継続を持つCの下位言語によるシステム記述

System Description in C with Continuation

河野 真治

Shinji Kono E-Mail: kono@ie.u-ryukyu.ac.jp *

継続を持つ C の下位言語のコンパイラ C with Continuation を作成した。それを用いたシステム記述の利点と欠点に付いて考察する。この言語は、サブルーチンよりも細かいプログラミング共有単位を持ち、実行時 Working set を小さくすることが可能である。また、既存の C 言語を、コンパイルすることも可能である。

1 ハードウェア記述、ソフトウェア記述のギャップ

オブジェクト指向技術と、それに基づく言語 Java 等が注目されているが、これらの言語は、動的な適合性を中心に設計されたものである。リフレクション等の動的変更技術と同様に、C などの低レベルな言語による記述に比べて、余分な条件判断 (Method search や Meta level での実行など) を増やしてしまう。このような言語は、コンパクトで高速な応答を期待される Real-time 処理や組み込み用途には適さない。

実行時の可変性に乏しい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、長年進歩していない。しかし、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、複雑なマルチメディア関係の命令などを持つ CISC 的な CPU が増えて来ている。そのために既存の言語に対するコンパイラを一々設計しなおすことが必要になって来ている。(図 1)。

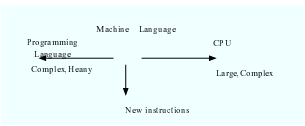

図 1: Towards Different Direction

そこで、本論文では、このギャップを埋めるべく、

- CwC: C with Continuation (Super Set of C) C に軽量継続を導入したもの
- CoC: C on Continuation (Subset of C) C の下位言語 (ループ構造、関数呼出しを持たない)

を導入する。これを新しいプログラミング単位として使う。

この言語 C with Continuation は、最初の要求仕様とは異なり、C の上位互換で作られている。C に継続と、継続用の code と呼ばれる呼び出し単位を導入したものである。通常の C の関数呼び出しを使わなければ、C の下位互換となる。

コンパイラは C で記述されており、太田昌孝氏の Micro C を拡張する形で実装されている。しかし、現在は、C の言語を、制限された C with Continuation にコンパイルするツールは、まだ作成していないので、 制限された形でのセルフコンパイルは実現していない。Micro C は、ANSI-C 対応、構造体のコピーなどの拡張は行なっているが、現在は浮動小数点には対応していない。

現在のターゲットは、i386 および、MIPS R3000 である。

継続 (Continuation) のアイデアは古く [1] に遡る。その後、MIT の Scheme [2] で call-with-continuation により、環境のイメージをそのまま引き渡す継続が実装されている。これは、C の setjump、C++ の catch, throw, Java の try などに相当するものである。C++ では、RWCP の MPC++[3] などが継続をサポートしている。

しかし、Baker, Hewitt らの継続のアイデアは、それよりもシンプルで効果的なものであったことはあまり知られてはいない。この C with Continuation では、Baker, Hewitt 流のよりシンプルな継続を実装している。

また、この言語は、仮想機械語ではない。固定されたレジスタや、決まった手続き呼出しや、フレームポインタなどの仕組みを持たない。しかし、この

*Information Engineering, University of the Ryukyus,

言語を特定の機械にコンパイルすることは難しくない。つまり、機械に依存しないアセンブラとして使えるように設計されている。

1.1 プログラム例

```
typedef struct fact_interface {
  int n;int result;int orig;
  code (*print)(struct fact_interface);
  code_with_return exit1(int);
} fact_interface;

code factorial(fact_interface args)
{
    if (args.n<0) {
        printf("err %d!\n",args.n);
        goto (*args.exit1)(0);
    }
    if (n==0) {
        goto (*print)(args);
    } else {
        args.result *= args.n;
        args.n--;
        goto factorial(args);
    }
}

int main( int ac, char *av[])
{
    int n = atoi(av[1]);
    fact_interface args =
       {n,1,n,print,return};
    goto factorial(args);
}

code print(fact_interface args)
{
    printf("%d! = %d\n",
       args.orig, args.result);
    goto (*args.exit1)(1);
}
```

最初の構造体 fact_interface の定義は、階乗を継続を使って計算するこのプログラムのインタフェースの定義である。基本的に状態遷移中心に記述されるこの言語では、状態はインタフェースで統一されて渡される。

この言語では、code と code_with_return という二つの組み込み型が C に追加されている。インターフェースでは、この状態遷移から抜け出る先を決める継続を二つ定義している。

code (図 2) は、C with Continuation の実行単位 (状態遷移) であり、以下の三つの要素からなる。実行文を省略すると、code のプロトタイプ宣言となる。

1. 入口の名前とインターフェース
2. 実行文
3. 出口 goto 文

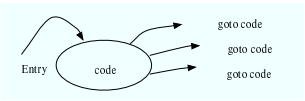

図 2: A Code in C with Continuation

code_with_return は、C との互換性のための型であり、C の return 文に相当する継続である。この継続は、Scheme の continuation と同じであり、値を返すことができる。制限された C with Continuation の場合は、これは、C の関数の環境と、return 文の機能を実行する code を持つ構造体である。code_with_return は返す値の型を指定する必要がある。C との互換性があり、code 内部から自由に C の関数を呼び出すことができる。

goto 文は、C のラベルへの goto の機能も持つが、他の code への状態遷移である。間接参照することにより、引数として持っている継続へ移り、現在の状態遷移系から脱出することができる。code_with_return への goto の場合は、code_with_return を生成する組み込み関数である return が呼び出された関数と、その環境からの return を実行する。

