C言語のContinuation based Cへの変換

Translation from C to Continuation based C

河野 真治　　揚 挺
Shinji Kono E-Mail: kono@ie.u-ryukyu.ac.jp You Tei

Information Engineering, University of the Ryukyus,
PRESTO, Japan Science and Technology Corporation

Abstract

継続を持つ言語 Ccontinuation based C は、C の下位言語として設計されている。スタックベースの関数呼び出しを持つ C を、継続のみをもつ言語である CbC に変換する手法に付いて考察する。本変換では明示的にスタックを使うので、変換は容易であるが、効率的な変換を行なうためには、さまざまな最適化が可能である。

1 Continuation based C

Continuation based C (以下 CbC) は、C からループ制御構造と、サブルーチン・コールを取り除き、継続を導入した言語である。 [3] この言語は、C に継続専用のコード単位 (code) と、継続 (goto) を導入した言語のサブセットとなっている。継続 [1] とは、次に実行すべきコードの部分を直接/間接に指定する手法である。

この言語は、サブルーチンよりも小さいプログラム単位であるコードを、コンピュータの助けを借りながらプログラムしていく。これにより、C よりも細かく、アセンブラよりも高度な記述を可能にすることを目標としている。

例えば、code は、コンパイラの内部表現である基本ブロック (サブルーチンコールや分岐の間にある命令のブロック) に相当する単位であり、基本ブロックレベルの最適化は、CbC によりすべて記述することが可能である。

この言語の一つの目的は、状態遷移記述と相性の良い高級プログラミング言語を提供することである。モデル検査や、タブロー法による検証など、状態遷移記述をサポートするシステムは多数存在する。したがって、それと相性の良い記述言語を提供することが重要であると考えられる。

一方で、多くのプログラムが、スタックマシンベースの高級言語によって記述されている。これらを、解析し、再利用や改良することも重要である。CwC は、C を完全に含むが、C の機能を使った部分は、状態遷移解析には向かない。

そこで、本論文では、C から、CbC へのコンパイルを考察する。このコンパイルは、スタックを明示したものであり、関数型言語で行なわれている CPS 変換とは若干異なる。

2 CbC ってなに?

CbC は、C の関数定義とは異なる code という単位を持っている。C- - [2] などでは、継続の対象は通常の関数定義であるが、CbC は、それを区別している。通常の関数呼び出しでは、

呼び出し側 (`caller`) の処理
(引数の積み上げ、戻り番地のセーブ)
呼び出され側 (`callee`) の処理
(局所変数の確保、スタック、
フレームポインタの移動)

となっており、後者は、関数定義に置かれることになる。これらは、継続の呼び出しの場合は、無駄になってしまう。そこで、CwC/CbC では、継続専用の単位を提供することにより、効率的な継続専用のフレーム構造を使用することを可能にした。

Scheme や、C++ 、Java 、あるいは、C も、大域脱出という形で、継続を導入している。これらの言語では、継続は、必ず環境 (入れ子になった局所変数を格納するスタック) のセーブを伴う。CbC では、関数呼び出しが存在しないために、この環境は存在しない。code 内部での局所変数は存在するが、それのネストは起こらない。そこで、環境抜きの継続を使用することが出来る。これは、基本的には、引数付の (直接/間接)goto 文である。本論文では、これを light weight continuation と称する。ただし、CwC は、C の関数呼び出しを持っているので、環境が存在し、環境を伴う継続 (normal continuation) も存在する。

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.1)

3 CbC に無いもの

CbC は、C をかなり制限したもので、機械依存性の無いアセンブラのような構成になっている。CbC は、通常の高級言語が持つような、

ループを含む制御構造

名前管理

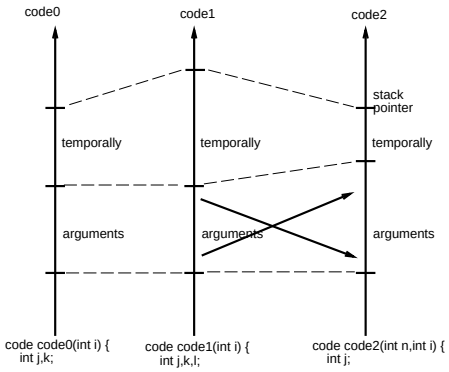

Fig1 継続呼び出し

メモリ管理

を持たない。これらは、CbC の上位の言語によって管理される。

したがって、CbC は、必ず有限時間内で、code の外に出るという特徴がある。この code の外に出るタイミングで、外部イベントの polling や、GC などのチェックを行なうこともできる。

