関数の宣言や繰返し制御構造、条件分岐、リターン文など、プログラムの構造を全てツリー構造で表現することができる。この構文木は言語に依存しないため、言語設計者は通常はミドルエンド以降については考慮する必要はない。

構文木がプログラムをデータ構造として表す様子を図 3.2 に示した。この図はコード 3.1 の関数 `funcT` を解析した結果を構文木で表現している。

図のように、一般的なコンパイラでは`#include` などのディレクティブを除くソースコードの全てを構文木で表現する。これによりプログラムの文脈をコンパイラが理解し、それぞれのブロック毎にアセンブラへの変換が可能になる。

**Gimplify と GIMPLE(SSA)**

フロントエンドでは Generic を生成した後、ミドルエンドにデータを渡す前に Generic を GIMPLE と呼ばれるデータ構造に変換する。この処理が Gimplify である。GIMPLE はデータ構造としては Generic と同じであるが、一つの枝に 4 つ以上の子がついてはいけないなどの制限が付加されている。

この制限されたデータ構造は一般的には静的単一代入 (Static Single Assignment) と呼ばれており、様々な最適化を簡略化する事を目的として導入されている。

### 3.2.2 ミドルエンドと RTL

GCC はフロントエンドにて構文木 GIMPLE の生成後、この GIMPLE を解析しながら RTL と呼ばれる中間コードを生成する。RTL はアセンブラとほぼ同等の命令列を表現可能であり、どのアーキテクチャでも同じように扱われる。また、GIMPLE にも言語の依存はないため、ミドルエンドは言語にもアーキテクチャにも依存しない、全ての GCC コンパイラに共通のルーチンとなっている。

コード 3.1: C 言語の解析例（解析結果は図 3.2）

```
bool
funcT(char a, char *b, int n) {
    int i;

    for (i=0; i<n; i++) {
        if ( a[i]==b ) {
            return f(b,i);
        }
    }
    return false;
}
```

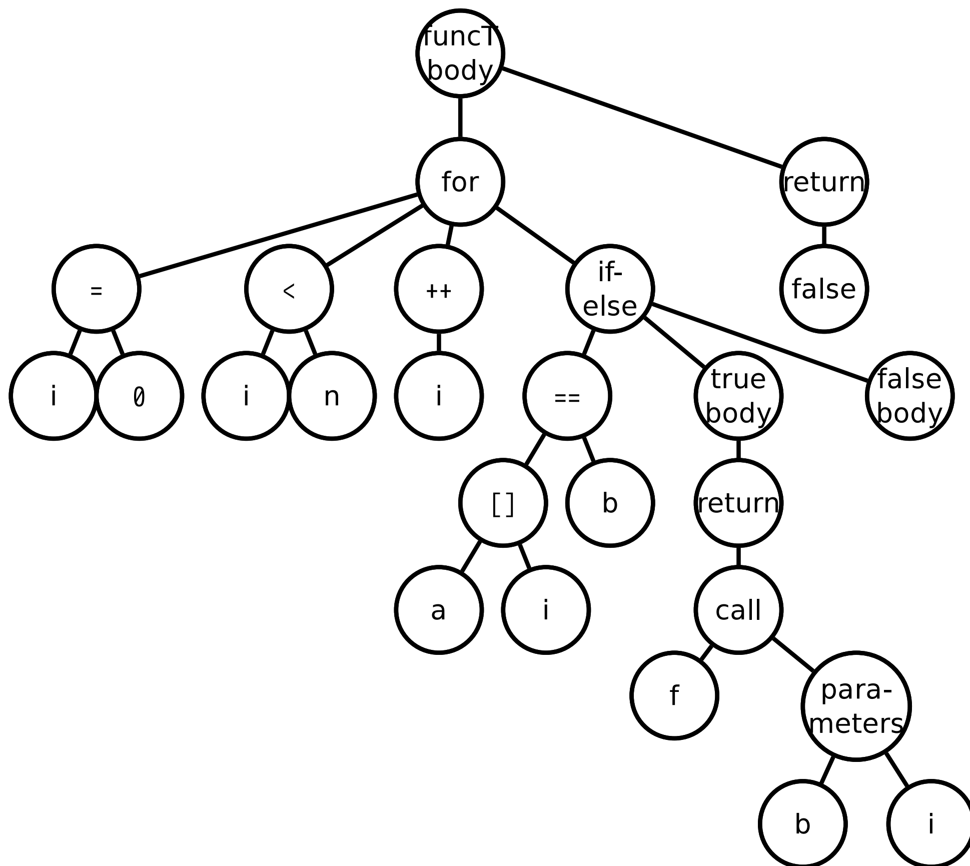


図 3.2: コード 3.1 の構文木の例

しかしながらアーキテクチャに依存した形に RTL を作ることは可能であり、特に最適化に関する RTL 生成はアーキテクチャ依存であることが多い。ただしその場合はアーキテクチャが対応してるか否かを判別するために次項に紹介する Machine Description が使われるため、共通のミドルエンドが使われることに変わりはない。

RTL はプログラム上はツリー構造として扱われるが、デバッグ表示や次のバックエンドで使う Machine Description のため、S 式を用いた表現が用いられている。例として、ある仮想レジスタに直接値 20 を乗算する命令を表す RTL の S 式表現は以下のようになる。

コード 3.2: レジスタに 20 を乗算する命令の RTL 表現例

```
(set (reg/f:SI 54 virtual-stack-vars)
     (mult:SI (reg:SI 58)
              (const_int 20 [0x14])))
```

この例では (reg:SI 58) で表される仮想レジスタの値と定数 20 との積を、(reg/f:SI 54) で表されるレジスタにセットしている。ミドルエンドでは GIMPLE を元にこの様な RTL の命令列を作成し、バックエンドに処理を引き渡している。

#### 最適化パス

最適化は GCC の中でももっとも重要な機能の一つといえる。様々な最適化の手法が GCC において実装され、実用化されている。

GCC では最適化は 2 つフェーズに分類される。

一つは GIMPLE を対象とした最適化である。 GIMPLE は、アーキテクチャはもちろん言語仕様にも依存しないため、どのコンパイラにおいてもこの最適化を適用することができる。

もう一つは RTL を対象とした最適化である。 RTL のデータ構造自体は言語にもアーキテクチャにも依存はないが、最適化にはレジスタの数やスタックの操作法などに依存する事が多いため、この最適化ではいくつかの制限が入る。

ミドルエンドには “pass” という概念があり、最適化処理や GIMPLE の変換、その他諸々の処理は、その処理のメインルーチンを pass に登録することでミドルエンド上にて実行可能になる。

pass の登録順序にも意味があり、pass の前半部は GIMPLE 対象の最適化など、続いて GIMPLE から RTL への変換処理、後半部には RTL の最適化処理が登録されている。

次章で説明するが、本研究では軽量継続の実装に GIMPLE 対象の最適化である末尾呼び出し最適化を利用している。そのため、CbC の言語実装であるがミドルエンドの修正も行っている。

### 3.2.3　バックエンドと Machine Description

バックエンドでは、ミドルエンドで生成された RTL を元にアセンブラを出力している。この処理は必然的にターゲットとするアーキテクチャにより処理が異なるため、バックエ

ンドはアーキテクチャ毎に用意されることになる。

アーキテクチャ毎に異なる RTL の変換規則を記述したものが Machine Description（以下 md）である。md は GCC の対応する全てのアーキテクチャにそれぞれ用意されており、バックエンドはこれを元にアセンブラを生成する。

md は RTL と同じく S 式で表現され、RTL の変換のために次の要素を定義する必要がある。

- その変換規則の名前

  GCC のプログラムから関数として呼び出すための名前である。

- 変換する RTL の構造（パターンマッチ）

  この規則がどのような RTL を変換できるかを表す。

- 変換する条件

  上記のパターンだけでは判別できない時の追加条件を C の構文で記述する。

- 出力するアセンブラ

  アセンブラ文字列か、もしくはアセンブラ文字列を出力する C の構文を記述する。

例として ARM アーキテクチャにおける md を一つ、コード 3.3 に示す。この md はコード 3.2 で紹介した乗算命令の RTL にマッチし、アセンブラ "mul r0 r2 r1" を出力する。2 行目の要素がマッチする RTL のパターンで、コード 3.2 と形が似ていることが分かる。5 行目が条件である。バックエンドプログラムの変数などをチェックしている。そして 6 行目が出力するアセンブラである。ここでは "%?" や "%2" を使い、 printf 関数と似たような書式変換を行っている。

コード 3.3: ARM での Machine Description の例（コード 3.2 をアセンブラに変換）

```
1 (define_insn "*arm_mulsi3"
2   [(set (match_operand:SI          0 "s_register_operand" "=&r,&r")
3         (mult:SI (match_operand:SI 2 "s_register_operand" "r,r")
4                  (match_operand:SI 1 "s_register_operand" "%?r,0")))]
5   "TARGET_32BIT && !arm_arch6"
6   "mul%?\\t%0, %2, %1")
```

**md からソースコードへの変換**

md の記述は上記の様に単なる生成規則でしかない。そのため通常のプログラムであれば実行時に md データを読み込みその通りに解釈する方法を取るが、コンパイラの様な大規模なソフトウェアではそれでは処理に時間がかかりすぎる。

そのため GCC ではこの md を直接プログラムに変換する手法を取っている。例として、i386.md（x86 アーキテクチャの生成規則である）は insn-emit.c や insn-output.c

などの、C 言語ソースファイルに変換され、バックエンドやその他のソースファイルと一緒にコンパイルされ、cc1 プログラムの一部となる。この様子を図 3.3 に表した。