インターフェースは通常の構造体である。しかし、一部はレジスタにマップされる。残りはスタック上に取られる。状態遷移系から脱出するための継続の間接参照が最低一つあるのが普通である。

直接呼出しする goto 文と、間接呼出しする goto 文により、code は自然な形でグループ化される。直接呼出しする code 群は、強連結肢集合であり、閉じた状態遷移を構成する。これは、オブジェクトと似ているが、異なるものである。

2 goto 文の特徴

code からの脱出は、基本的に goto そのものであるので、余計なキーワードを増やさないためにも、goto をそのまま使用した。コンパイラは、goto の引数の型を見て、適切なコード生成を行なう。goto 文には以下の 4 種類があることになる。

1. goto code_name() 直接の名前呼び出し

2. `goto (*code_pointer)()` 間接の呼び出し (= 継続への goto)
3. `goto (*code_with_return)` C の関数への return
4. `goto label` 従来の `goto` 文

複数の継続を持ち歩くことにより大域脱出として使うことができる。状態遷移のみでプログラムする場合は大域脱出に環境は関係ないので、この場合の継続は、Scheme の継続とは異なり環境全体を指すことはない。また、継続を返す特別な `call-with-continuation` も必要ない。

しかし、C との互換性を持つためには、`return` を使って、環境込の継続を作る必要がある。これは、Scheme の継続と同等のものである。

goto 文を用いて code に戻るというこの動きは、再帰的実行ではなくループであるしたがって、以下のような特徴がある。

- 無駄なスタックの操作が無くなる
- フレームポインタの処理の必要なし
- 全体のワーキングセットを小さくすることができる
- 死んで放置されるスタック上のデータがない
- code の共有の単位が関数よりも小さい→より共有できる可能性がある

2.1 従来の goto 文との違い

従来 `goto` 文は以下のような理由で、良くないとされてきた。

- 制御の流れを分かりにくくする (スパゲティ化)
- プログラムが構造化されない
- `goto` の飛び先での状況が複雑

今までの goto 文はラベルの場所から命令文の途中に割り込む形なので、このようなことが起きてしまう。

今回使われている
goto 文は行き先が必ず `code` の先頭である。したがって、以下のような特徴がある。

- 状態遷移という形で制御が常に明確
- 飛び先での状況は入力変数として確定している
- code のグループ化によりプログラムを構造化する
- 大域脱出の際に環境を持ち歩く必要がない

つまり従来の `goto` 文の欠点は、C with Continuation には引き継がれない。

しかし、状態遷移そのものを分かりにくく記述することは可能であるので、C with Continuation で書けば、必ずプログラムがわかりやすくなると言うことではない。

有限状態遷移系は、良く研究されている分野であり、モデル検証や、同等性判定などの手法を C with Continuation に使うことができることも、この言語の利点の一つである。

3 状態遷移記述に適した言語

C with Continuation 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。

ゲームや User Interface などでは状態遷移記述が多用されている。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

などが使われており、記述性が低く、効率が悪い。

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とみなすことはできない。

巨大な switch 文は、コンパイルも難しく適切な最適化を行なうことができない。また、人間が読む場合に読みやすいとはいえない。

C with Continuation を状態遷移記述言語 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 (インピーダンスマッチ) の向上をはかることができる。

3.1 生成されるコード

入口と出口のインタフェースが一致している場合は、レジスタのマッピングや、メモリ上のデータの移動が必要ないので、`goto` 文はただ一つのジャンプ命令にコンパイルされる (図 3)。

RISC では、実際上、インタフェースのすべての値がレジスタ上に載ると考えられる。i386 の少ないレジスタの場合でも、インタフェースの良く使う変数の配置を工夫することによりレジスタ上での演算に

図 3: Compile Result

```
#       result *= n;
        movl %esi,%eax
        imull %eax,%edi
        movl %edi,%ecx
#       n--;
        movl %esi,%ecx
        addl $-1,%esi
#       goto factorial(....
        jmp factorial
```

コンパイルすることが可能である。また、レジスタに割り当てられなかった場合でも、決まったアドレスが使われるので、仮想レジスタなどの技術をターゲット CPU が持っていれば、有効に働くと考えられる。

また、ほとんどの演算がレジスタ同士で行なわれるので、code 内の演算は、直接にアセンブラ命令と対応することになる。MMX などの特殊な命令を使う場合は、パターンを使ったピープホール最適化が適している。

4 まとめ

C with Continuation は、状態遷移記述に適した C 互換の言語である。継続中心に動作する言語を設計し、実装を行なった。今後は、GCC を使った本格的な実装をめざすとともに、C の C with Continuation へのコンパイラや、自動検証系やプログラム変換系などの設計と実装を行なう予定である。

参考文献

[1] Carl Hewitt and Jr. Henry Baker. Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, Dec. 1977.

[2] Jonathan Rees and WilliamClinger . The revised 4 report on the algorithmic language Scheme. In *Lisp Pointers*, 1991.

[3] Y. Ishikawa. Parallel Programming in MPC++ Version 2. In *Future Directions for Parallel C++*, June 1997.