名前管理が存在しないので、code 間の接続は、OS や Linker が行なう。再帰呼び出しを構成する場合でも、実際の行き先を実行時に変更するようなこともありえる。再帰呼び出しは、`loop` という構文を使用する。

また、隣接する code 間での `goto` は、jump 命令さえも必要がない。このような隣接する code の連続は、一つの大きな code を構成する。この場合は、`next` という構文を使用する。

4 CbC の利点

CbC の code は、コンパイラの基本ブロックに相当する。これにより、以下のような利点がある。

4.1 基本ブロックレベルの最適化を、portable な形で行なうことが可能

コンパイラの最適化は、さまざまなレベルで行なわれるが、分岐と分岐の間の実行単位である基本ブロックが最適化の基本である。例えば、

```
while(i>0) {
    j++;
}
```

この while 文をそのままコンパイルすると、

```
.L2:
        cmpl $0,i
```

```
        jle .L3
        incl j
        jmp .L2
.L3:
```

のような形にコンパイルされる。これを以下のような形に変更するのがコンパイラの最適化の定番である。

```
        cmpl $0,i
        jle .L3
.L4:
        incl j
        cmpl $0,i
        jg .L4
.L3:
```

基本ブロックである j++ の移動により、jump 命令が一つ減っていることがわかる。このような最適化は、通常は、C 言語レベルでは制御できない。

CbC では、

```
code while_1(i,j) {
    if (i>0) {
        j++;
        loop;
  /* same as goto while_1(i,j); */
    }
    /* fall thru while_3 */
}
code while_3( ...
```

と、

```
code while_1(i,j) {
    if (! i>0)
        goto while_3(i,j);
    /* fall thru while_2 */
}
code while_2(i,j) {
    j++;
    if (i>0)
        loop;
  /* same as goto while_2(i,j); */
    /* fall thru while_3 */
}
code while_3( ...
```

という形で区別することができる。この変換は、CbC のレベルで portable である。もちろん、個々のcode の実行速度は、アーキテクチャに依存しているので、後者のcode が常に前者よりも速いと言うことは保証されてはいない。

4.2 code 自体は、関数的

code 自体がループを持たないので、(loop は、自分自身のcode への移動である) code の操作的意味論は、関数的である。つまり、

入力 interface と実行前の大域変数の値
出力 interface と実行後の大域変数の値

の組によりcode の意味が決まる。

4.3 Call Semantics を自分で記述することができる

関数呼び出しに相当する部分は、CbC では、スタックを明示して実現する。スタックを明示的に制御することにより、より効果的な最適化が可能となる場合がある。例えば、実行途中で、スタック上のデータを大域変数に移す様な記述を行なうことができる。

4.4 状態遷移系との相性が良い

code に着目し、引数を省略して考えると、CbC の記述は状態遷移記述となる。間接継続などがあるので、単純な有限状態遷移系まで落すのは単純ではないが、状態遷移系に対して使用可能なさまざまな技術との相性が良いと考えられる。

4.5 高度なアセンブラとしても使用可能

スタックの制御ができるので、コンパイラターゲットとして使いやすい。また、アセンブラを特定の形のcode を使用するという形で実装することもできる。

5 C から CbC に変換する

CbC は、C の下位言語として使うことを意図している。C から CbC への変換を定義できれば、C の下位言語ということができる。C を CbC に変換することにより、以下のことが期待できる。

既存のプログラムをそのまま使用可能
コードの単位が細かくなるために、共有できる範囲が広がる

ただし、今回の変換手法は、スタックを明示的に使用するので、データセットに関しては、元の C プログラムと同じである。末尾再帰を含む最適化は、CbC に変換した後、CbC のプログラム変換として行なうことができる。この最適化は、基本ブロックに関する最適化と本質的な差は

ない。

6　変換の例題

まず、簡単な変換の例を考察する。

```
f(int i) {
    int k,j;
    k = 3+i;
    j = g(i+3);
    return k+4+j;
}

g(int i) {
    return i+4;
}
```

