Vol.2019-OS-146 No.4
2019/5/30

継続を用いたx.v6 kernelの書き換え

坂本昂弘[†1]　桃 原 優[†2]　河野真治[†1]

概要：x.v6 は MIT で教育用の目的で開発されたオペレーティングシステムで基本的な Unix の構造を持っている．当研究室で開発している Continuation based C (CbC) は継続を中心とした言語で，継続を用いることにより適切なプログラミング単位を提供し, 検証しやすくなると考えている．x.v6 を CbC で書き換えることで，オペレーティングシステムの基本的な機能を stack 抜きで実現することができると考えれる．また, CbC では Interface という module 化を提供しており, これにより実装と API そして, メタ計算 API を分離することができる．実際に書き換えを行なったが, 現状での書き換えは x.v6 の基本的な構造は変更していない．また, 未変換の部分と両立するように作られている．現在の書き換え手法について述べ, その有効性, 将来の書き換えの方針について考察する．

Abstract: x.v6 is an operating system developed at MIT for educational purposes and has a basic Unix structure. Continuation based C (CbC), developed by our laboratory, is a continuation-oriented language, and we believe that using continuation will provide an appropriate programming unit and make it easier to verify. By rewriting x.v6 with CbC, it is thought that the basic functions of the operating system can be realized without stack. In addition, CbC provides a modularization called Interface, and this makes it possible to implement, API, and The meta-calculation API can be separated. The current rewriting does not change the basic structure of x.v6, and it is made to be compatible with the unconverted part. This paper describes the current rewriting method and discusses its effectiveness and future rewriting policy.

1. x.v6 を継続で書き換える意味

現代の OS では拡張性と信頼性を両立させることが要求されている．信頼性をノーマルレベルの計算に対して保証し，拡張性をメタレベルの計算で実現することを目標に Gears OS を設計中である．

Gears OS は Continuation based C (CbC)[1] によってアプリケーションと OS そのものを記述する．OS の下ではプログラムの記述は通常の処理の他に，メモリ管理，スレッドの待ち合わせやネットワークの管理，エラーハンドリング等の記述しなければならない処理が存在する．これらの計算をメタ計算と呼ぶ．メタ計算を通常の計算から切り離して記述するために，Code Gear，Data Gear という単位を提案している．CbC はこの Code Gear と Data Gear の単位でプログラムを記述する．

Code Gear は goto による継続で処理を表すことができる．これにより，状態遷移による OS の記述が可能となり，複雑な OS のモデル検査を可能とする．また，CbC は 定理証明支援系 Agda に置き換えることができるように構築されている．

これらを用いて OS の信頼性を保証したい．CbC の有効性を示すために，基本的な機能を揃えた OS である xv6 を CbC で書き換える．これにより，OS の個々のシステムコールの持つ状態を明確にすることができると考えている．

CbC でシステムやアプリケーションを記述するためには，Code Gear と Data Gear を柔軟に再利用する必要があり，機能を接続する API と実装の分離が可能であることが望ましい．CbC では形式化されたモジュールシステムが提供されている．xv6 の CbC 書き換えでは, このモジュールシステムを用いる．

書き換えはまだ部分的であるが, 既存の部分と両立するように設計されている．従って, x.v6 の基本的な動作には変更はない．実際に実行速度などはほとんど差がない．逆に言えばオーバヘッドが存在しないことが確認できた．

2. Continuation based C

CbC は処理を Code Gear という単位を用いて記述するプログラミング言語である． Code Gear 間では軽量継続 (goto 文) による遷移を行うので，継続前の Code Gear に

†1 現在，琉球大学工学部情報工学科
Presently with Information Engineering, University of the Ryukyus.
†2 現在，琉球大学大学院理工学研究科情報工学専攻
Presently with Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.

戻 ることはない．この記述方法は図 1 のように状態遷移ベースのプログラミングに適している．

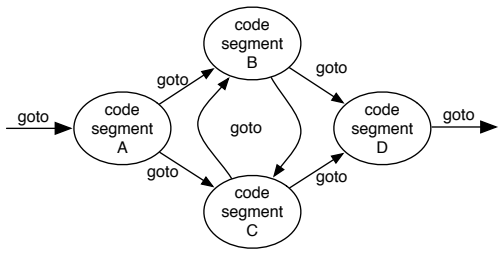

図 1: goto による code gear 間の継続

現在 CbC は C コンパイラである GCC[2] 及び LLVM[3] をバックエンドとした clang 上で実装されている．今回 xv6 の kernel の部分をこの CbC を 用いて書き換える．

3. Gears におけるメタ計算

プログラムを記述する際，ノーマルレベルの処理の他に，メモリ管理，スレッド管理，CPU や GPU の資源管理等，記述しなければならない処理が存在する．これらの計算をメタ計算と呼ぶ．

Code Gear，Data Gear にはそれぞれメタレベルの単位である Meta Code Gear，Meta Data Gear が存在し，これらを用いてメタ計算を実現する．