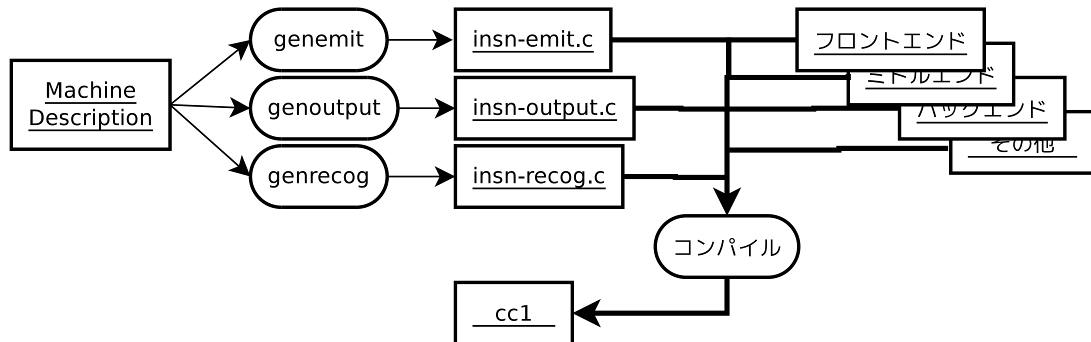


図 3.3: md からソースコードを生成、さらに cc1 をコンパイルする様子

## 3.3　GCC

以上のように GCC はフロントエンド、ミドルエンド、バックエンドがそれぞれの役割を持ち、全体を通して最終的にアセンブラの生成を行う。

GCC ではこのようにアセンブラを出力した後、アセンブル、リンクまでを行う。しかしそれらは本研究では関連しないので説明は割愛する。

# 第4章　GCCにおける実装・改善

この章では、GCC における CbC コンパイラの実装方法の説明と、2 章にて示した項目の実装を行う。

実装には GCC のフロントエンドである cc1 というプログラムを直接変更する。この cc1 は C からアセンブラへ変換を行う純粋なコンパイラとして実行されるプログラムである。この cc1 を CbC の構文解析に対応させる。

過去の研究においては GCC のバージョン 4.2.3 が用いられた。現在は GCC のリリースに並ぶ形で 4.4.2（2010 年 1 月時）を用いている。

## 4.1　過去の研究における実装部分

今回の改善においての予備知識として、過去の研究での実装部分であるコードセグメントと軽量継続がどのように実装されたかを簡単に説明する。

### 4.1.1　コードセグメントの実装

コードセグメント内部の実装は実際は単なる関数で良い。変更の必要があったのは関数の返り値に当たる部分である。コードセグメントでは返り値が存在しないのでここは“code” キーワードを入力できるようにする。この code は内部で void 型に変換する。

GCC（及び一般的なコンパイラ）ではコンパイルに必要な全ての要素、変数や式、関数、構文などをすべて Generic と呼ばれる構文木に保持している。よって Generic を生成する Parser のルーチンにおいて、コードセグメントの構文木を関数の構文木と同じように作成すれば良い。

コード 4.1 はその構文木を作成している部分である。

コード 4.1: 構文木生成 (gcc/c-typeck.c)

```
if (is_code_segment)
    t1 = build_code_segment_type (valtype, TYPE_ARG_TYPES (t2));
else
    t1 = build_function_type (valtype, TYPE_ARG_TYPES (t2));
t1 = build_type_attribute_variant (t1, attributes);
return qualify_type (t1, t2);
```

build_code_segment_type 関数においてコードセグメントの構文木を作成している。内部の処理は build_function_type とほぼ同じだが、関数のテーブルに登録せず、軽量継続の際にそれがコードセグメントであることを示すためのフラグをセットしている。

### 4.1.2 軽量継続の実装

軽量継続は GCC の末尾呼び出し最適化の機構を用いて実装する。

#### 末尾呼び出し最適化

プログラム中、関数を呼び出すときには通常はスタックを積み上げ、現在の環境を保持した上で呼び出し先の処理を行う。これは元の関数に復帰して残りの処理を続行する必要があるためである。しかし関数の最後、リターン直前に呼び出しを行う場合は環境を保持する必要がない（図 4.1 参照)。そのためスタックの状態を変更することなく呼び出すことができる。この最適化は末尾呼び出し最適化 (tailcall) と呼ばれている。

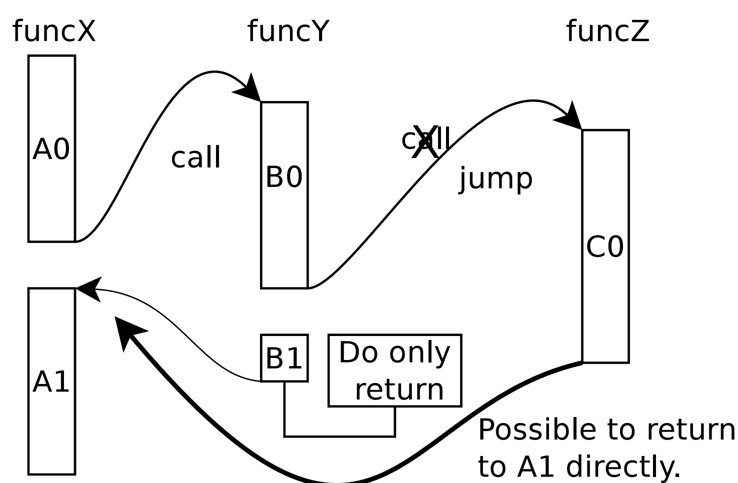


図 4.1: 末尾呼び出し最適化が可能な関数 funcY の例

Scheme 処理系では仕様上この最適化が必須となっているが、C はそうではない。しかし GCC はこの最適化をデフォルトで行っている。

#### 軽量継続への適用

tailcall をコードセグメントの呼び出しに適用することで軽量継続が実装できる。具体的にはソースコード上にコード 4.2 のような式があった場合に、これをコード 4.3 と同じ

ように解釈する。つまり、“goto” が前置する関数呼び出しは、必ず後ろに `return;` がつくと解釈するのである。これで tailcall の条件が一部満たされる。
この構文解析は GCC フロントエンドの gcc/c-parser.c 内で行う。

コード 4.2: goto 文の例

```
code somesegment( ... ) {
   if (..  ) {
      /*  */
      goto nextsegment( ... );
   } else {
      goto nextsegment( ... );
   }
}
```

コード 4.3: 構文木での解釈

```
code somesegment( ... ) {
   if (..  ) {
      /*  */
      nextsegment( ... );
      return ;
   } else {
      nextsegment( ... );
      return ;
   }
}
```

しかし構文木の変更だけでは tailcall が行われるとは限らない。引数などが条件を満たさないとは最適化はカットされる場合がある。そのため最適化を判断する条件式を修正、また構文木から中間コード RTL を生成する部分でも修正が必要になる。

**expand-call** 関数は関数を表す構文木から RTL を生成する処理である（cc1 のミドルエンドに当たる)。この関数内では呼び出される関数のアドレスを取得するコードの生成、スタックへの引数をプッシュするコードの生成、引数のプッシュの度に tailcall が可能かのチェックなどが行われている。
問題となるのは tailcall の可否をチェックする部分である。ここでは主に以下の条件で tailcall が不可能だと判定される。

- 呼出先関数の全引数が占めるスタックサイズが、呼出元関数のそれより大きい場合
- 引数を順にスタックに格納すると、書き込み前のデータが上書きされてしまう場合

そのため、この条件を回避するための処理が必要となる。
スタックサイズの問題に関しては、呼出元関数のスタックサイズをごまかす方法をとった。全てのコードセグメントは一定の（今回は 4096）バイト数のスタックサイズを持つと決めうちすることでこの条件は回避できる。
引数を書き込む順番の問題は、書き込む順序を工夫することで回避した。書き込んでも次に読み込む引数に影響を与えない引数から順に書き込むように実行順序を操作する。
この二つの処理はどちらもミドルエンドの `expand_call` 関数内で行われている。
上記処理の追加により軽量継続が実装された。継続の際にコードセグメントに渡す引数は関数と同じようにスタック上に格納されるが、このスタックは拡張することはなく、図 4.2b のように連続した継続の中でスタックポインタは常に同じアドレスを指し示す。（比較のため、図 4.2a には関数呼び出しの際のスタックの状態を例示した）

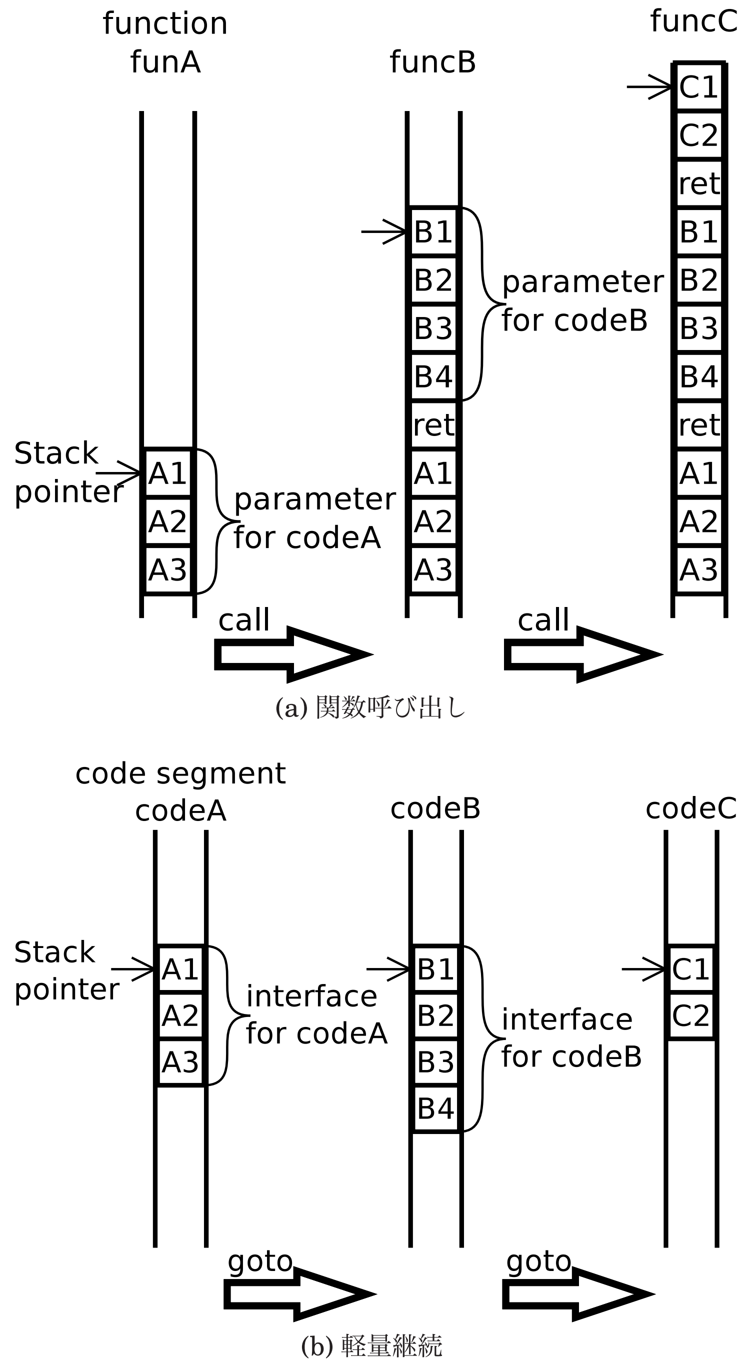


図 4.2: 継続制御と関数呼び出しでのスタックの違い

しかし、引数の書き込み順序を変更するだけでは、複数の引数の格納位置が互いに影響し合うような場合には正しいコードを生成することができないでいた。

## 4.2　本研究における実装

ここから、2.6 節で示した項目について、それぞれ実装方法を説明する。

### 4.2.1　並列代入

前節で説明した様に、コードセグメントへの継続の際の引数書き込みが、別の引数の読み込みに影響を与えるような場合に正しく引数を渡せないという問題がある。

前の実装の際には、ミドルエンドの `expand_call` 関数という関数呼出の RTL を生成するルーチンにおいて、引数格納の順序を工夫することでこの問題を一部回避していた。