この例では、関数 f が関数 g を呼び出している。関数 f には、局所変数 k,j が存在する。

C のサブルーチン・コールは、 一時変数のスタックへの保持、戻り番地の管理、フレーム・ポインタの管理、を行なっている。これらは、比較的複雑な処理であり、CPU によっては、専用のレジスタ、専用の命令を持っている場合がある。(fig.2)

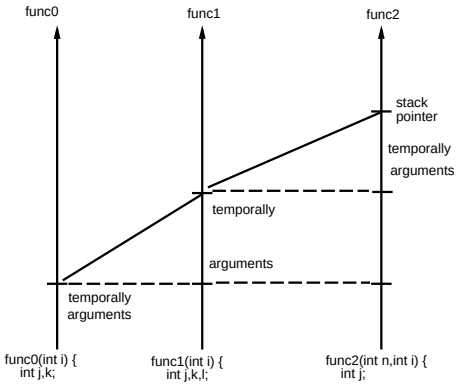

Fig2　C Stack layout

手順としては、呼び出し側 (caller) は、
　引数の計算とスタックへの積み込み
　局所変数のスタックへの退避
　call 命令の実行
　戻り番地の退避と jump
呼び出され側 (callee) では、
　フレームポインタの退避
　フレームポインタの移動
　スタックポインタの変更による
　局所変数の確保
という手順になる。呼び出し側から戻る時には、
　フレームポインタの修正
　スタックの修正
　退避した番地への jump
という手順になる。関数呼び出しのたびに、これらの操作が繰り返される。フレームポインタは必須ではないが、フレームポインタあると、コンパイルされた結果が若干簡単になる。

7　変換例

ここで、int k の値を保存する必要があることに注意する。i を保存する必要は無い。これらの値を退避するためのスタックを定義する。スタックは、さまざまなデータの退避が行なわれるので、char *として定義する。

```
typedef char *stack;

struct cont_save {
   /* General Return
    Continuation */
    code (*ret)();
};
```

関数呼び出しに関係する退避には、必ず戻り番地を含む。これは、CbC では、code へのポインタである。この構造体は、他の退避データ構造体に継承される。関数 f では、局所変数 i,k,j を保存するので、それを含む構造体を用意する。

```
struct f_g0_save {
/* Specialized Return Continuation */
    code (*ret)();
    int i,k,j;
};
```

この退避データを使った C の return に相当する CbC のコードは、stack の先頭に対しての間接継承の呼び出しとなる。

```
code g(int i,stack sp) {
    goto (* ((struct
     cont_save *)sp)->ret)
       (i+4,sp);
}
```

g では局所変数がないので、スタックに触れ

る必要はない。呼び出し側 f は、関数呼び出し前と後で二つに分割される。

```
code f(int i,char *sp) {
    int k,j;
    struct f_g0_save *c;

    k = 3+i;

    sp -= sizeof(
       struct f_g0_save);
    c = sp;
    c->k = k;
    c->i = i;
    c->j = j;
    c->ret = f_g1;
    goto g(i,sp);
}
```

スタックの確保と、そこへの値の確保が、明示的に行なわれている。継続を間接的に指定して、g へ jump する。

g は、分割された後半部分f_g1に jump する。

```
code f_g1(int j,stack sp) {
     /* Continuation  */
    int k;
    struct f_g0_save *c;

    c = sp;
    k = c->k;
    sp += sizeof(struct
       f_g0_save);
    goto (* ((struct
      cont_save *)sp)->ret)
       (k+4+j,sp);
}
```

分割部分では、スタックからデータを復帰すると共に、スタックを元に戻す。

8 変換されたコードのコンパイル結果

変換されたコードに置いて code 側の呼び出し形式は、通常の C とは異なる形式のフレーム構造を持つ。

インタフェースが同一な継続の場合には、goto 文は単一の jump 命令となるが、インタフェースが異なる場合は、レジスタの並べ換えが必要となる。継続と関数呼び出しを両方持つ CwC の場合には、明示されたスタックと、明示されないスタックの二重構造を持つことになる。CwC では、code の局所変数の量に合わせて明示されなスタックの増減を行なう必要がある。

したがって、生成されたコードを CwC でコンパイルする場合には、元の C のコードよりも明示的なスタックが増える分だけ大きくなる。

生成されたコードで、暗黙のシステムスタックを明示されたスタックとして使うことも可能だと思われる。この場合は、code の局所変数を同じスタック上にとるのか、別にとるのかという選択肢が生じる。局所変数を明示的なスタック上にとることも可能である。どのような方法をとるかは、実装と設計によると思われる。