Gears OS には Context と呼ばれる全ての Code Gear，Data Gear のリストを持つ Meta Data Gear が存在する．Gears OS ではこの Context を常に持ち歩いているが，これはノーマルレベルでは見えることはない．

ノーマルレベルの処理とメタレベルを含む処理は同じ動作を行う．しかしメタレベルの計算を含むプログラムとノーマルレベルでは，Data Gear の扱いなどでギャップがある．ノーマルレベルでは Code Gear は Data Gear を引数の集合として引き渡しているが，メタレベルでは Context に格納されており，ここを参照することで Data Gear を扱っている．

このギャップを解消するためにメタレベルでは stub Code Gear と呼ばれる Context から Data Gear の参照を行う Meta Code Gear が Code Gear 継続前に挿入されこれを解決する．

従来の研究でメタ計算を用いる場合は，関数型言語では Monad を用いる [4]．これは Haskell では実行時の環境を記述する構文として使われている．OS の研究では，メタ計算の記述に型つきアセンブラ (Typed Assembler) を用いることがある [5]．Python や Haskell による記述をノーマルレベルとして採用した OS の検証の研究もある [6], [7]．SMIT などのモデル検査を OS の検証に用いた例もある [8]．

本研究で用いる Meta Gear は制限された Monad に相当し，型つきアセンブラよりは大きな表現単位を提供する．Haskell などの関数型プログラミング言語では実行環境が複雑であり，実行時の資源使用を明確にすることができない．CbC はスタック上に隠された環境を持たないので，メタレベルで使用する資源を明確にできるという利点がある．ただし，Gear のプログラミングスタイルは，従来の関数呼出を中心としたものとはかなり異なる．

4. Interface

Interface は Gears OS のモジュール化の仕組みである．Interface は呼び出しの引数になる Data Gear の集合であり，そこで呼び出される Code Gear のエントリである．呼び出される Code Gear の引数となる Data Gear はここで全て定義される．Interface を定義することで複数の実装を持つことができる．図 2 は Stack の Interface とその実装を表したものである．

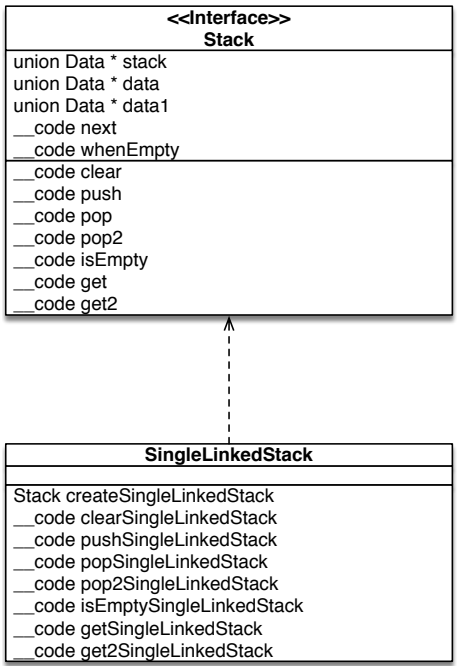

図 2: Stack の Interface とその実装

Data Gear は，ノーマルレベルとメタレベルで見え方が異なる．ノーマルレベルの Code Gear では Data Gear の引数に見える．しかし，メタレベルでは Data Gear は Context が持つ構造体である．この見え方の違いを Meta Code Gear である stub Code Gear によって調整する必要がある．

また，CbC は関数呼び出しと異なり，goto による継続で遷移を行う．このため CbC の継続にはスタックフレームがなく引数を格納する場所がない．

Context は初期化の際に引数格納用の Data Gear の領域を確保する．Code Gear が継続する際にはこの領域に引数の Data Gear を格納する．この領域に確保された Data

Gear へのアクセスは Interface の情報から行われる．

ソースコード 1 は，pushSingleLinkedStack のソースコードである．ノーマルレベルの Code Gear では Stack の push の操作は，push するデータと次の継続先の Code Gear という引数の集合のように見える．しかしメタレベルでは Context が持つ構造体である．

stub Code Gear では Context が確保した 引数格納用の領域に格納した Data Gear を取り出している．Interface を導入することでノーマルレベルとメタレベルのズレの調整を解決した．

ソースコード 1: pushSingleLinkedStack

```
1  __code stackTest1(struct Stack* stack) {
2      Node* node = new Node();
3      node->color = Red;
4      goto stack->push(node, stackTest2);
5  }
6
7  __code pushSingleLinkedStack_stub(struct
       Context* context) {
8      SingleLinkedStack* stack = (
       SingleLinkedStack*)context->data[D_Stack]->
       Stack.stack->Stack.stack;
9      Data* data = context->data[D_Stack]->Stack.
       data;
10     num Code next = context->data[D_Stack]->
       Stack.next;
11     goto pushSingleLinkedStack(context, stack,
       data, next);
12 }
13
14 __code pushSingleLinkedStack(struct
       SingleLinkedStack* stack, union Data* data,
        __code next(...)) {
15     Element* element = new Element();
16     element->next = stack->top;
17     element->data = data;
18     stack->top = element;
19     goto next(...);
20 }
```