しかし完全に、任意の引数の組み合わせでも引数渡しを可能にするにはこの処理だけでは足りず、“並列代入” を導入する必要がある。

#### 並列代入とは

複数の変数に同時に値を代入する事を並列代入 (Parallel Assignment) という。例えば Python では `a, b = 0, 1` として並列代入を行える。この場合は単純に二つの代入を順に実行したものと結果は同じだが、`a, b = b, a` という場合には結果は同じにはならない。順に `a=b, b=a` と分割すると元の `a` の値が失われてしまうからである。処理を正しく行うには、一部の変数の値を一時変数に保持するなどの処理が必要である。

CbC の継続制御ではコード 4.4 の場合などに並列代入が必要になる。これは継続元の引数の格納場所と継続先のそれが互いに逆の位置にあるからである。

コード 4.4: 並列代入の必要な軽量継続の例

```
code somesegment(int a, int b) {
    /* do something */
    goto nextsegment(b, a);
}
```

#### 一時変数への退避

そのため、このような場合に並列代入を行うことでこの問題が解消できる。しかし実際には GCC は元より並列代入を実装しているため、独自の並列代入の実装は必要としない。余分な一時変数への確保は最適化により省かれるが、この最適化を利用して、継続制御の引数渡しを並列代入にする。

この実装では一時変数に全ての引数を退避する手法をとった。具体的には、コード 4.5 の様に、一旦全ての引数を局所変数に代入し、それらの局所変数を継続の引数とする。

コード 4.5: 引数の退避

```
code somesegment(int a, int b) {
    /* do something  */
    int a1, b1;
    a1 = a;
    b1 = b;
    goto nextsegment(b1, a1);
}
```

この処理はもちろんユーザがソースコードで行うのではなく、GCC が自動で判定してそのような構文木を生成するべきである。

こうすることで引数が一時変数に確保され、その後そこからコピーする形で所定のメモリ位置に戻されるため問題が回避できる。

#### 最適化による並列代入

この手法でどのように引数を入れ替えても正しく代入可能になる。ただし、一時変数の使用は処理速度に問題がある。特にレジスタの少ないアーキテクチャでは一時変数の確保にメモリ上のスタックを用いるため、余計なメモリアクセスや冗長な命令が増えてしまう。このため、この手法を実践したコードではそうでないコードに比べて若干の速度低下が見込まれる。

その代わり、この余分な一時変数への退避の生成は GCC の最適化により最小限に抑えられるものである。これにより、全ての引数を一時変数にとるという命令列は最小限の一時変数を使うことで並列代入と同じ効果が得られると考えられる。

そのため、最適化を有効にした場合はこの処理速度の低下は起きないと考えられる。この影響に関しては 5 章にて検証する。

#### 一時変数への退避の実装

この手法の実装は、中間コード生成時ではなく構文木生成で可能である。tailcall の関数呼び出しを表す構文木の生成時に以下の処理を追加する。

1. 関数呼び出しを表す構文木 a の取得
2. a から引数を表す構文木を取得、それぞれについて
    (a) 同じ型の名前なし一時変数を作成
    (b) 引数の値を一時変数に代入
    (c) 関数に渡す引数を一時変数に変更

3. 呼び出す関数がポインタだった場合
   (a) 関数と同じ型（関数ポインタ）の一時変数を作成
   (b) 関数アドレスを一時変数に代入
   (c) 呼び出す関数を一時変数に変更

ここでは関数ポインタも引数と同じように扱い、一時変数に退避する。実際のプログラムはコード 4.6 の様になる。この関数は継続制御の構文木を生成した際に呼び出されるフロントエンドの関数である。

cbc_replace_arguments は関数呼び出し構文木を引数として受け取り、上記の処理を行う。引数として渡される tree call がその構文木である。build_decl は名無し一時変数の宣言、 build_modify_expr は一時変数への代入を行う構文木の生成をしている。

コード 4.6: 上記の処理を行う関数

```
static tree
cbc_replace_arguments (location_t loc, tree call)
{
  tree args;
  tree fn;
  tree tmp_decl;
  int i=0;

  fn = CALL_EXPR_FN (call);
  if ( TREE_CODE (fn)==PARM_DECL || !TREE_CONSTANT (fn) )
    {
      tmp_decl = build_decl (VAR_DECL, NULL_TREE, TREE_TYPE(fn));
      pushdecl (tmp_decl);

      add_stmt (build_modify_expr (loc, tmp_decl, NOP_EXPR, fn));
      CALL_EXPR_FN (call) = tmp_decl;
    }

  args = CALL_EXPR_ARGS (call);
  for ( ;args; args = TREE_CHAIN (args), i++)
    {
      tree arg = TREE_VALUE (args);

      if ( TREE_CODE (arg)==PARM_DECL || !TREE_CONSTANT (arg) )
        {
          tmp_decl = build_decl (VAR_DECL, NULL_TREE, TREE_TYPE(arg));
          pushdecl (tmp_decl);

          add_stmt (build_modify_expr (loc, tmp_decl, NOP_EXPR, arg));
          CALL_EXPR_ARG (call, i) = tmp_decl;
```

```
        }
    }

  return call;
}
```

ソースコードの構文解析時、軽量継続をパースしてその構文木を生成した際に、この関数 cbc_replace_arguments を実行することで、この軽量継続は並列代入に対応できるようになった。

### 4.2.2 環境付き継続

環境付き継続は過去の研究では実装されていなかった。これは C との互換性のために必要な制御構造である。

環境付き継続には 2.4.1 で述べたように、__return という擬似変数を使う。この変数の値を継続先のコードセグメントに渡すことで、そのコードセグメントから関数の環境へ復帰することを可能にする。渡された__return の値は、コードセグメント側からは他のコードセグメントと区別する必要はない。

この環境付き継続に用いる__return 擬似変数の実装には様々な方法が考えられる。例えば 2.4.1 節で紹介した setjmp/longjmp を使った実装も可能である。しかしこの方法は特に longjmp のオーバヘッドが大きく、また実行環境によっては setjmp/longjmp そのものがないことも考えられる。ポータビリティを考えると GCC の機能で実装することが望ましい。

今回の実装には内部関数をもちいることにした。内部関数は GCC による C の拡張機能である [14]。

#### GCC により追加されるコード

環境付き継続で使う__return 変数は特殊なコードセグメントへのポインタとなる必要がある。このコードセグメントはユーザでは定義せず、その変数を参照した関数の返り値型を基にコンパイラが自動で生成する事が望ましい。

具体的には、コード 4.7 の関数 funcB をコンパイラは次のコード 4.8 の様に解釈し、内部コードセグメントを自動生成する。

コード 4.7: __return の例

```
int funcB()
{
    code (*ret)(int);
    ret = __return;
    /* do something.  */
    goto cs(ret);

    /* never reached.  */
    return -1;
}
```

コード 4.8: コード 2.3 の funcB に追加される処理

```
1  int funcB()
2  {
3      code (*ret)(int);
4
5      int _retval;
6      code _segment(int _val){
7          _retval = _val;
8          goto _label;
9      }
10     if (0) {
11       _label:
12         return _retval;
13     }
14     __return = _segment;
15
16     ret = __return;
17     /* do something.  */
18     goto cs(ret);
19
20     /* never reached.  */
21     return -1;
22 }
```

5–14 行が GCC により追加される処理である。内部コードセグメント_segment は受け取った引数を関数の返り値として保持し、ラベル_label に jump する。この時点で内部コードセグメントを抜けて元の関数 funcB の環境に復帰する。

さらに jump 先も GCC により自動で追加される。しかしこの jump 先は_segment 以外からは実行してはならない。そのため条件式が真にならない if 文で囲み、実行を回避している。jump 先での処理は、_segment 内で代入された値を持ってリターンするのみである。

### 内部コードセグメント自動生成の実装方法

GCC は変数や関数、また文字列や数値などのリテラルに関する処理を c_parser_postfix_expression で行っている。この関数では変数や数値、文字列などの判定に 500 行にわたる switch 文を使っているが、ここに__return の判定も追加する。

必要な処理は以下の様になる。

- ラベル_label の宣言
- 返り値を保持しておく変数の宣言

- 内部関数の定義
- 条件分岐制御の構文木生成
- 条件分岐内でのラベルの定義
- 条件分岐内での復帰構文の構文木生成

参考のため付録 A-2 にこの処理のコードを掲載する。
ここでは実際に出力されるアセンブラをコード 4.9 に示す。

コード 4.9: 4.8 の funcB で出力されるアセンブラ (x86)

```
 1 funcB:
 2     pushl    %ebp
 3     movl     %esp, %ebp
 4     subl     $1080, %esp
 5     leal     -24(%ebp), %eax
 6     movl     %ebx, -12(%ebp)
 7     leal     -24(%ebp), %ebx
 8     movl     %eax, -48(%ebp)
 9     movl     $_segment.1243+2, %eax
10     leal     -48(%ebp), %edx
11     subl     %ebx, %eax
12     leal     -36(%ebp), %ecx
13     movl     %esi, -8(%ebp)
14     movl     %edi, -4(%ebp)
15     movl     %esp, -44(%ebp)
16     movb     $-71, -36(%ebp)
17     movl     %edx, -35(%ebp)
18     movb     $-23, -31(%ebp)
19     movl     %eax, -30(%ebp)
20     call     cs
21 .L5:
22     movl     -12(%ebp), %ebx
23     movl     -8(%ebp), %esi
24     movl     -4(%ebp), %edi
25     movl     %ebp, %esp
26     popl     %ebp
27     ret
28 .L3:
29     addl     $24, %ebp
30     movl     retval.1240, %eax
31     jmp    .L5
32 _segment.1243:
33     pushl    %ebp
34     movl     %esp, %ebp
35     movl     8(%ebp), %eax
36     movl     %eax, retval.1240
37     movl     $.L3, %eax
38     movl     (%ecx), %ebp
39     movl     4(%ecx), %esp
40     jmp    *%eax
41     .local   retval.1240
```

