継続を基本とした言語 CbC の gcc 上の実装

Implementing C with Continuation by gcc modification

河野 真治 †
Shinji KONO

† 琉球大学情報工学科,
Information Engineering, University of the Ryukyus,
PRESTO, Japan Science and Technology Corporation
kono@ie.u-ryukyu.ac.jp

継続を持つ言語 CbC の gcc 上の実装の問題点に付いて考察する。gcc 上の RTL 表現上で CbC を実装することにより、多くのアーキテクチャ上で、CbC を実行することが可能になる。しかし、CbC は、C と異なるスタックフレーム構造を持つため、それを gcc のもともと持つスタックフレーム構造と共存させる必要がある。

1 CwC/CbC とは

CwC [1] は、C に継続を導入した言語であり、CbC (Continuation based C) は、CwC からループ構造と関数呼び出しをなくしたものである。CwC は C の上位言語である。[4]

(fig.2) よりハードウェアに近い C としては、C – – [5] が知られているが、C – – が普通の C の代わりとして用いられるのに対し、CbC は、C を CbC にコンパイルできるという意味で下位言語である。CbC 自体は、それ自体をアプリケーション記述に使うというよりは、動作記述、記述単位として使うべきものである。また、CbC は、実行できるセマンティクスを持っていて、最適化自体を記述できる高級言語となっている。

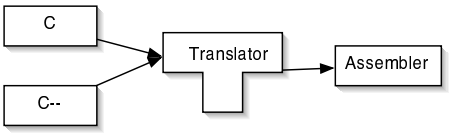

図 1: C - -

例えば、アセンブラの命令の動作そのものを記述することができ、アセンブラ抜きでアセンブラ・プログラミングを行うことができる。(fig.2)

我々は、CbC 用いて、ゲームプログラムの記述 [2] や、状態遷移プログラム、拡張されたハードウェアのソフトウェアの記述 [3] などを行って来た。これらの用途では、CbC を高速に実行することが、もちろん、重要である。

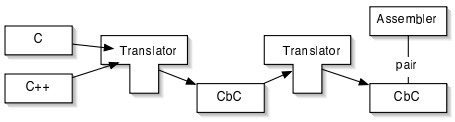

図 2: CbC/CwC

本論文では、GCC 3.0 を変更して、CwC/CbC のコンパイラを作成する方法に付いて考察する。

2 例題

以下の例は、CbC による階乗の計算である。

```
code fact(int n,int result,
       code (*print)()){
    if(n>0){
        result *= n;
        n--;
        goto fact(n,result,print);
    } else
        goto (*print)(result);
}
```

`goto fact(n,result,print);` は、直接の継続であり、その引数は、interface と呼ばれる。属する `code` と同じ interface を持つ `goto` 文は、一つの jump 命令にコンパイルされる。

`goto (*print)(result);` 間接の継続である。しかし、Scheme などの継続と異なり環境を切替えない。これが light weight continuation である。

継続呼び出しの実装は、interface と一時変数の配置変えと、`jump` 命令となる。CwC として関数呼び出しと併用する場合には、スタックポインタを一時変数の上になるように制御する必要がある。

(fig.3)

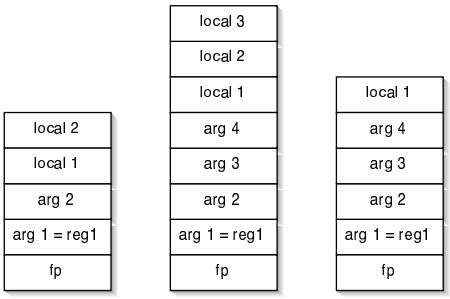

図 3: CbC の stack frame

一時変数と引数は、interface によって `code` 間で伸長する。CbC では引数の一部はレジスタに割り当てることが可能である。フレームポインタが存在する場合は、引数の一番下を指している。この `goto` の間、これは不動である。

これは、通常の C の関数呼び出しとは状況が異なる。この場合は、呼び出しの間に、フレームポインタが移動する。フレームポインタは、引数と一時変数の間を指している。また、そこに、前の呼び出しのフレープポインタの値がセーブされている。C − − の用に、通常の関数に対して、引数付き goto 文を許すようにすると、このフレームポインタの位置を維持しなければならない。

(fig.4)

CbC では、関数と `code` が区別されているため、`code` 間の移動をより効率的に行うことができる。また、これにより、CbC の言語から、スタック的な操作意味論を分離している。これは、C − − にはない特徴である。

3 関数から `code` の呼び出し

CwC の場合は、通常の関数呼び出しの中から、`code` への `goto` を行うことができる。この場合、その関数へ戻る継続 `continuation()` を引数として渡すことができる。この継続は、呼び出された関数内

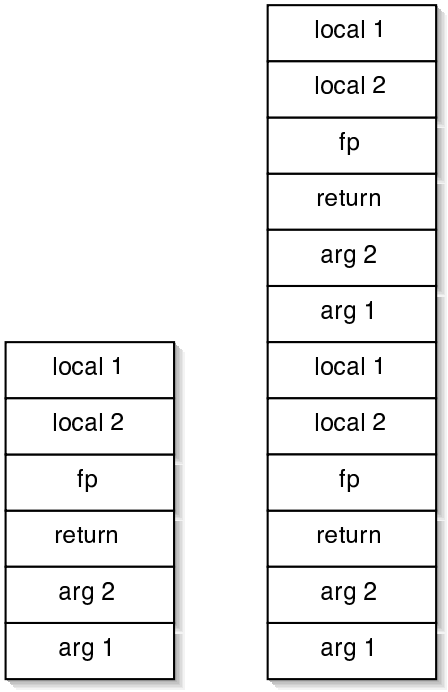

図 4: C の stack frame

での戻値の型を引数として持ち、また、関数呼び出しの環境、つまり、スタックの情報を持っているクロージャーである。

`code` かた関数の呼び出しは、通常の関数呼び出しと同じで良い。この場合は、フレームポインタは移動する。

4 gcc による実装

CwC/CbC は、継続関連の構文を除けば C と同じ構造を持っている。従って、gcc を用いて実装するのが自然だと思われる。ただし、gcc は、既に巨大なソフトであり、実際の変更は容易ではない。