ソースコード 2 は Stack の Interface である．typedef struct Stack で Interface を定義する．Impl には実装の型が入る．ソースコード 2，2〜4 行目の union Data で定義されてるものは，Interface の API で用いる全ての Data Gear である．Interface の全ての API で用いる全ての Data Gear は Interface で定義される．ソースコード 2，5〜13 行目の __code で記述されているものは，Interface の API である．ここでは Stack の API である push や pop などの Code Gear となっている．

ソースコード 2: Stack の Interface

```
1  typedef struct Stack<Impl>{
2      union Data* stack;
3      union Data* data;
4      union Data* data1;
5      __code whenEmpty(...);
6      __code clear(Impl* stack,__code next(...));
7      __code push(Impl* stack,union Data* data,
       __code next(...));
8      __code pop(Impl* stack, __code next(union
       Data*, ...));
9      __code pop2(Impl* stack, __code next(union
       Data*, union Data*, ...));
10     __code isEmpty(Impl* stack, __code next
       (...), __code whenEmpty(...));
11     __code get(Impl* stack, __code next(union
       Data*, ...));
12     __code get2(Impl* stack, __code next(union
       Data*, union Data*, ...));
13     __code next(...);
14 } Stack;
```

通常 Code Gear，Data Gear に参照するためには Context を通す必要があるが，Interface を記述することでデータ構造の API と Data Gear を結びつけることが出来る．これによりノーマルレベルとメタレベルの分離が可能となった．

ソースコード 3 は Stack の実装の例である．createImpl は実装の初期化を行う関数である．Interface の実装を呼び出す際，この関数を呼び出すことで ソースコード 1 4 行目のように実装の操作を呼び出せるようになる．ソースコード 3 2 行目は操作する Stack のデータ構造の確保をしている．SingleLinkedStack のデータ構造は ソースコード 4 である．ソースコード 3 6〜12 行目で実装の Code Gear に代入しているものは Context が持つ enum で定義された Code Gear の番号である．

ソースコード 3: SingleLinkedStack の実装

```
1  Stack* createSingleLinkedStack(struct Context*
       context) {
2      struct Stack* stack = new Stack();
3      struct SingleLinkedStack* singleLinkedStack
        = new SingleLinkedStack();
4      stack->stack = (union Data*)
       singleLinkedStack;
5      singleLinkedStack->top = NULL;
6      stack->push = C_pushSingleLinkedStack;
7      stack->pop  = C_popSingleLinkedStack;
8      stack->pop2  = C_pop2SingleLinkedStack;
9      stack->get  = C_getSingleLinkedStack;
10     stack->get2  = C_get2SingleLinkedStack;
11     stack->isEmpty = C_isEmptySingleLinkedStack
       ;
12     stack->clear = C_clearSingleLinkedStack;
13     return stack;
14 }
15
16 __code clearSingleLinkedStack(struct
       SingleLinkedStack* stack,__code next(...))
       {
17     stack->top = NULL;
18     goto next(...);
19 }
20
21 __code pushSingleLinkedStack(struct
       SingleLinkedStack* stack,union Data* data,
       __code next(...)) {
22     Element* element = new Element();
23     element->next = stack->top;
24     element->data = data;
25     stack->top = element;
```

```
26      goto next(...);
27  }
```

ソースコード 4: SingleLinkedStack のデータ構造

```
1   struct SingleLinkedStack {
2       struct Element* top;
3   } SingleLinkedStack;
4
5   struct Element {
6       union Data* data;
7       struct Element* next;
8   } Element;
```

5. xv6 の CbC への書き換え

xv6 は 2006 年に MIT のオペレーティングシステムコースで教育用の目的として開発されたオペレーティングシステムである．xv6 はプロセス，仮想メモリ，カーネルとユーザの分離，割り込み，ファイルシステムなどの基本的な Unix の構造を持つにも関わらず，シンプルで学習しやすい．

CbC は 継続を中心とした言語であるため状態遷移モデルに落とし込むことができる．xv6 を CbC で書き換えることにより，OS の機能の保証が可能となる．

また，ハードウェア上でのメタレベルの計算や並列実行を行いたい．このため，xv6 を ARM[9] プロセッサを搭載したシングルボードコンピュータである Raspberry pi 用に移植した xv6-rpi を用いて実装する．

xv6 全体を CbC で書き換えるためには Interface を用いてモジュール化する必要がある．xv6 をモジュール化し，CbC で書き換えることができれば，Gears OS の機能を置き換えることもできる．図 3 は xv6 の構成図となる．