この出力によると、`funcB` は cs に継続（20 行目）する前にいくつかのレジスタをスタック領域に確保している。これは内部関数が呼ばれた際にこの `funcB` の環境を再現するためのものである。内部関数は`_segment.1243` として表されており、`jmp *%eax` をもって`.L3` にジャンプし、最終的に`.L5` のコード内で関数 `funcB` からリターンする。

以上でコード 4.8 に示すような処理がコンパイル時に自動で追加され、環境付き継続の使用が可能になった。

### 関数からの継続

ここで軽量継続の実装に tailcall を用いたことの弊害がでてくる。4.1.2 節の実装では関数からの継続は考慮していない。通常の継続の際は現コードセグメントのもつ引数は保

持しないため、その環境やスタック領域は破壊される。同様に関数から直接継続しようとすると、その関数やその関数を呼び出した関数の持つ環境（スタック）を破壊してしまうことになる（23 ページ、図 4.2b）。

この問題を回避するため、関数からの継続に限り、スタックを拡張し関数の環境を保持する手法をとった。この動作は本来の軽量継続の概念とは相容れないものだが、C との互換性維持のために必要である。また、CbC 部分での軽量継続ではいずれもスタックは定常なので、CbC の目的である検証、状態遷移記述などの問題にはならない。

### 4.2.3　PowerPC における間接継続

軽量継続の実装に tailcall を用いたことは 4.1.2 で説明した。しかし、実際には tailcall が行われないアーキテクチャがいくつか存在する。PowerPC もその一つで、このアーキテクチャでは間接呼び出しの場合は tailcall が行われない。これは tailcall を表す RTL をアセンブラに変換する md が定義されていないためである。このため、これまで PowerPC での間接継続は変換規則がみつからないというコンパイルエラーで実行できなかった。

間接呼び出しの tailcall には専用の RTL 表現がある。PowerPC で問題となるのは、この RTL からアセンブラへの変換が定義されていないことである。この問題に対処するため、PowerPC アーキテクチャにおける md を記述する。

#### 間接 tailcall の RTL と Machine Description

GCC では関数呼び出しは全て一つの RTL に置き換えられる。これは tailcall が行われた場合も、呼び出し関数がポインタである場合も同様である。しかし tailcall かポインタかによって RTL の形が異なるため、PowerPC ではこの両方の場合（つまり間接呼び出しの tailcall）の RTL の規則が md で定義されていない。そのため、これはエラーになる。

この問題となっている RTL を次のコード 4.10 に示す。

コード 4.10: PowerPC における間接継続の RTL

```
(call_insn/j 25 24 0 (parallel [
            (call (mem:SI (reg/f:SI 129) [0 S4 A8])
                (const_int 256 [0x100]))
            (use (const_int 0 [0x0]))
            (use (reg:SI 130))
            (return)
        ]) -1 (nil)
    (nil)
    (expr_list:REG_DEP_TRUE (use (reg:SI 6 r6))
        (expr_list:REG_DEP_TRUE (use (reg:SI 5 r5))
            (expr_list:REG_DEP_TRUE (use (reg:SI 4 r4))
                (expr_list:REG_DEP_TRUE (use (reg:SI 3 r3))
                    (nil))))))
```

この RTL 内の (mem:SI (reg/f:SI 129) が関数のポインタを示すレジスタである。間接呼び出しでない場合はこれが (mem:SI (symbol_ref:SI (``cs0'') となり、コードセグメントの関数を直接表している。また、下に続く expr_list は引数の列である。この RTL の表す継続制御の引数がこの S 式で表されている。

PowerPC をこの間接継続に対応させるにはこの RTL に対応する md を定義する必要がある。

### 間接継続の md

PowerPC において間接継続を実装するには、上記の RTL を変換する md を記述すれば良い。この RTL に近い形が間接でない tailcall の md として使われているのでそれを使用する。 次のコード 3.3 が新しく記述された md である。

コード 4.11: 4.10 の変換規則

```
(define_insn "*sibcall_indirect_nonlocal_sysv<mode>"
  [(call (mem:SI (match_operand:P 0 "register_operand" "c,*l,c,*l"))
         (match_operand 1 "" "g,g,g,g"))
   (use (match_operand:SI 2 "immediate_operand" "O,O,n,n"))
   (use (reg:SI LR_REGNO))
   (return)]
  "DEFAULT_ABI == ABI_V4
   || DEFAULT_ABI == ABI_DARWIN"
{
  if (INTVAL (operands[2]) & CALL_V4_SET_FP_ARGS)
    output_asm_insn ("crxor 6,6,6", operands);

  else if (INTVAL (operands[2]) & CALL_V4_CLEAR_FP_ARGS)
    output_asm_insn ("creqv 6,6,6", operands);

  return "b%T0";
}
  [(set_attr "type" "jmpreg,jmpreg,jmpreg,jmpreg")
   (set_attr "length" "4,4,8,8")])
```

このコードの 3 番目の要素はコード 4.10 の RTL とよく似ていることがわかる。これは変換対象としてこの型に合うものに制限するためである。

ここでは出力するアセンブラとして b%T0 が使われている。%T0 はレジスタ名に置き換えられる部分である。このアセンブラは最終的には bctr と置き換えられて PowerPC のアセンブラとして出力されることになる。

### 4.2.4　x86 における引数渡し

コードセグメントの間の軽量継続は、C の関数呼び出しと同じように引数を渡すことができる。関数呼び出しでのこの引数の渡し型はほとんどの場合アーキテクチャやオペレーティングシステム、また各プログラミング言語毎に違った規約があり、これは一般に呼出規約 (Calling convention) と呼ばれている。

CbC では同じアーキテクチャでもコンパイラによってこの呼出規約は異なる。mc の軽量継続では、なるべく多くの引数をレジスタに格納するようになっており、PowerPC では最大 11 個の int 型をレジスタに格納する。レジスタの少ない x86 でも 2 つだけだが、やはりレジスタを使用している。

GCC ベースコンパイラでは継続制御の引数渡しに関数の呼出規約と同じ方法を使っている。そのため、x86 では引数渡しに全てスタックを用いることになり、mc に比べて速度低下がみられた。

引数渡しにレジスタを使用できるようにすることでこの問題を解決したい。

#### fastcall

そもそも引数渡しがスタックだけだということは、CbC だけでなく C においても速度面で問題をはらんでいる。そのため GCC ではもとより、x86 でのレジスタ渡しを可能にする拡張機能を実装している。それが fastcall である。

この fastcall も使用するレジスタ数は 2 つだけではあるが、継続制御でもこれを使うことにより高速化が図れるはずである。

#### コードセグメントを全て fastcall に

通常、GCC の拡張機能を用いて関数を fastcall にするにはコード 4.12 の様に “attribute” キーワードを関数宣言の後ろに記述する。

コード 4.12: fastcall な関数 fastfunc を宣言する例

```
int fastfunc() __attribute__((fastcall));

int fastfunc(int a, int b) {
    /* do something.    */
}

void normalfunc() {
    fastfunc(20, 30);
}
```

しかし全てのコードセグメントに対してこの属性を宣言するのは現実的でなく、mc とのソースコードレベルの整合性もとれない。そこで GCC ではコードセグメントの解析時に全て fastcall 属性を付加することにする。

具体的には「型」の構文解析の際、キーワード "code" で関数の型が宣言されている場合に、属性値を表す構文木を付加する。c_parser_declspecs 関数が「型」に関する構文解析部である。この関数内の型名キーワードを処理する switch 文内で、"code" のみ fastcall 属性を付加する。

コード 4.13 がその処理である。このコードの 12–14 行目が fastcall 属性付加の処理になる。それ以外の行は void や int など他の型の処理と変わらない。

コード 4.13: c_parser_declspecs におけるキーワード "code" の処理

```
case RID_CbC_CODE:
    if (!typespec_ok)
        goto out;
    attrs_ok = true;
    seen_type = true;
    if (c_dialect_objc ())
        parser->objc_need_raw_identifier = true;
    t.kind = ctsk_resword;
    t.spec = c_parser_peek_token (parser)->value;
    declspecs_add_type (specs, t);

    attrs = get_identifier("fastcall");
    attrs = build_tree_list(attrs, NULL_TREE);
    declspecs_add_attrs(specs, attrs);

    c_parser_consume_token (parser);
    break;
```

この処理で全てのコードセグメントが fastcall 対応となり、軽量継続の際にはレジスタ ecx,edx に引数をのせることが可能となる。

### 4.2.5 プロトタイプ自動生成

C のプロトタイプ宣言はコンパイル時のエラー検出に役立っている。しかし CbC のコードセグメントには返り値は存在しない。また状態遷移記述という性質上、プログラムを記述する際は上から下に実行順にコードセグメントを並べることが多いため、プロトタイプ宣言をするとそれが膨大な数になる。

また、mc ベースコンパイラの方ではプロトタイプ宣言を減らすため、一種の簡単な型推論を実装している。そのためこれまでに作られた CbC のプログラムでは特殊な場合を除いてプロトタイプ宣言がほとんどなく、GCC でコンパイルする際に問題となる。

これらの問題に暫定的に対処するため、Python を用いてプロトタイプの自動生成を行うスクリプトを作成した。このスクリプトでは関数の定義部を正規表現で検索し、マッチする部分を変換して関数宣言として出力する。この出力例として、6 ページにある階乗計算を行うコード 2.2 をスクリプトに通した結果を、コード 4.14 に示す。

コード 4.14: プロトタイプ自動生成スクリプトの出力例

```
/* defined in file factorial.cbc at offset 0 */
code print_factorial (int prod);

/* defined in file factorial.cbc at offset 83 */
code factorial0 (int prod, int x);

/* defined in file factorial.cbc at offset 218 */
code factorial (int x);

/* defined in file factorial.cbc at offset 272 */
int main (int argc, char **argv);
```

このスクリプトの全コードは付録 A-3 に掲載する。

このプトロタイプ自動生成により、CbC プログラムのヘッダファイルを自動生成することができる。プログラムではこのヘッダファイルをインクルードすることで、micro-c コンパイラとの互換性を確保することができる。

