xv6の構成要素の継続の分析

清水 隆博[1,a)]　河野 真治[2,b)]

概要：OS 自体そのものは高い信頼性が求められるが、 OS を構成するすべての処理をテストするのは困難である。テストを利用して信頼性を高めるのではなく、 OS の状態を状態遷移を基本としたモデルに変換し形式手法を用いて信頼性を高めたい。
状態遷移単位での記述に適した言語である CbC を用いて、小さな unix である xv6 kernel の書き換えを行っている。このためには現状の xv6 kernel の処理がどのような状態遷移を行うのかを分析し、継続ベースでのプログラミングに変換していく必要がある。本稿では xv6kernel の構成要素の一部に着目し、状態遷移系の分析と状態遷移系を元に継続ベースで xv6 の再実装を行う。

1. OS の信頼性

様々なアプリケーションは OS の上で動作するのが当たり前になってきた。アプリケーションの信頼性を向上させるのはもとより、 土台となる OS 自体の信頼性は高く保証されていなければならない。OS そのものも巨大なプログラムであるため、 テストコードを用いた方法で信頼性を確保する事が可能である。しかし並列並行処理などに起因する動かしてみないと発見できないバグなどが存在するため、 テストで完全にバグを発見するのは困難である。また、OS を構成する処理も巨大であるため、 これら全てをテスト仕切るのも困難である。テスト以外の方法で OS の信頼性を高めたい。

数学的な背景に基づく形式手法を用いて OS の信頼性を向上させることを検討する。OS を構成する要素をモデル検査してデッドロックなどを検知する方法や、 定理証明支援系を利用した証明ベースでの信頼性の確保などの手法が考えられる。形式手法で信頼性を確保するには、 まず OS の処理を証明などがしやすい形に変換して実装し直す必要がある。これに適した形として、状態遷移モデルが挙げられる。OS の内部処理の状態を明確にし、 状態遷移モデルに落とし込むことでモデル検査などを通して信頼性を向上させたい。既存の OS はそのままに処理を状態遷移モデルに落とし込む為には、 まず既存の OS の処理中の状態遷移を分析し、仕様記述言語などによる再実装が必要となる。しかし仕様記述言語や定理証明支援系では、 実際に動く OS と検証用の実装が別の物となってしまうために、 C 言語などでの実装の段階で発生するバグを取り除くことができない。実装のソースコードと検証用のソースコードは近いセマンティクスでプログラミングする必要がある。

さらに本来行いたい処理の他に、メモリ管理やスレッド、CPU などの資源管理も行う必要がある。本来計算機で実行したい計算に必要な計算をメタ計算と呼び、 意図して行いたい処理をノーマルレベルの計算と呼ぶ。ノーマルレベル上での問題点をメタ計算上で発見し信頼性を向上させたい。プログラマからはノーマルレベルの計算のみ実装するが、整合性の確認や拡張を行う際にノーマルレベルと同様の記述力でメタ計算も実装できる必要がある。

ノーマルレベルの計算とメタ計算の両方の実装に適した言語として Continuation Based C(CbC) がある。CbC は C と互換性のある C の下位言語であり、 状態遷移をベースとした記述に適したプログラミング言語である。C との互換性のために、 CbC のプログラムをコンパイルすることで動作可能なバイナリに変換が可能である。また CbC の基本文法は簡潔であるため、 Agda などの定理証明支援系との相互変換や、 CbC 自体でのモデル検査が可能であると考えられる。すなわち CbC を用いて状態遷移を基本とした単位でプログラミングをすると、形式手法で証明が可能かつ実際に動作するコードを記述できる。

現在小さな unix である xv6 kernel を CbC を用いて再実装している。再実装の為には、 既存の xv6 kernel の処理の状態遷移を分析し、継続を用いたプログラムに変換していく必要がある。本論文ではこの書き換えに伴って得られた xv6 kernel の継続を分析し、 現在の CbC による書き換えについて述べる。

1　琉球大学大学院理工学研究科情報工学専攻
2　琉球大学工学部工学科知能情報コース
a)　anatofuz@cr.ie.u-ryukyu.ac.jp
b)　kono@ie.u-ryukyu.ac.jp