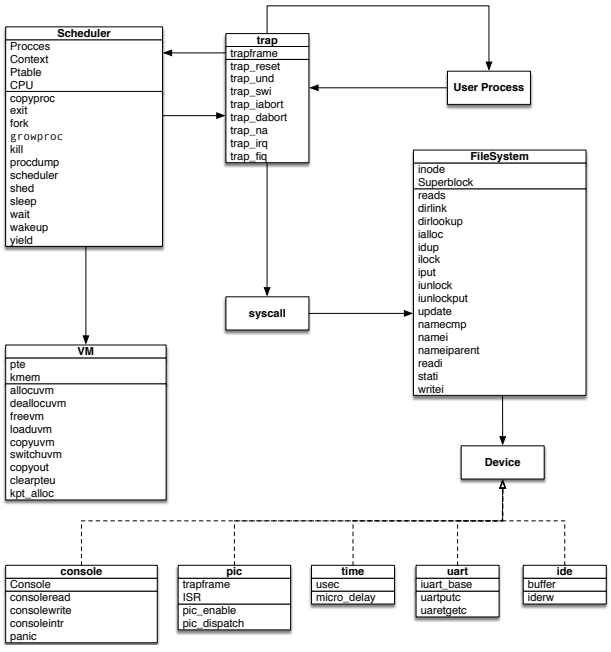

図 3: モジュール化した xv6 の構成

xv6 はカーネルと呼ばれる形式をとっている．カーネルは OS にとって中核となるプログラムである．xv6 ではカーネルとユーザープログラムは分離されており，カーネルはプログラムにプロセス管理，メモリ管理，I/O やファイルの管理などのサービスを提供する．ユーザープログラムがカーネルのサービスを呼び出す場合，システムコールを用いてユーザー空間からカーネル空間へ入りサービスを実行する．カーネルは CPU のハードウェア保護機構を使用して，ユーザー空間で実行されているプロセスが自身のメモリのみアクセスできるように保護している．ユーザープログラムがシステムコールを呼び出すと，ハードウェアが特権レベルを上げ，カーネルのプログラムが実行される．この特権レベルを持つプロセッサの状態をカーネルモード，特権のない状態をユーザーモードという．

ユーザープログラムがカーネルの提供するサービスを呼び出す際にはシステムコールを用いる．ユーザープログラムがシステムコールを呼び出すと，トラップが発生する．トラップが発生すると，ユーザープログラムは中断され，カーネルに切り替わり処理を行う．ソースコード 5 は xv6 のシステムコールのリストである．

ソースコード 5: xv6 のシステムコールのリスト

```
1   static int (*syscalls[])(void) = {
2           [SYS_fork]    =sys_fork,
3           [SYS_exit]    =sys_exit,
4           [SYS_wait]    =sys_wait,
5           [SYS_pipe]    =sys_pipe,
6           [SYS_read]    =sys_read,
7           [SYS_kill]    =sys_kill,
8           [SYS_exec]    =sys_exec,
9           [SYS_fstat]   =sys_fstat,
10          [SYS_chdir]   =sys_chdir,
11          [SYS_dup]     =sys_dup,
12          [SYS_getpid]  =sys_getpid,
13          [SYS_sbrk]    =sys_sbrk,
14          [SYS_sleep]   =sys_sleep,
15          [SYS_uptime]  =sys_uptime,
16          [SYS_open]    =sys_open,
17          [SYS_write]   =sys_write,
18          [SYS_mknod]   =sys_mknod,
19          [SYS_unlink]  =sys_unlink,
20          [SYS_link]    =sys_link,
21          [SYS_mkdir]   =sys_mkdir,
22          [SYS_close]   =sys_close,
23  };
```

プロセスとは，カーネルが実行するプログラムの単位である．xv6 のプロセスは，ユーザー空間メモリとカーネル用のプロセスの状態を持つ空間で構成されている．プロセスは独立しており，他のプロセスからメモリを破壊されたりすることはない．また，独立していることでカーネルそのものを破壊することもない．各プロセスの状態は struct proc によって管理されている．プロセスは fork システムコールによって新たに生成される．fork は新しく，親プロセスと呼ばれる呼び出し側と同じメモリ内容の，子プロセスと呼ばれるプロセスを生成する．fork システムコール

は，親プロセスであれば子プロセスの ID，子プロセスであれば 0 を返す．親プロセスと子プロセスは最初は同じ内容を持っているが，それぞれ異なるメモリ，レジスタで実行されているため，片方のメモリ内容を変更してももう片方に影響はない．exit システムコールはプロセスの停止を行い，メモリを解放する．wait システムコールは終了した子プロセスの ID を返す．子プロセスが終了するまで待つ．exec システムコールは呼び出し元のプロセスのメモリをファイルシステムのファイルのメモリイメージと置き換え実行する．ファイルには命令，データなどの配置が指定されたフォーマット通りになっていなければならない．xv6 は ELF と呼ばれるフォーマットを扱う．