我々は、最初に、Micro-C を用いた実装を行ったが、浮動小数点がないなど、比較的簡単なプログラムしか実行することができなかった。

gcc を用いて、gcc の抽象出力コードである RTL レベルで、CbC を実装できれば、かなりひろい範囲のアーキテクチャで CbC が実行できることになる。

また、CbC の `goto` では、引数の入れ換えが起きることになる。これは、本質的には、並列代入の最適化である。gcc を用いることにより、RTL で代入を記述するだけで、あとは、gcc の最適化が並列代入の最適化を行ってくれると期待できる。(fig.5)

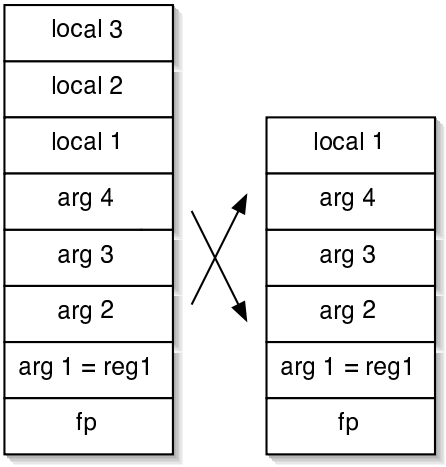

図 5: 引数の入れ換え

構文的に追加する必要があるのは、

```
code 型
引数付き goto 文
continuation 疑似変数
```

だけである。

5 gcc の内部構造

gcc は、

```
yacc (c-parse.y) による構文解析
expand\_expr.c による
RTL 持つ仮想コードの生成
RTL の最適化
目的アセンブラの生成
```

の要素からなる。特に、function.c で関数定義の生成が行われ、call.c で、関数呼び出しが生成される。

CbC での変更は、主に、c-parse.y, expand_expr.c, function.c, call.c で行われる。

6 gcc による実装の問題点

CbC は、code と関数の二つの異なるスタックフレームを持っており、これを共存させなくてはならない。gcc から見ると、CbC のスタックフレームは、局所変数だけから構成されているように見える。

RTL レベルで、CbC を実装するためには、code セグメントは、関数を用いて実装する必要がある。そうしないと、複数のファイルに分割コンパイルすることが出来ない。しかし、スタックフレーム構造は、まったく別である。

このためには、すべての code セグメントは、末尾再帰最適化可能な関数として実装する。code の入口は、末尾再帰の入口となる。gcc の末尾再帰最適化の条件は、かなり厳しいが、強制する必要がある。

code セグメントから、通常の関数呼び出しを行う場合は、セグメント内の局所変数を保護するために、スタックポインタを局所関数の上に設定する必要がある。これは、ぴったりに合わせるのがメモリ効率上は望ましい。しかし、ある程度よりも上にあれば、通常の goto 文でスタックポインタを動かす必要はない。この位置は、code セグメント全体を見て、引数と局所変数全体を調べることにより決めることができる。しかし、それを分割コンパイル前に行うことはできない。

interface が等しい code セグメント間の goto は、jump 1 命令にコンパイルされる。また、一部でも、同じ構造を持っている方が効率の良いコードにコンパイルされる。従って、Interface の同一性を、構造体などで表すことが望ましい。しかし、そのような場合に、gcc は、memory copy サブルーチンを呼び出すことが多い。これを防ぐためには、interface 構造体を、基本要素まで分解してから、RTL 生成を行うのが望ましい。

interface の一部を特定のレジスタに割り振ることが可能である。これは、interface を共有するすべてのコードで同じレジスタを使う必要がある。このためには、関数のプロトタイプにレジスタ宣言が必要となる。

7 CbC on Gcc の使い方

CbC は、直接、アプリケーションを実行するための言語ではなく、中間言語的に使用する。

例えば、CbC にコンパイルすることを前提に新しいプログラミング言語を作成することができる。この時に、CbC の方ではスタックを持たないので、末尾再帰などの処理が可能である。

逆に、CbC は、スタックのハードウェアサポート、例えば、IA32 の push, pop を使用しない。CbC の特定のレジスタをハードウェア・スタック・ポインタに指定することも可能であるが、特殊な命令をこの gcc compiler 版が生成することはない。そのためには、CbC に変換した結果を別に変換する手法が必要である。

8 まとめ

ここでは、継続を基本とした言語を gcc を使ってコンパイルする方法について考察した。直接、コンパイルして実行する手法は、CbC の使い方の一部であるが、重要である。また、CbC の設計は C に合わせてあるので、比較的簡単に gcc に乗せることができるようになっている。

参考文献

[1] 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, September, 2000

[2] 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

[3] 河野 真治, 佐渡山 陽, Continuation Based C による Tchnology Mapping のサポート,FIT 2002, September,2002

[4] 言語の Continuation based C への変換, 河野真治 (琉球大/科学技術振興事業団), 揚 挺 (琉球大), SWoPP 2001, Okinawa, July , 2001

[5] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.