# 第5章　評価・考察

本章では本研究の評価を行う。

## 5.1　本研究での改善による成果

本研究では、2008年に実装されたGCCベースコンパイラの改善を行った。まずはその改善による成果をここで述べる。

**並列代入**

並列代入の改善により、これまで存在した軽量継続の際のバグが取り除かれた。特に引数で渡されたコードセグメントポインタへ継続する際に出ていたバグに対する影響が大きい。

**環境付き継続の実装**

この実装により、Cとの互換性が確保できた。これにより名実ともにCwCコンパイラとして完成したと言える。

**PowerPCでの間接軽量継続**

これまで実質的にはPowerPCでは使用不能であった。

本研究室ではPS3を用いた研究も行っており、その研究ではPowerPCアーキテクチャが必要となる。この問題の解決により、当研究室の提案するCeriumはCbCベースへの移行が可能になる。

**プトロタイプ宣言の自動生成**

GCCとmicro-cの間にある、コードセグメントの宣言に関する差異が、この自動生成によって改善された。これにより、これまでmicro-c用に作成されていたプログラムはほとんど修正することなく動く。

**x86でのfastcall**

未だに主流であるx86アーキテクチャ（x86_64への移行は進みつつあるが）において、若干の速度低下が見られていたものを改善した。この測定については5.3節で行う。

## 5.2 GCC を使うことの利点・欠点

これまで CbC のコンパイルに使用してきた micro-c に対し、新しく GCC が CwC のフルセットとして使用可能となった。ここで GCC を用いることの利点と欠点について考察する。

### アーキテクチャ

micro-c においては PPC, x86, MIPS, ARM, SPU など、多数の CPU アーキテクチャをサポートしてきた。しかし他の CPU に新しく対応するには多大な時間、労力が必要となる。GCC は現在、既に 20 を越える CPU に対応しており、また OS 毎の ABI の差異も吸収可能である。これは GCC をコンパイラとすることの最大の利点である。

またそれだけでなく、GCC は新しいアーキテクチャへの対応も早い。この特徴は、GCC がフロントエンドとバックエンドという形で言語実装とアーキテクチャを分離していることからくる。一般的に新しい CPU アーキテクチャが開発された場合にはその開発者自身が GCC にコミットすることが多いため、組み込み用途を目的の一つとする CbC ではよりその強みがます。

### 最適化の恩恵

GCC は豊富な最適化機構を備えている。代表的な最適化だけでもループ最適化、分岐スレッディング (jump threading) 、共通式除去 (common subexpression elimination)、命令スケジューリング (instruction scheduling) などがある。

とくに、プログラムにおいては類似した形の式 (expression) を扱うことがよくあるため、共通式除去は非常に効果が高い。同様の効果は同じ式を保持する変数を用意することでも実現できるがソースコードの修正が必要になる。micro-c にはこの最適化は含まれていないため、複雑な計算式を含むプログラムにおいては GCC の方が良いコンパイル結果を示すものと考えられる。

### デバッガ

これまで CbC にはデバッガが存在しなかった。デバッガの実装には出力するアセンブラに行番号や変数名、関数名などの情報を付加する必要があるが、GCC は標準でこれを行っている。そのため C のデバッガとして広く一般的に使われている gdb をそのまま CbC のデバッガとして使用することが可能であり、ソフトウェア開発の大きな助力となる。

### 関数呼出しの名残り

上記の利点に対し、GCC であるゆえの欠点も存在する。

本研究による軽量継続制御の実装には 4 章で説明したように関数の末尾最適化を利用した。それゆえコードセグメントのアセンブラ出力の命令列には関数呼び出し時のスタック処理が一部残ってしまうことが分かっている。特にレジスタの少ないアーキテクチャ、x86 などではそれが顕著に現れる。

micro-c ではコードセグメントと関数は完全に別物として取り扱っており、この様なスタック操作はコードセグメントには現れないため、このオーバヘッドが GCC では不利な点である。

### 互換性、ABI

また、同じく関数呼び出しの名残りから、GCC では micro-c とのバイナリレベルでの互換性がない。つまり GCC でコンパイルしたコードセグメントから micro-c でコンパイルしたコードセグメントに継続することはできない。

これは micro-c での軽量継続の ABI が関数とはまったく異なるものだからである。今回は tailcall を実装に用いたため、関数としての制限があり、micro-c の ABI に合わせることはできなかった。

この問題は GCC の欠点というわけではないが、CbC ベースの共有ライブラリを生成・使用する場合には注意が必要となる。

## 5.3 性能評価

次にコンパイラの性能評価を行う。

### 5.3.1 評価項目、比較対象

コンパイラの出力した実行ファイルを複数回実行し、その実効速度を測定する。CbC は実用的なプログラムの記述を目的としているので、プログラムの動作速度は性能の評価として妥当だと考えられる。

またもう一つの項目として、出力した実行ファイルのファイルサイズも評価する。一般的なプログラムではファイルサイズを気にすることは少ないが、CbC の用途には組み込みなども考えられているため、ファイルサイズの影響は大きい。比較する際は strip コマンドを用いてデバグ情報等を取り除いている。

実効速度、ファイルサイズの比較対象として 2 つ用意した。一つは過去の研究での GCC ベースコンパイラ、つまり今回の改善を含めてないものである。こちらは GCC のバージョン 4.2.3 をベースとしている。

もう一つの比較対象には micro-c ベースのコンパイラを用いる。さらに GCC では最適化による効果も評価するため、***1)*** 最適化なし "-O0" ***2)*** 速度最適化 "-O2 -fomit-framepointer" ***3)*** サイズ最適化 "-Os" についてもそれぞれ比較する。

### 5.3.2　評価手法と環境

実行するプログラムとして、クイックソートのテストプログラムを作成した。クイックソートは再帰呼び出しを伴うため、スタック操作が必須となる。そのためより様々な状態でコードセグメントへの継続制御が使用されることになり、CbC の性能評価に適していると考えられる。クイックソートは CbC に先立って C で実装し、参考文献 [11] で紹介する手法を用いて CbC に変換した。このプログラムは付録 A-4 に添付する。

測定環境は両コンパイラが対応しているアーキテクチャ、OS から以下の 5 つの組み合わせ [CPU アーキテクチャ/OS 種別] を選択した。（ppc は PowerPC の意である）

- ppc/OS X
- ppc/linux
- ppc/linux on PS3
- x86/OS X
- x86/linux

なお、micro-c は MIPS, ARM にも対応しているが、現在その処理系が用意できなかったので割愛している。また、GCC-4.2.3 ベースコンパイラは ppc では実行不能であったため x86 のみとなる。

各評価マシンの詳細は付録 A-1 に掲載する。

### 5.3.3　評価結果

実効速度の測定結果を表 5.1 に示す。ただし環境毎に CPU 速度は異なるので、上下の比較には意味はない。

実行ファイル strip 後のファイルサイズを表 5.2 に示す。

本研究での実装 GCC-4.4.2 と以前のバージョン GCC-4.2.3 との比較を表 5.3 に示す。こちらは x86 のみ、最適化も-Os は対応していない。

### 5.3.4　評価結果考察

#### 速度面

まずどのアーキテクチャにおいても、GCC の最適化が大きな速度差を生み出している事が分かる。最適化なしと速度最適化を比較すると、x86 では 2.4 倍、ppc では 5〜7 倍

| コンパイラ / CPU/OS | GCC | | | micro-c |
|---|---|---|---|---|
| | 最適化なし | 速度最適化 | サイズ最適化 | |
| x86/OS X | 5.901 | 2.434 | 2.785 | 2.857 |
| x86/Linux | 5.732 | 2.401 | 2.876 | 2.254 |
| ppc/OS X | 14.875 | 2.146 | 2.170 | 4.811 |
| ppc/Linux | 19.793 | 3.955 | 4.013 | 6.454 |
| ppc/PS3 | 39.176 | 5.874 | 6.111 | 11.121 |

表 5.1: アーキテクチャ毎の GCC と micro-c の速度比較（単位: 秒）

| コンパイラ / CPU/OS | GCC | | micro-c |
|---|---|---|---|
| | 速度最適化 | サイズ最適化 | |
| x86/OS X | 9176 | 9176 | 9172 |
| x86/Linux | 5752 | 5752 | 5796 |
| ppc/OS X | 8576 | 8576 | 12664 |
| ppc/Linux | 10068 | 10068 | 9876 |
| ppc/PS3 | 6960 | 6728 | 8636 |

表 5.2: 実行ファイルのファイルサイズ比較 stripped（単位: bytes）

| コンパイラ / CPU/OS | CbC on GCC-4.4.2 | | CbC on GCC-4.2.3 | |
|---|---|---|---|---|
| | 最適化なし | 速度最適化 | 最適化なし | 速度最適化 |
| x86/OS X | 5.907 | 2.434 | 4.668 | 3.048 |
| x86/Linux | 5.715 | 2.401 | 4.525 | 2.851 |

表 5.3: GCC-4.2.3 ベースと GCC-4.4.2 ベースの速度比較（単位: 秒）

もの差が生じている。ただし ppc のこの異常な速度差は 4.2.1 並列代入で示した様に、継続の引数を全て一時変数に入れていることが大きい。その場合最適化なしではすべての引数を一度メモリに確保するので、その分逆に遅くなっているのだと考えられる。しかしながら最適化を有効にすることでそのメモリへの一時変数の確保も解消されるということが分かった。

x86 は OS X と Linux の環境で測定を行った。速度最適化の GCC と micro-c を比べると、 OS X では micro-c に比べて 20%ほど早くなった事が分かる。しかし逆に Linux 環境では 6%の速度低下が示された。どちらにしても ppc ほどの良い結果ではない。これは自由に使えるレジスタが極めて少ないという x86 の特殊なアーキテクチャが要因だと考えられる。そのため GCC の最適化が十分に機能できなかった可能性がある。この 6%の差は実用レベルでは問題なく、プログラムの構成によっては結果は逆転する事も十分にある。

ppc においてはどのオペレーティングシステムでも、速度最適化を使った GCC は micro-c に比べて早い事が分かる。いずれも約 2 倍、もしくはそれ以上に速度が向上している。これは GCC の最適化機構が十分に働いている要因が大きい。