ファイルディスクリプタは，カーネルが管理するプロセスが読み書きを行うオブジェクトを表す整数値である．プロセスは，ファイル，ディレクトリ，デバイスを開く，または既存のディスクリプタを複製することによって，ファイルディスクリプタを取得する．xv6 はプロセス毎にファイルディスクリプタのテーブルを持っている．ファイルディスクリプタは普通，0 が標準入力，1 が標準出力，2 がエラー出力として使われる．ファイルディスクリプタのテーブルのエントリを変更することで入出力先を変更することができる．1 の標準出力を close し，ファイルを open することでプログラムはファイルに出力することになる．ファイルディスクリプタはファイルがどのように接続するか隠すことでファイルへの入出力を容易にしている．

xv6 のファイルシステムはバイト配列であるデータファイルとデータファイルおよび他のディレクトリの参照を含むディレクトリを提供する．ディレクトリは root と呼ばれる特別なディレクトリから始まるツリーを形成している．絶対パスである ”/dir1/dir2/file1” というパスは root ディレクトリ内の dir1 という名前のディレクトリ内の dir2 という名前のディレクトリ内の file というデータファイルを指す．相対パスである ”dir2/file2” のようなパスは，現在のディレクトリ内の dir2 という名前のディレクトリ内の file というデータファイルを指す．

6. xv6-rpi の CbC 対応

オリジナルの xv6 は x86 アーキテクチャで実装されたものだが，xv6-rpi は Raspberry Pi 用に実装されたものである．

Raspberry Pi は ARM[9] プロセッサを搭載したシングルボードコンピュータである．教育用のコンピューターとして開発されたもので，低価格であり，小型であるため使い勝手が良い．ストレージにハードディスクや SSD を使用するのではなく，SD カードを用いる．HDMI 出力や USB ポートも備えており，開発に最適である．

Raspberry Pi には Raspberry Pi 3，Raspberry Pi 2，Raspberry Pi 1，Raspberry Pi Zero といったバージョンが存在する．

xv6-rpi を CbC で書き換えるために，GCC 上で実装した CbC コンパイラを ARM 向けに build し xv6-rpi をコンパイルした．これにより，xv6-rpi を CbC で書き換えることができるようになった．

7. CbC によるシステムコールの書き換え

CbC によるシステムコールの書き換えは，従来の xv6 のシステムコールの形から，大きく崩すことなく可能である．CbC は Code Gear 間の遷移は goto による継続で行われるため，状態遷移ベースでのプログラムに適している．xv6 を CbC で書き換えることにより，OS のプログラムを状態遷移モデルに落とし込むことができる．これにより状態遷移系のモデル検査が可能となる．図 4 は CbC 書き換えによる read システムコールの遷移図である．

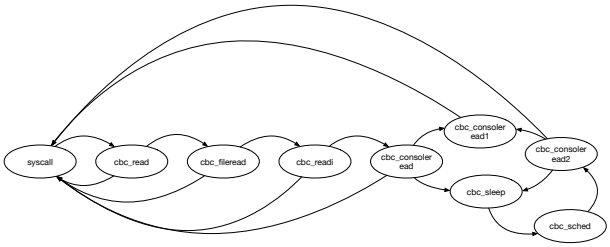

図 4: read システムコールの遷移図

ソースコード 6 は syscall() におけるシステムコールの呼び出しを行うコードである．システムコールはソースコード 5 の関数のリストの番号から呼び出される．CbC でも同様に num で指定された番号の cbccodes のリストの Code Gear へ goto する．ソースコード 6 6 行目でトラップフレームからシステムコールの番号を取得する．通常のシステムコールであればソースコード 6 13 行目からの分岐へ入るが，cbc システムコールであれば ソースコード 6 8 行目へ入り，goto によって遷移する．引数に持つ cbc_ret は 継続した先でトラップに戻ってくるための Code Gear である．

ソースコード 6: syscall()

```
1  void syscall(void)
2  {
3      int num;
4      int ret;
5
6      num = proc->tf->r0;
7
8      if((num >= NELEM(syscalls)) && (num <=
       NELEM(cbccodes)) && cbccodes[num]) {
9          proc->cbc_arg.cbc_console_arg.num = num
       ;
10         goto (cbccodes[num])(cbc_ret);
11     }
12
13     if((num > 0) && (num <= NELEM(syscalls)) &&
        syscalls[num]) {
```

```
14          ret = syscalls[num]();
15
16          // in ARM, parameters to main (argc,
        argv) are passed in r0 and r1
17          // do not set the return value if it is
         SYS_exec (the user program
18          // anyway does not expect us to return
        anything).
19          if (num != SYS_exec) {
20              proc->tf->r0 = ret;
21          }
22      }  else {
23          cprintf("%d %s: unknown sys call %d\n",
         proc->pid, proc->name, num);
24          proc->tf->r0 = -1;
25      }
26  }
```