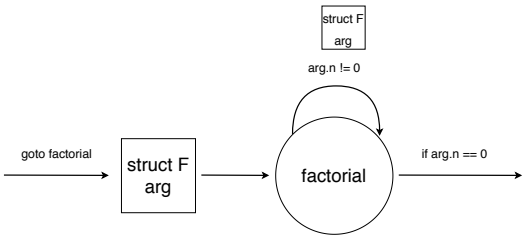

図 1: CbC で階乗を求める例題の状態遷移

2. Continuation Based C

Continuation Based C(CbC) とは C 言語の下位言語であり、関数呼び出しではなく継続を導入したプログラミング言語である。CbC では通常の関数呼び出しの他に、関数呼び出し時のスタックの操作を行わず、次のコードブロックに `jmp` 命令で移動する継続が導入されている。この継続は Scheme などの環境を持つ継続とは異なり、スタックを持たず環境を保存しない継続である為に軽量である事から軽量継続と呼べる。また CbC ではこの軽量継続を用いた再帰呼び出しを利用することで `for` 文などのループ文を廃し、関数型プログラミングに近いスタイルでプログラミングが可能となる。現在 CbC は GCC 及び LLVM/clang 上にそれぞれ実装されている。

CbC では関数の代わりに CodeGear という単位でプログラミングを行う。CodeGear は通常の C の関数宣言の返り値の型の代わりに`__code` で宣言を行う。

CbC で階乗を求める例題を Code 1 に示す。例題では CodeGear として `factorial` を宣言している。`factorial` は CodeGear の引数として `struct F` 型の変数 `arg` を受け取り、`arg` のメンバー変数によって `factorial` の再帰呼び出しを行う。`arg` の様な CodeGear の引数のことを `DataGear` と呼ぶ。CodeGear の呼び出しは `goto` 文によって行われる。この例題を状態遷移図にしたものを図 1 に示す。図中の四角が DataGear、円が CodeGear に対応する。

```
__code factorial(struct F arg) {
    if (arg.n<0) {
        exit(1);
    }
    if (arg.n==0) {
      goto arg.next(arg);
    } else {
      arg.r *= arg.n;
      arg.n--;
      goto factorial(arg);
    }
}
```

Code 1: CbC で階乗を求める例題

CodeGear は関数呼び出し時のスタックを持たない為、一度ある CodeGear に遷移してしまうと元の処理に戻ってくることができない。しかし CodeGear を呼び出す直前のスタックは保存されるため、部分的に CbC を適用する場合は CodeGear を呼び出す `void` 型などの関数を経由することで呼び出しが可能となる。

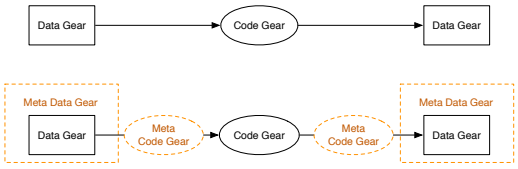

図 2: CodeGear と MetaCodeGear

この他に CbC から C へ復帰する為の API として、環境付き goto という機能がある。これは GCC では内部コードを生成、LLVM/clang では `setjmp` と `longjmp` を使うことで CodeGear の次の継続対象として呼び出し元の関数を設定することが可能となる。したがってプログラマから見ると、通常の C の関数呼び出しの返り値を CodeGear から取得する事が可能となる。

3. CbC を用いた OS の実装

軽量継続を持つ CbC を利用して、証明可能な OS を実装したい。その為には証明に使用される定理証明支援系や、モデル検査機での表現に適した状態遷移単位での記述が求められる。CbC で使用する CodeGear は、状態遷移モデルにおける状態そのものとして捉えることが可能である。CodeGear を元にプログラミングをするにつれて、CodeGear の入出力の Data も重要であることが解ってきた。CodeGear とその入出力である DataGear を基本とした OS として、GearsOS の設計を行っている。現在の GearsOS は並列フレームワークとして実装されており、実用的な OS のプロトタイプ実装として既存の OS 上への実装を目指している。

GearsOS では、CodeGear と DataGear を元にプログラミングを行う。遷移する各 CodeGear の実行に必要なデータの整合性の確認などのメタ計算は、MetaCodeGear と呼ばれる各 CodeGear ごと実装された CodeGear で計算を行う。この MetaCodeGear の中で参照される DataGear を MetaDataGear と呼ぶ。

各 CodeGear の入出力や、各 CodeGear そのものの関数ポインタなどは、関数型プログラミングの側面から見るとプログラマが直接操作するのを禁じる必要がある。このために GearsOS には実行する処理に必要な CodeGear 及び DataGear を管理する、context という MetaDataGear が存在する。コード上では別の CodeGear に直接遷移している様に見えるが、実際は Context 内の遷移先の CodeGear に対応するスロットから、対応する MetaCodeGear に遷移する。CodeGear から別の CodeGear に遷移する際の DataGear などの関係性を、図 2 に示す。これらの変換は Perl スクリプトによって GearsOS のビルド時に行われる。