### アセンブラ比較

実際に出力されたアセンブラから速度向上の要因を確かめるため、quicksort プログラムで使用されているコードセグメントを一つ例に挙げる。CbC のプログラムソースがコード 5.1 である。このコードセグメントの速度最適化を使った GCC による出力がコード 5.2、micro-c による出力がコード 5.3 である。どちらもアーキテクチャは ppc である。

コード 5.1: quicksort プログラムで使われているコードセグメント

```
__code  divider_e(QS_IF *recvif,
                  int s, int e, int p,
                  stack sp)
{
   if (p<recvif->v[e]) {
      goto divider_e(recvif, s, e-1, p, sp);
   } else
      goto swapper(recvif, s, e, p, sp);
}
```

コード 5.2: divider_e の GCC による出力 (ppc)

```
quicksort_divider_e:
        lwz 11,0(3)
        slwi 0,5,2
        lwzx 0,11,0
        cmpw 7,0,6
        bgt 7,.L13
        b quicksort_swapper
.L13:
        addi 5,5,-1
        b quicksort_divider_e
```

コード 5.3: divider_e の micro-c による出力 (ppc)

```
quicksort_divider_e:
        la 1,.LC22@l(31)
        addis 1,1,.LC22@ha
        mr 3,27
        slwi 3,3,2
        mr 11,29
        lwz 11,0@l(11)
        add 11,11,3
        lwz 11,0@l(11)
        mr 10,26
        cmpw 7,10,11
        bge 7,.LC24
        mr 10,27
        addi 10,10,-1@l
        mr 27,10
        b quicksort_divider_e
.LC24:
        b quicksort_swapper
```

もっとも比較しやすい箇所は `e-1` の処理である。コード 5.2 の GCC ではこれを 1 命令の `addi 5,5,-1` で行っている。 **micro-c** ではこれが `mr`, `addi`, `mr` という 3 命令になっている。これは変数 `s` の値を一度別のレジスタに移して計算するという処理である。この様な細かい命令の展開が速度に差が出る要因である。

またこの ppc のアセンブラからも、x86 での速度差が少ないことが頷ける。引数のほとんどをメモリに格納する x86 では、計算のために一度レジスタに格納しないといけないことから、この命令は結局 3 命令になるはずであり、実際に x86 では GCC, micro-c 共にそのようなコードが出力されていた。

この結果より、CbC で記述されたプログラムではレジスタが多い方が実効速度の面で有利であるということが分る。これは他のコンパイラ言語でも同じ事が言えるが、(手続きやメソッドにおける) 前の環境を保持する必要がない CbC ではその影響がより強い。

#### ファイルサイズ

次に、実行ファイルのファイルサイズの面から考察する。

実行ファイルのファイルサイズは組み込み用途のプログラムには重要な要素となる。多くの場合、組み込み機器では大容量のメモリは用意されておらず、OS も存在しないため仮想記憶の概念がない。そのためメモリに乗り切らないプログラムはそもそも実行不能である。

まず、評価の主な特徴として、strip 後のファイルサイズ 5.2 をみると、x86 では micro-c

と GCC でほとんど差がない事が分かる。この環境では速度面でも大きな差はなく、micro-c の精度の良さがわかる。

また興味深い特徴として、速度最適化とサイズ最適化の差が ppc/PS3 以外は全くないことも分かった。 サイズ最適化は速度最適化の最適化機能から、ファイルサイズが大きくなるものを除外したものである。評価結果にはサイズ最適化によるファイルサイズの減少はほとんどなく、しかし速度は少々遅くなっている。このことから CbC によるプログラムではサイズ最適化を用いる必要はなく、速度最適化で十分であることが分かった。

#### 以前のバージョンとの速度比較

古いバージョンとの速度差についても考察を重ねる。実行環境に ppc が存在しないのは、4.2.3 節における問題のためである。今回用意したプログラムは間接継続を用いているため、古いバージョンではバグにより実行できなかった。また、速度向上に関する改善は 4.2.4 節における fastcall の追加のみなであり、この fastcall は x86 環境にしか影響しないはずである。

表を見ると、-O0 の場合は新バージョンの方が旧バージョンより遅くなっているのが分かる。これは 4.2.1 節の一時変数への退避処理のためだと考えられる。この処理では、最適化により無駄なスタックへのアクセスは排除されることを期待して実装していた。-O0 は最適化を行わないので、この場合は逆に遅くなっている。これは予想通りの結果である。しかし最適化を行った場合は新バージョンに劣化はない。したがって一時変数への退避処理においては、期待通り無駄な命令は十分に排除されていることが分かった。

また、それだけなら速度はほぼ同じ結果がでるところだが、ここではいずれの環境でも新しいバージョンの方が速い。15–20%ほど高速化していることがわかる。これは本研究で行った改善の一つ、fastcall の影響である。

## 5.4　メンテナンス性の向上に関する取り組み

本研究室ではこれまで CbC コンパイラとして micro-c を利用していた。このコンパイラはベースとなる micro-c には依存せずに、ほぼ独立な開発を続けている。

これに対し GCC は現在も精力的に開発が続けられており、年数回のアップデートではバグの除去や最適化の改善などが行われている。そのため CbC コンパイラでもそのリリースに沿ってアップデートすることが望ましく、実際に今回の改善の際にも 2010 年 1 月現在での最新リリースである 4.4.2 をベースとして行い、本稿執筆中に 4.4.3 へのアップデートが行われた。

しかしアップデートの度に新しいソースコードを書き換えるのは無理があり、現実的ではない。最良の方法は GCC の正式な機能として開発リポジトリにマージしてもらうことだが、現段階ではそこには至っていない。

そのため現在は Mercurial を使ったソースコード管理を行っている。ここではその手法を説明する。

### 5.4.1 二つのリポジトリ

Mercurial は分散型のバージョン管理システムである。開発環境毎に複数のリポジトリを分散して持つすることができ、そのためそれぞれのリポジトリのマージの機能に優れる。

CbC コンパイラの管理ではこの特徴を利用する。具体的には CbC 開発用に二つのリポジトリを持つ。一つは本家の GCC リリースとまったく同一のソースをもった GCC-copy と言うリポジトリである。もう一つはこの GCC-copy からブランチする形で作成した CbConGCC というリポジトリである。こちらが CbC に関するメインの開発環境となる。図 5.1 では中央と右のラインがそのリポジトリを表している。

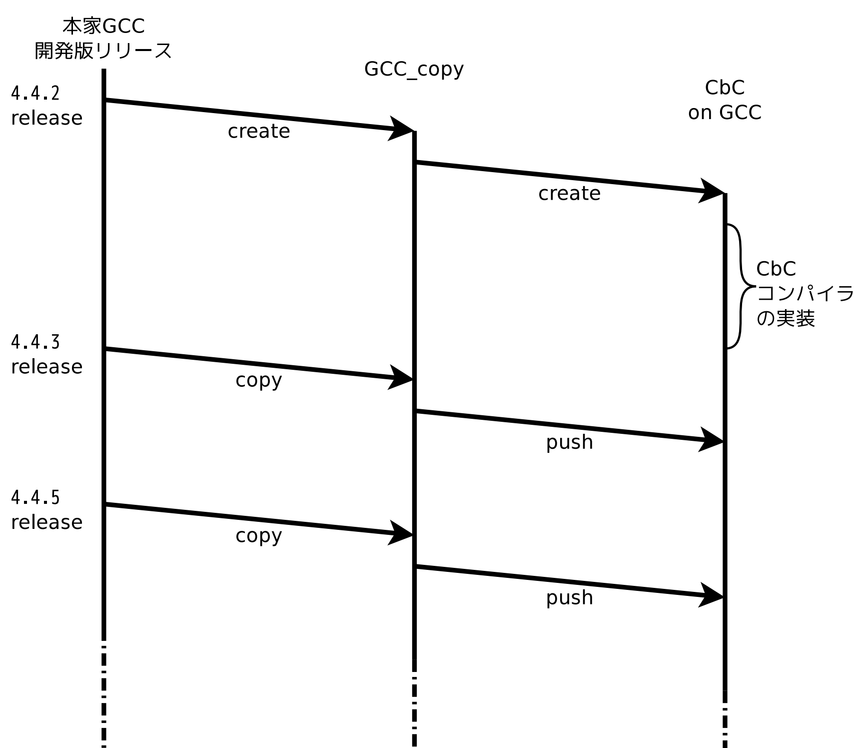


図 5.1: CbC コンパイラ開発でのリポジトリ管理（左が本家のリリースタイムライン、中央が GCC-copy、右が CbC の開発用リポジトリのタイムライン）

新しいバージョンがリリースされた際の CbConGCC でのアップデートは次の手順で実現できる。

- GCC-copy リポジトリにて
  1. GCC-copy リポジトリ中のファイル全てを消す（バージョン管理情報以外）
  2. gcc-core-*version*.tar.gz を展開し、全ファイルを GCC-copy に追加
  3. `hg status` で追加ファイル、削除ファイルを確認

4. コミット
5. gcc-*version* タグの追加

- CbConGCC リポジトリにて

1. GCC-copy から `pull`.
2. `hg merge` でマージ実行
3. 衝突のあったファイルを修正
4. 実際にビルドしてテストファイルが動くことを確認
5. コミット
6. cbc-*version* タグの追加

以上でアップデートが完了する。

### 5.4.2　このリポジトリ管理方法の評価

実際にこのリポジトリ管理方法を用いてアップデートを行った。この評価ではバージョン 4.4.0 から 4.4.2 へのアップデートと、4.4.2 から 4.4.3 へのアップデートを行った。

アップデートの際に何らかの問題が生じるのは CbConGCC リポジトリでの衝突ファイルの修正だけである。4.4.3 へのアップデートでは特になにも衝突することはなかったが、4.4.2 ではある関数の引数が変わっており、その修正に手作業を要した。しかし複雑な作業はこの衝突ファイルの修正だけに抑えられる。

この手法を用いず、これまでの様に一つのリポジトリのみで行っていた場合には、本家 GCC の新旧の差分をとるか、もしくは本家の旧 GCC と CbC での差分をとり、新しく適用する必要がある。この差分の取得は diff を使って手動で行う必要があるが手順は非常に複雑になり、どこに問題が生じたかも判別しにくくなる。

新しいリポジトリ管理方法では diff を用いた複雑な作業は必要なく、作業は衝突したファイルのみに抑えられる。これによりソースコードアップデートに関するメンテナンス性の向上が実現できた。

# 第6章　結論

## 6.1　まとめ

本研究では Continuation based C(CbC) の GCC ベースコンパイラの改善を行った。
CbC は軽量継続をベースとした言語で、 プログラムの分割による開発効率の向上や、状態遷移という特徴を元にタブロー法を用いたプログラム検証などを目的として設計されている。 これまで、CbC 言語のコンパイラには micro-c をベースとしたものがあり、2008 年の研究にて GCC をベースとしたコンパイラが開発されていた。
本研究ではこの GCC ベースコンパイラを実用的な CbC コンパイラにすることを目標とし、そのために必要な項目のリストアップとその実装を行った。
2 章では目標とする達成項目を洗い出した。この中には C との互換性を向上させる機能や継続制御での制限の除去、また GCC の開発リリースに追従するためのメンテナンス性の向上などが含まれた。
4 章ではこれらの項目の実装を行った。ここでは最初に GCC に移植した際のコードセグメントや継続制御の実装方法を再確認し、その上で今回の実装の手法を提案し、実装した。
5 章では、4 章における改善点の評価とともに、実用的なプログラムが動作可能になったことで、以前のコンパイラと GCC ベースのコンパイラで性能の比較を行った。さらにメンテナンス性向上のための手法の評価も行った。
本研究の成果としては、**a)** 継続制御での引数の順序による制限を除去した。**b)** 様々なアーキテクチャでの CbC プログラムの実行を可能にした。**c)** fastcall 最適化により x86 アーキテクチャでの動作速度が改善された。**d)** micro-c ベースコンパイラと比べて、PowerPC では速度に大きな向上が見られた。**e)** リポジトリ管理方法の改善により、メンテナンス性が向上した。などが挙げられる。

## 6.2　今後の課題

実用的な CbC プログラムが実行可能になった事で、CbC を用いた研究もこれまでにない応用が可能になる。本研究室の提案する Cerium はこれまでは C++を用いて実装されていたが、現在は CbC への移植作業が進行中である。その他、CbC を用いた検証や分散プログラミングなどの研究もこれからの研究課題となる。
また、CbC 言語自体の仕様拡張も検討されている。特にオブジェクト指向は現在のプログラミングの主流であり、CbC でもその実装を行いたい。しかし 1 章でも述べたよう

に CbC の開発動機にはオブジェクト指向の問題点も含まれる。オブジェクティブな CbC の導入には、CbC という言語の特徴を活かしつつ、この問題を回避していく必要がある。

# 謝辞

本研究を進めるに当たって、日頃よりご指導ご助言を賜りました河野真治准教授に心より感謝致します。

　また、お世話になりました宮國渡さん、赤嶺悠太さん、宮城健太さん、並びに研究室の後輩のみなさま、そして直接研究では関わることのなかった先輩方（特に金城拓実さん、下地篤樹さんの研究論文は本研究の上でも大いに参考になりました）にも深く感謝致します。

　最後に、長年に渡り理解を示し、支援してくださった家族に感謝致します。

# 参考文献


[1] 河野真治. “Implementing Continuation based language in GCC” Continuation Festa 2008, April, 2008

[2] 河野真治. “検証を自身で表現できるハードウェア、ソフトウェア記述言語 Continuation based C と、その Cell への応用”. 電子情報通信学会 VLSI 設計技術研究会, March, 2008

[3] 河野真治, 渕田良彦, 宮國渡. “継続を基本とする言語 CbC による分散プログラミング”. 日本ソフトウェア科学会第 23 回大会論文集, Sep, 2006

[4] 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

[5] 河野真治, 池村正之. “状態集合の分割による時相論理検証の並列化”. 電気学会・電子情報通信学会合同講演会, Dec, 1998

[6] 金城拓実. “軽量継続を用いたゲームプログラムの分割と再構成の考察”. 琉球大学理工学研究科情報工学専攻 平成 17 年度学位論文, 2006.

[7] 宮國渡. “Cell 用の Fine-Grain Task Manager の実装”. 琉球大学理工学研究科情報工学専攻 平成 20 年度学位論文, 2009.

[8] 下地篤樹, 河野真治. “タブロー法を用いた Continuation based C プログラムの検証”. 日本ソフトウェア科学会第 23 回大会, 2006.

[9] 下地篤樹, 河野真治. “線形時相論理による Continuation based C プログラムの検証”. 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, 2007.

[10] 神里晃 宮國渡, 杉山千秋, 河野真治. “C から Cell アーキテクチャを利用した CbC への変換” 電子情報通信学会 VLSI 設計技術研究会, March, 2008

[11] 金城拓実, 河野真治. “ゲームプログラムからの一部の仕様の抽出に関する考察”. 日本ソフトウェア科学会第 22 回大会論文集, Sep, 2005

[12] 金城拓実, 河野真治. “ゲームプログラムの シナリオに基づいた状態遷移系を生成するシステムの提案”. 日本ソフトウェア科学会第 21 回大会論文集, Sep, 2004

[13] 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” 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

[14] “Nested Function”. http://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html

# 発表文献

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

与儀健人, 河野真治.

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.

**組み込み向け低レベル言語 CbC の GCC による実装**

与儀健人, 河野真治.

第 6 回ディペンダブルシステムワークショップ, July, 2008

# 付 録A　付録

## A-1　測定環境

5 章の性能評価では CPU アーキテクチャとオペレーティングシステムの 5 つの組み合わせで測定を行った。ここでその 5 つの環境を一覧する。

- x86/OS X

  **機種** Mac mini

  **CPU** 2.26GHz Intel Core 2 Duo

  **メモリ** 2GB 1067MHz DDR3

  **OS** Mac OS X 10.6.2

- x86/Linux

  **機種** 自作

  **CPU** 2.4GHz Intel Core 2 Quad Q6600

  **メモリ** 4GB 800MHz DDR2

  **OS** Gentoo Linux

- PPC/OS X

  **機種** Power Mac G5

  **CPU** 2GHz PowerPC G5

  **メモリ** 2GB DDR

  **OS** Mac OS X 10.5.8

- PPC/Linux

  **機種** PowerBook 17” -1.67GHz

  **CPU** 1.66GHz PowerPC G4 7447A

  **メモリ** 1.5MB DDR

  **OS** Gentoo Linux

- PPC/PS3

  **機種** PlayStation3 モデル CECHB00

  **CPU** Cell Broadband Engine 3.2GHz

  **メモリ** 210MB

  **OS** Fedora release 10

## A-2 __return 擬似変数の実装

環境付き継続の実装のための、__return 擬似変数を追加する処理をコード A.1,A.2 に掲載する。これは gcc/c_parser_postfix_expression 関数内の処理である。

コード A.1: c_parser_postfix_expression での処理

```
case RID_CbC_RET:

  stmt = c_begin_stmt_expr ();

  /* create label declaration.  */
  label = get_identifier ("_cbc_exit0");
  tlab = declare_label (label);
  add_stmt (build_stmt (DECL_EXPR, tlab));

  /* declare retval.  (int retval;) */
  tree decl_cond =
      build_decl(VAR_DECL,get_identifier ("retval"),
                 TREE_TYPE(current_function_decl));
  pushdecl (decl_cond);

  /* define nested function.  */
  decl = cbc_define_nested_code(label, decl_cond);

  /* define if-ed goto label and return statement. */
  cbc_define_if_closed_goto (label, decl_cond);

  /* get pointer to nested function.  */
  value = build_addr (decl , current_function_decl);
  add_stmt (value);

  expr.value = c_finish_stmt_expr (stmt);
```

コード A.2: c_parser_postfix_expression での処理

```
static void
cbc_finish_labeled_goto (location_t loc,
                         tree label, tree retval)
{
  tree tlab;
  tree cond;

  tree cstmt = c_begin_compound_stmt (true);

  tlab = define_label (loc, label);
  gcc_assert (tlab);
  decl_attributes (&tlab, NULL_TREE, 0);
  add_stmt (build_stmt (LABEL_EXPR, tlab));
```

```
  tree ret = c_finish_return (retval);
  TREE_USED(ret) = 1;

  cond = integer_zero_node;
  tree if_body = c_end_compound_stmt (cstmt, true);
  TREE_SIDE_EFFECTS (cstmt) = 1;
  c_finish_if_stmt (loc, cond, if_body, NULL_TREE, false);
}


static tree
cbc_finish_nested_function (location_t loc,
        tree label, tree retval_decl)
{

  tree fnbody;
  tree _retval_decl, _envp_decl;
  struct c_declarator *declarator;
  tree ident;
  struct c_arg_info *args;
  struct c_declspecs *specs;
  struct c_typespec t;
  {
    push_scope ();
    declare_parm_level ();
    /*tree retval_type = TREE_TYPE(retval_decl);*/

    _retval_decl = build_decl (PARM_DECL,
            get_identifier ("_retval"),
            TREE_TYPE (retval_decl));
    DECL_SOURCE_LOCATION (_retval_decl) = loc;
    DECL_ARTIFICIAL (_retval_decl) = 1;
    DECL_ARG_TYPE (_retval_decl) = TREE_TYPE(retval_decl);
    pushdecl (_retval_decl);
    finish_decl (_retval_decl, NULL_TREE, NULL_TREE);

    _envp_decl = build_decl (PARM_DECL,
            get_identifier ("_envp"),
            ptr_type_node );
    DECL_SOURCE_LOCATION (_envp_decl) = loc;
    DECL_ARTIFICIAL (_envp_decl) = 1;
    DECL_ARG_TYPE (_envp_decl) = ptr_type_node;
    pushdecl (_envp_decl);
    finish_decl (_envp_decl, NULL_TREE, NULL_TREE);

    args = get_parm_info(false);
    pop_scope();
  }

  t.kind = ctsk_resword;
  t.spec = get_identifier("void");
  specs = build_null_declspecs();
  declspecs_add_type (specs, t);
  finish_declspecs (specs);

  /* make nested function.  */
  declarator =
      build_id_declarator (
              get_identifier ("_cbc_internal_return"));
  declarator = build_function_declarator (args, declarator);

  c_push_function_context ();

  if (!start_function (specs, declarator, NULL_TREE))
    {
      c_pop_function_context();
      gcc_assert (0);
```

```
    }
  store_parm_decls ();


  /* start compound statement.  */
  tree cstmt = c_begin_compound_stmt (true);

  add_stmt (build_modify_expr
                  (loc, retval_decl,
                   NOP_EXPR, _retval_decl));
  tree stmt = c_finish_goto_label (label);

  /* end compound statement.  */
  fnbody = c_end_compound_stmt (cstmt, true);
  TREE_SIDE_EFFECTS (cstmt) = 1;

  /* finish declaration of nested function.  */
  tree decl = current_function_decl;
  add_stmt (fnbody);
  finish_function ();
  c_pop_function_context ();

  add_stmt (build_stmt (DECL_EXPR, decl));
  return decl;

}
```