ソースコード 7 は，read システムコールであるソースコード 8 を CbC で書き換えたコードである．CbC は C の関数を呼び出すことも出来るため，書き換えたい部分だけを書き換えることができる．Code Gear であるため，ソースコード 7 7，9 行目のように関数呼び出しではなく goto による継続となる．

ソースコード 7: cbc_read システムコール

```
1  __code cbc_read(__code (*next)(int ret)){
2      struct file *f;
3      int n;
4      char *p;
5
6      if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0
        || argptr(1, &p, n) < 0) {
7          goto next(-1);
8      }
9      goto cbc_fileread(f, p, n, next);
10 }
```

ソースコード 8: read システムコール

```
1  int sys_read(void)
2  {
3      struct file *f;
4      int n;
5      char *p;
6
7      if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0
        || argptr(1, &p, n) < 0) {
8          return -1;
9      }
10
11     return fileread(f, p, n);
12 }
```

ソースコード 9 は，ソースコード 10 を CbC で書き換えたコードである．継続で Code Gear 間を遷移するため関数呼び出しとは違い元の関数には戻ってこない．このため，書き換えの際には ソースコード 9 のように分割する必要がある．プロセスの状態を持つ struct proc は大域変数であるため Code Gear を用いるために必要なパラメーターを cbc_arg として持たせることにした．継続を行う際には必要なパラメーターをここに格納する．

ソースコード 9: fileread の CbC 書き換えの例

```
1  __code cbc_fileread1 (int r)
2  {
3      struct file *f = proc->cbc_arg.
       cbc_console_arg.f;
4      __code (*next)(int ret) = cbc_ret;
5      if (r > 0)
6          f->off += r;
7      iunlock(f->ip);
8      goto next(r);
9  }
10
11 __code cbc_fileread (struct file *f, char *addr
       , int n, __code (*next)(int ret))
12 {
13     if (f->readable == 0) {
14         goto next(-1);
15     }
16
17     if (f->type == FD_PIPE) {
18         goto cbc_piperead(f->pipe, addr, n,
       next);
19         goto next(-1);
20     }
21
22     if (f->type == FD_INODE) {
23         ilock(f->ip);
24         proc->cbc_arg.cbc_console_arg.f = f;
25         goto cbc_readi(f->ip, addr, f->off, n,
       cbc_fileread1);
26     }
27
28     goto cbc_panic("fileread");
29 }
```

ソースコード 10: 書き換え前の fileread

```
1  int fileread (struct file *f, char *addr, int n
       )
2  {
3      int r;
4
5      if (f->readable == 0) {
6          return -1;
7      }
8
9      if (f->type == FD_PIPE) {
10         return piperead(f->pipe, addr, n);
11     }
12
13     if (f->type == FD_INODE) {
14         ilock(f->ip);
15
16         if ((r = readi(f->ip, addr, f->off, n))
        > 0) {
17             f->off += r;
18         }
19
20         iunlock(f->ip);
21
22         return r;
23     }
24
25     panic("fileread");
26 }
```

ソースコード 11 は，ソースコード 12 を書き換えたコー

ドである．CbC では cbc_devsw を定義しておりソースコード 11 11 行目で次の Code Gear へと継続する．cbc_devsw はソースコード 13 で初期化されており，cbc_consoleread へと継続する．

ソースコード 11: readi の CbC 書き換えの例

```
1  __code cbc_readi (struct inode *ip, char *dst,
       uint off, uint n, __code (*next)(int ret))
2  {
3      uint tot, m;
4      struct buf *bp;
5
6      if (ip->type == T_DEV) {
7          if (ip->major < 0 || ip->major >= NDEV
       || !cbc_devsw[ip->major].read) {
8              goto next(-1);
9          }
10
11         goto cbc_devsw[ip->major].read(ip, dst,
        n, next);
12     }
13
14     if (off > ip->size || off + n < off) {
15         goto next(-1);
16     }
17
18     if (off + n > ip->size) {
19         n = ip->size - off;
20     }
21
22     for (tot = 0; tot < n; tot += m, off += m,
       dst += m) {
23         bp = bread(ip->dev, bmap(ip, off /
       BSIZE));
24         m = min(n - tot, BSIZE - off%BSIZE);
25         memmove(dst, bp->data + off % BSIZE, m)
       ;
26         brelse(bp);
27     }
28
29     goto next(n);
30 }
```

ソースコード 12: 書き換え前の readi

```
1  int readi (struct inode *ip, char *dst, uint
       off, uint n)
2  {
3      uint tot, m;
4      struct buf *bp;
5
6      if (ip->type == T_DEV) {
7          if (ip->major < 0 || ip->major >= NDEV
       || !devsw[ip->major].read) {
8              return -1;
9          }
10
11         return devsw[ip->major].read(ip, dst, n
       );
12     }
13
14     if (off > ip->size || off + n < off) {
15         return -1;
16     }
17
18     if (off + n > ip->size) {
19         n = ip->size - off;
20     }
21
22     for (tot = 0; tot < n; tot += m, off += m,
       dst += m) {
23         bp = bread(ip->dev, bmap(ip, off /
       BSIZE));
24         m = min(n - tot, BSIZE - off%BSIZE);
25         memmove(dst, bp->data + off % BSIZE, m)
       ;
26         brelse(bp);
27     }
28
29     return n;
30 }
```