この変換は、CPU に依存したサブルーチンの呼び出しを使用しない。したがって、jump 命令とレジスタセーブよりも、高速なサブルーチン呼び出しを持つ CPU の場合は、生成されるcodeが元の C よりも速度的に不利になる場合がある。例えば、レジスタウィンドが有効に働いている SPARC のような場合、リンクレジスタを使ったサブルーチン呼び出しを行なう PowerPC などである。

ただし、フレームポインタは存在しなくなるので、その分、効率が良くなる場合もある。

9 変換アルゴリズム

変換は、コンパイラの基本ブロックを抜き出すものと、ほとんど同じである。まず、C のプログラムを、ループ、および、関数呼び出し部分で分割する。

次に、使用されている中間変数を抜き出す。この時に、式の途中のサブルーチンコールなどでは、必要な中間変数を追加する必要がある。

```
a = g(i)+a;
```

は、

```
a' = g(i);
a = a'+a;
```

のように中間変数 a' が追加する。

サブルーチン呼び出しを、

- レジスタのスタックへのセーブ
- 戻り先 code のスタックへのセーブ
- サブルーチン code への goto

と、

戻り先 code でのレジスタのスタックからの復帰を追加する。最後に、return を、スタックにセーブされた戻り先code へのgoto へ置き換える

10 最適化に関して

CbC では、インタフェースが異なる継続の呼び出しでは、レジスタの入れ換えが必要となる。したがって、インタフェースをできるだけ共通な形で変換する必要がある。インタフェースのいくつかの変数はレジスタに割り当てられ、いくつかはメモリ上に取られる。

連続したインタフェースの等しい code へのgoto は、jump 命令が不要である。したがって、code の配置の工夫が有効である。ただし、jump 命令一つ自体よりも、code の共有の効果の方が大きい場合もある。

スタックを用いた間接継続を使用して、変換されたコードがもっとも汎用的である。しかし、呼び出される関数のコピーを作れば、スタックを使用しなくても、継続にコンパイルできる場合がある。

```
code f(int i,char *sp) {
    int k,j;
    k = 3+i;
    goto g_f0(i,k,j,sp);
}
code g_f0(int i,int k, int j,
      stack sp) {
    i=i+4;
    goto f_g1(i,k,j,sp);
}
code f_g1(int i,int k, int j,
      stack sp) {
    goto (* ((struct
      cont_save *)sp)->ret)
       (k+4+j,sp);
}
```

コピーされた g_f0 は、呼び出し毎に生成される。これを、スタックを使わない間接呼び出しを用いて、再度、共有することも可能である。ただし、インタフェースは、そろえる必要がある。

```
code g_f1(int i,int k, int j,
      code (*cont)(),stack sp) {
    i=i+4;
    goto (*cont)(i,k,j,sp);
}
```

このような共有は、似ているcode に対して、積極的に行なうことができる。これは、通常の C コンパイラでは記述不可能なコード領域の最適化である。

末尾再帰の最適化は、この変換では自動的には行なわれない。関数呼び出しにより、分割された後半部分で、スタックに退避した変数を使用しない場合に、スタックへの退避を省略することにより、末尾再帰の最適化が行なわれる。このような最適化自体を記述することができるのが CbC の特徴であるので、この最適化は、CbC 自体のプログラム変換として記述することが望ましい。

11 まとめ

スタックを用いた関数呼び出し機構を持つ C 言語から、継続のみを持つ言語 CbC への変換手法について考察した。

変換自体は単純だが、変換プログラムは、使用する中間変数を数え上げる必要があるので、コンパイラの基本ブロックへの変換と同程度の複雑なプログラムとなる。したがって、sed や awk などで簡単に記述できる変換とはならない。

この変換では、C 言語をそのまま変換してしまうので、実行速度や、その他のメリットがない。CbC は、より細かいレベルでの最適化をアセンブラのレベルよりも上で記述することが目的なので、C 言語そのもの操作的意味を表した CbC 記述から始めて、より効率の良い、応答性の良い CbC のコードへ変換する手法を提供する必要がある。

References

[1] Carl Hewitt. Control Structure as Patterns of Passing messages. In *Artificial Intelligence: An MIT Perspective*, Vol. 2, pp. 435,465. 1979.

[2] 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.

[3] 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.