## A-3 プロトタイプ生成スクリプト

4.2.5 節で紹介した Python スクリプトをコード A.3 に掲載する。

コード A.3: プロトタイプ生成スクリプト

```
#!/usr/bin/env python3.0

import sys
import re
import getopt

reserved_words = [ "if", "for", "switch", "return", "while", "else", ]

PATTERN =  "([a-zA-Z_][\w\s]*\**)\s([a-zA-Z_]\w*)\s*\(([^{/;]*)\)\s*\{"
PROG = re.compile(PATTERN, re.S)

omit_static=False
add_extern=""

def truncate_comments(data):
        pass

def check_reserved_word(decl):
        """ return true if decl's type and name is not reserved word. """

        if decl["name"] in reserved_words or decl["type"] in reserved_words:
                return False
        return True

def read_decls(file):
        declarators = []

        # open the file and read all lines into a string.
        try:
                fo = open(file, 'r')
```

```
                lines = fo.readlines()
                data = "".join(lines)
                truncate_comments(data)
        except IOError:
                print("cannot read file %s" % file)
                return None

        # find all matched strings.
        # moiter is iterator of MatchObject.
        moiter = PROG.finditer(data)
        for mo in moiter:
                tmp = { "type": mo.group(1),
                                "name": mo.group(2),
                                "parms": mo.group(3),
                                "offset": mo.start() }
                if check_reserved_word(tmp):
                        declarators.append(tmp)

        return declarators

def debug_print(decl):
        for (key,value) in list(decl.items()):
                if isinstance(value, str):
                        decl[key] = value.replace("\n"," ").replace("\t"," ")

        print("Type:\t{0:s}".format(decl["type"]))
        print("Name:\t{0:s}".format(decl["name"]))
        print("Params:\t{0:s}".format(decl["parms"]))
        print("offset:\t{0:d}".format(decl["offset"]))
        print("")

def format_print(decl, file):
        for (key,value) in list(decl.items()):
                if isinstance(value, str):
                        decl[key] = value.replace("\n"," ").replace("\t"," ")

        print("/* defined in file {0:s} at offset {1:d} */".format(file,decl["offset"]))
        print("{3:s}{0:s} {1:s} ({2:s});".format(decl["type"],decl["name"],decl["parms"], add_extern))
        print("")

def getoptions():
        global omit_static, add_extern

        try:
                opts, args = getopt.getopt(sys.argv[1:], 'se', [ 'omit-static', 'add-extern' ])
        except getopt.GetoptError as err:
                print(err)
                usage()
                sys.exit(2)

        for opt,a in opts:
                if opt in ("-s", "--omit-static"):
                        omit_static=True
                elif opt in ("-e", "--add-extern"):
                        add_extern="extern "
                else:
                        print("unhandled option {0}".format(opt))
                        usage()

        return args

def usage():
        print( """\
Usage: {0:s} OPION... [FILE]...
OPTIONS:
        -s, --omit-static       omit static functions
        -e, --add-extern        add extern to all function declarations
        """.format(sys.argv[0]))
```

```
def main():

        # option handling.
        args = getoptions()

        for file in args:
                # read function declaration from each file.
                decls = read_decls(file)
                if decls==None or len(decls)==0:
                        # no function found.
                        print("{0} have no function definition!".format(file))
                        continue

                for decl in decls:
                        if omit_static and 0 <= decl["type"].find("static"):
                                # static function is ignored.
                                continue
                        #debug_print(decl)
                        format_print(decl, file)

main()
```

## A-4 quicksort 例題

5 章での速度、ファイルサイズの性能評価に用いた CbC による quicksort の例題プログラムをコード A.4, A.5 に掲載する。

コード A.4: quicksort_cbc.cbc

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

#define  code __code

typedef void *stack;
typedef struct {
    int size;
    void *interface;
    code  (*ret)(void*, stack) ;
} frame, *framep;

/* quickstart main routine. */
typedef struct {
    int *v;
    int s;
    int e;
} QS_IF ;
typedef code  (*RET)(void*);

#include"quicksort_cbc.h"

/* for check. */
void *mustbefreed;

code  returner(stack sp)
{
    framep fp = (framep)sp;
    sp += fp->size;
    goto fp->ret(fp->interface, sp);
}
```

```
code  quicksort_start(void *arg, stack sp)
{
    QS_IF *recvif = arg;
    int a,b,c,p;
    a = recvif->v[recvif->s];
    b = recvif->v[recvif->e];
    c = recvif->v[(recvif->s+recvif->e)/2];

    //printf("quicksort_start: s=%d,e=%d", recvif->s, recvif->e);
    if (recvif->e <= recvif->s) goto returner(sp);

    if (a < b) {
        if (b < c)
            p = b;
        else if (a < c)
            p = c;
        else
            p = a;
    } else {
        if (a < c)
            p = a;
        else if (b < c)
            p = c;
        else
            p = b;
    }

    goto quicksort_divider (recvif, recvif->s, recvif->e, p, sp);
}
/* main routine end. */

/* divide routine. */
code  quicksort_divider(QS_IF *recvif, int s, int e, int p, stack sp)
{
    goto quicksort_divider_s(recvif, s, e, p, sp);
}
code  quicksort_divider_s(QS_IF *recvif, int s, int e, int p, stack sp)
{
    if (recvif->v[s]<p) {
        goto quicksort_divider_s(recvif, s+1, e, p, sp);
    } else
        goto quicksort_divider_e(recvif, s, e, p, sp);
}
code  quicksort_divider_e(QS_IF *recvif, int s, int e, int p, stack sp)
{
    if (p<recvif->v[e]) {
        e--;
        goto quicksort_divider_e(recvif, s, e, p, sp);
    } else
        goto quicksort_swapper(recvif, s, e, p, sp);
}
code  quicksort_swapper(QS_IF *recvif, int s, int e, int p, stack sp)
{
    if (s<e) {
        int tmp;
        tmp = recvif->v[s];
        recvif->v[s] = recvif->v[e];
        recvif->v[e] = tmp;
        goto quicksort_divider(recvif, s+1, e-1, p, sp);
    } else {
        goto quicksort_treecall(recvif, s, e, sp);
    }
}
/* divide routin end. */


/* recursive call routine. */
```

```
code  quicksort_treecall(QS_IF *recvif, int s, int e, stack sp)
{
    framep fp;
    QS_IF *outif;

    /* interface for first quicksort_start this segment directly jump to.  */
    outif = (sp-=sizeof(QS_IF));
    outif->v = recvif->v;
    outif->s = recvif->s;
    outif->e = e;
    fp = (sp-=sizeof(frame));
    fp->ret = quicksort_start;
    fp->interface = recvif;
    fp->size = sizeof(frame)+sizeof(QS_IF);

    /* recvif is used by second quicksort_start.  */
    recvif->s = e+1;
    goto quicksort_start(outif, sp);
}
/* recursive call routine end. */

#define STACK_SIZE 10240

typedef struct {
    code  (*ret)(void*);
    void *ret_arg;
    stack *sp;
} QS_FINISH;
code
quicksort(int *v, int s, int e,  RET ret, void *arg )
{
    framep fp;
    stack sp0, sp;
    sp0 = mustbefreed = malloc(STACK_SIZE);
    sp = sp0 + STACK_SIZE;
    QS_FINISH *finish_if;
    QS_IF *outif;

    /* interface for quicksort_finish.  */
    finish_if = (sp -= sizeof(QS_FINISH));
    finish_if->ret = ret;
    finish_if->ret_arg = arg;
    finish_if->sp = sp0;

    /* interface for quicksort_start.  */
    outif = (sp -= sizeof(QS_IF));
    outif->v = v;
    outif->s = s;
    outif->e = e;
    /* frame for quicksort_finish.  */
    fp = (sp -= sizeof(frame));
    fp->ret = quicksort_finish;
    fp->interface = finish_if;
    fp->size = sizeof(frame)+sizeof(QS_IF);

    goto quicksort_start(outif, sp);
}
code
quicksort_finish(void *arg, stack sp)
{
    QS_FINISH interface;
    interface = *(QS_FINISH*)arg;
    //assert((void*)interface.sp==(void*)mustbefreed);
    free(interface.sp);
    goto interface.ret(interface.ret_arg);
}
```

コード A.5: quicksort_test.cbc

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<unistd.h>
#define code __code

#include"quicksort_test.h"

extern code quicksort(int *,int,int, code (*)(void*), void*);


void
random_initialize(int *v, int size, int min, int max)
{
    int i;
    int diff = max-min+1;

    for (i=0; i<size; i++) {
        v[i] = min+random()%diff;
    }
    return;
}

static void
print_array(int *v, int size)
{
    int i;
    printf("[");
    for (i=0; i<size; i++) {
        printf("%s%4d", (i%10==0)? "\n  ":" ", v[i]);
    }
    printf(" ]\n");
}

void
starter(int size)
{
    int *target;

    target = (int*)malloc(sizeof(int)*size);
    if (!target) {
        perror("malloc");
        exit(1);
    }

    random_initialize(target, size, 0, 90);

    //print_array(target, size);
    goto quicksort(target, 0, size-1, exit0, (void*)target);

    printf("bad region\n");
}

static int size=100;

int
main(int argc, char **argv)
{
    unsigned int seed=0;
    int opt;

    while ((opt = getopt(argc, argv, "s:n:")) != -1) {
        switch (opt) {
            case 's':
                seed = atoi(optarg);
                break;
```

```
            case 'n':
                size = atoi(optarg);
                break;
            default:
                fprintf(stderr, "Usage: %s [-t times] [-n sizeofarray] [-s seed]\n", argv[0]);
                exit(1);
        }
    }

    srandom(seed);
    starter(size);
    return 0;
}

static int
check_sort(int *v, int size)
{
    int i;
    for (i=0; i<size-1; i++) {
        if (v[i] > v[i+1])
            return 0;
    }
    return 1;
}

code
exit0(void *arg)
{
    int *v = arg;
    int b;
    //print_array(arg, size);
    b = check_sort(arg, size);
    if (b) {
        printf("sorting successful!\n");
        exit(EXIT_SUCCESS);
    } else {
        printf("sorting failure! \n");
        exit(EXIT_FAILURE);
    }
}
```