ソースコード 13: consoleinit

```
1  void consoleinit (void)
2  {
3      initlock(&cons.lock, "console");
4      initlock(&input.lock, "input");
5
6      devsw[CONSOLE].write = consolewrite;
7      devsw[CONSOLE].read = consoleread;
8      cbc_devsw[CONSOLE].write = cbc_consolewrite
       ;
9      cbc_devsw[CONSOLE].read = cbc_consoleread;
10
11     cons.locking = 1;
12 }
```

ソースコード 14 の cbc_consoleread は，ソースコード 15 を書き換えたものである．ソースコード 14 11，50，行目 では sleep へ継続する際，戻るべき継続先を一緒に送ることで，sleep から consoleread に戻ってくることができる．

ソースコード 14: consoleread の CbC 書き換えの例

```
1  __code cbc_consoleread2 ()
2  {
3      struct inode *ip = proc->cbc_arg.
       cbc_console_arg.ip;
4      __code(*next)(int ret) = proc->cbc_arg.
       cbc_console_arg.next;
5      if (input.r == input.w) {
6          if (proc->killed) {
7              release(&input.lock);
8              ilock(ip);
9              goto next(-1);
10         }
11         goto cbc_sleep(&input.r, &input.lock,
       cbc_consoleread2);
12     }
13     goto cbc_consoleread1();
14 }
15
16 __code cbc_consoleread1 ()
17 {
18     int cont = 1;
19     int n = proc->cbc_arg.cbc_console_arg.n;
20     int target = proc->cbc_arg.cbc_console_arg.
       target;
21     char* dst = proc->cbc_arg.cbc_console_arg.
       dst;
22     struct inode *ip = proc->cbc_arg.
       cbc_console_arg.ip;
```

```
23      __code(*next)(int ret) = proc->cbc_arg.
        cbc_console_arg.next;
24
25      int c = input.buf[input.r++ % INPUT_BUF];
26
27      if (c == C('D')) {  // EOF
28          if (n < target) {
29              // Save ^D for next time, to make
        sure
30              // caller gets a 0-byte result.
31              input.r--;
32          }
33          cont = 0;
34      }
35
36      *dst++ = c;
37      --n;
38
39      if (c == '\n') {
40          cont = 0;
41      }
42
43      if (cont == 1) {
44          if (n > 0) {
45              proc->cbc_arg.cbc_console_arg.n = n
        ;
46              proc->cbc_arg.cbc_console_arg.
        target = target;
47              proc->cbc_arg.cbc_console_arg.dst =
         dst;
48              proc->cbc_arg.cbc_console_arg.ip =
        ip;
49              proc->cbc_arg.cbc_console_arg.next
        = next;
50              goto cbc_sleep(&input.r, &input.
        lock, cbc_consoleread2);
51          }
52      }
53
54      release(&input.lock);
55      ilock(ip);
56
57      goto next(target - n);
58  }
59
60  __code cbc_consoleread (struct inode *ip, char
        *dst, int n, __code(*next)(int ret))
61  {
62      uint target;
63
64      iunlock(ip);
65
66      target = n;
67      acquire(&input.lock);
68
69      if (n > 0) {
70          proc->cbc_arg.cbc_console_arg.n = n;
71          proc->cbc_arg.cbc_console_arg.target =
        target;
72          proc->cbc_arg.cbc_console_arg.dst = dst
        ;
73          proc->cbc_arg.cbc_console_arg.ip = ip;
74          proc->cbc_arg.cbc_console_arg.next =
        next;
75          goto cbc_consoleread2();
76      }
77      goto cbc_consoleread1();
78  }
```

ソースコード 15: 書き換え前の consoleread

```
1   int consoleread (struct inode *ip, char *dst,
        int n)
2   {
3       uint target;
4       int c;
5
6       iunlock(ip);
7
8       target = n;
9       acquire(&input.lock);
10
11      while (n > 0) {
12          while (input.r == input.w) {
13              if (proc->killed) {
14                  release(&input.lock);
15                  ilock(ip);
16                  return -1;
17              }
18
19              sleep(&input.r, &input.lock);
20          }
21
22          c = input.buf[input.r++ % INPUT_BUF];
23
24          if (c == C('D')) {  // EOF
25              if (n < target) {
26                  // Save ^D for next time, to
        make sure
27                  // caller gets a 0-byte result.
28                  input.r--;
29              }
30
31              break;
32          }
33
34          *dst++ = c;
35          --n;
36
37          if (c == '\n') {
38              break;
39          }
40      }
41
42      release(&input.lock);
43      ilock(ip);
44
45      return target - n;
46  }
```