4. xv6 kernel

xv6 とはマサチューセッツ工科大学で v6 OS を元に開発された教育用の UNIX OS である。xv6 は ANSI C で実装されており、x86 アーキテクチャ上で動作する。Raspberry Pi 上での動作を目的とした ARM アーキテクチャのバージョンも存在する。本論文では最終的に Raspberry Pi 上での動作を目指しているために、 ARM アーキテクチャ上で動作する xv6 を扱う。

xv6 は小規模な OS だがファイルシステム、 プロセス、システムコールなどの UNIX の基本的な機能を持つ。またユーザー空間とカーネル空間が分離されており、 シェルや ls などのユーザーコマンドも存在する。

本論文では xv6 のファイルシステム関連の内部処理と、システムコール実行時に実行される処理について分析を行う。xv6 kernel のファイルシステムは階層構造で表現されており、 最も低レベルなものにディスク階層、 抽象度が最も高いレベルのものにファイル記述子がある。

5. xv6 のファイルシステムの一部の分析

xv6 のファイルシステムに関する定義ファイルは fs.c 中に記述されている。この中に出てくる関数に着目し、 この関数をさらに CodeGear に変換していくことで状態遷移単位での記述を試みた。

まず関数内で if 文などの分岐を持たない基本単位である Basic Block に着目した。CbC の CodeGear の粒度は C の関数とアセンブラの中間であるといえるので、 BasicBlock を CodeGear に置き換える事が可能である。したがって特定の関数内の処理の BasicBlock を分析し、 BasicBlock に対応した CodeGear へ変換することで状態遷移系への変換を行った。

参考文献

[1] Knuth, D. E.: *Fundamental Algorithms*, Art of Computer Programming, Vol. 1, chapter 2, pp. 371–381, Addison-Wesley, 2nd edition (1973).
[2] Schwartz, A. J.: Subdividing Bézier Curves and Surfaces, *Geometric Modeling: Algorithms and New Trends* (Farin, G. E., ed.), SIAM, Philadelphia, pp. 55–66 (1987).
[3] Baraff, D.: Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation, *SIGGRAPH '90 Proceedings* (Beach, R. J., ed.), Dallas, Texas, ACM, Addison-Wesley, pp. 19–28 (1990).
[4] Nakashima, H. et al.: OhHelp: A Scalable Domain-Decomposing Dynamic Load Balancing for Particle-in-Cell Simulations, *Proc. Intl. Conf. Supercomputing*, pp. 90–99 (online), DOI: http://doi.acm.org/10.1145/1542275.1542293 (2009).
[5] Adobe Systems Inc.: *PostScript Language Reference Manual*, Reading, Massachusetts (1985).
[6] 山下義行：文脈自由文法への否定の導入，修士論文，筑波大学大学院工学研究科 (1989).
[7] Weihl, W.: Specification and Implementation of Atomic Data Types, PhD Thesis, MIT, Boston (1984).
[8] Institute for New Generation Computer Technology: *Proc. Intl. Conf. on Fifth Generation Computer Systems*, Vol. 1 (1992).
[9] Aredon, I.: TEX 独稽古，Seminar on Mathematical Sciences 13, Department of Mathematics, Keio University, Yokohama (1989).
[10] 情報処理学会：コンピュータ博物館設立の提言，情報処理学会（オンライン），入手先 ⟨http://www.ipsj.or.jp/03somu/teigen/museum200702.html⟩ (参照 2007-02-05).
[11] 情報処理学会論文誌編集委員会：「情報処理学会論文誌（IPSJ Journal）」原稿執筆案内，情報処理学会（オンライン），入手先 ⟨http://www.ipsj.or.jp/08editt/journal/shippitsu/ronbunJ-prms.pdf⟩ (参照 2010-10-28).
[12] Kay, A.: Welcome to Squeakland, Squeakland (online), available from ⟨http://www.squeakland.org/community/biography/alanbio.html⟩ (accessed 2007-04-05).
[13] Nakashima, H.: A WEB Page, Kyoto University (online), available from ⟨http://www.para.media.kyoto-u.ac.jp/ nakashima/a.web.page.of.long.url/⟩ (accessed 2010-10-30).
[14] Nakashima, H.: Another WEB Page, Kyoto University (online), available from ⟨http://www.para.media.kyoto-u.ac.jp/ nakashima/a.web.page.of.much.longer.url/⟩ (accessed 2010-10-30).