CbC による書き換えにより，状態遷移ベースへと書き換えることができた．現在はシステムコールの書き換えのみであるが，カーネル全体を書き換えることで，カーネルを状態遷移モデルに落とし込むことができる．

8. 結論

本論文では Gears OS のプロトタイプの設計と実装，メタ計算である Context と stub の生成を行う Perl スクリプトの記述を行った．さらに Raspberry Pi 上での Gears OS 実装の考察，xv6 の機能の一部を CbC で書き換えを行った．

Code Gear , Data Gear を処理とデータの単位として用いて Gears OS を設計した．Code Gear，Data Gear には

メタ計算を記述するための Meta Code Gear，Meta Data Gear が存在する．メタ計算を Meta Code Gear，によって行うことでメタ計算を階層化して行うことができる．Code Gear は関数より細かく分割されてるためメタ計算を柔軟に記述できる．Gears OS は Code Gear と Input/Output Data Gear の組を Task とし，並列実行を行う．

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される．Interface は使用される Data Gear の定義と，それに対する操作を行う Code Gear の集合である．Interface は複数の実装をもち，Meta Data Gear として定義される．従来の関数呼び出しでは引数をスタック上に構成し，関数の実装アドレスを Call するが，Gears OS では引数は Context 上に用意された Interface の Data Gear に格納され，操作に対応する Code Gear に goto する．

Context は使用する Code Gear，Data Gear をすべて格納している Meta Data Gear である．通常の計算から Context を直接扱うことはセキュリティ上好ましくない．このため Context から必要なデータを取り出して Code Gear に接続する Meta Code Gear である stub Code Gear を定義した．stub Code Gear は Code Gear 毎に記述され，Code Gear 間の遷移の前に挿入される．

これらのメタ計算の記述は煩雑であるため Perl スクリプトによる自動生成を行なった．これにより Gears OS のコードの煩雑さは改善され，ユーザーレベルではメタを意識する必要がなくなった．

ハードウェア上での Gears OS の実装を実現させるために Raspberry Pi 上での実装を考察した．比較的シンプルな OS である xv6 を CbC に書き換えることにした．

xv6 を CbC で書き換えることによって，実行可能な CbC プログラムで記述された OS がそのまま，抽象的な状態遷移モデルになる．それによりモデル検査，Agda による定理証明が可能となると考えている．

現在は xv6 のシステムコールの一部のみの書き換えと，設計のみしか行っていないので，カーネル全ての書き換えをおこなう．

Gears OS にはメタ計算を実装する context は par goto の機能がある．これらの機能を xv6 に組み込む方法について考察する必要がある．また，xv6-rpi は QEMU のみの動作でしか確認してないため，実機上での動作が可能なように実装する必要がある．

参考文献

[1] TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
[2] GNU Compiler Collection (GCC) Internals: *http://gcc.gnu.org/onlinedocs/gccint/*.
[3] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California (2004).
[4] Moggi, E.: Notions of Computation and Monads, *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92 (online), DOI: 10.1016/0890-5401(91)90052-4 (1991).
[5] Yang, J. and Hawblitzel, C.: Safe to the Last Instruction: Automated Verification of a Type-safe Operating System, *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, New York, NY, USA, ACM, pp. 99–110 (online), DOI: 10.1145/1806596.1806610 (2010).
[6] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp. 207–220 (online), DOI: 10.1145/1629575.1629596 (2009).
[7] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F. and Zeldovich, N.: Using Crash Hoare Logic for Certifying the FSCQ File System, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 18–37 (online), DOI: 10.1145/2815400.2815402 (2015).
[8] Sigurbjarnarson, H., Bornholt, J., Torlak, E. and Wang, X.: Push-button Verification of File Systems via Crash Refinement, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, USENIX Association, pp. 1–16 (online), available from ⟨http://dl.acm.org/citation.cfm?id=3026877.3026879⟩ (2016).
[9] ARM Architecture Reference Manual: *http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html*.
[10] Andrew S.Tanenbaum, H. B.: Modern Operating Systems (2015).
[11] : Raspberry Pi — Teach, Learn, and Make with Raspberry Pi, https://www.raspberrypi.org.
[12] Wang, Z.: xv6-rpi, https://code.google.com/archive/p/xv6-rpi/ (2013).
[13] MASATAKA, H. and KONO, S.: GearsOS の Hoare Logic をベースにした検証手法，ソフトウェアサイエンス研究会 (2019).
[14] Norell, U.: Dependently Typed Programming in Agda, *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, New York, NY, USA, ACM, pp. 1–2 (online), DOI: 10.1145/1481861.1481862 (2009).
[15] 河野真治, 伊波立樹, 東恩納琢偉：Code Gear、Data Gear に基づく OS のプロトタイプ，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) (2016).
[16] Russ Cox, Frans Kaashoek, Robert Morris: xv6 a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf.