修士 (工学) 学位論文
Master's Thesis of Engineering

GearsOS のメタ計算

2021 年 3 月
March 2021

清水 隆博
Takahiro Shimizu

琉球大学
大学院理工学研究科
情報工学専攻

Information Engineering Course
Graduate School of Engineering and Science
University of the Ryukyus

指導教員：教授 和田 知久
Supervisor: Prof. Tomohisa Wada

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　和田 知久

印

(副　査)　　山田 孝治

印

(副　査)　　當間 愛晃

印

(副　査)　　河野 真治

要旨

ここに要旨を書く

Abstract

hogefuga

発表履歴

- 宮城 光希, 桃原 優, 河野真治. GearsOS のモジュール化と並列 API. 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, 2018
- 桃原 優, 東恩納琢偉, 河野真治. GearsOS の Paging と Segmentation・システムソフトウェアとオペレーティング・システム（OS）, May, 2019

目 次

図 目 次

表 目 次

ソースコード目次

第1章　OSとアプリケーションの信頼性

コンピューター上では様々なアプリケーションが常時動作している。動作しているアプリケーションは信頼性が保証されていてほしい。信頼性の保証には、 実行してほしい一連の挙動をまとめた仕様と、 それを満たしているかどうかの確認である検証が必要となる。アプリケーション開発では検証に関数や一連の動作をテストを行う方法や、デバッグを通して信頼性を保証する手法が広く使われている。

実際にアプリケーションを動作させるOSは、アプリケーションよりさらに高い信頼性が保証される必要がある。OSはCPUやメモリなどの資源管理と、 ユーザーにシステムコールなどのAPIを提供することで抽象化を行っている。OSの信頼性の保証もテストコードを用いて証明することも可能ではあるが、 アプリケーションと比較するとOSのコード量、 処理の量は膨大である。またOSはCPU制御やメモリ制御、 並列・並行処理などを多用する。テストコードを用いて処理を検証する場合、テストコードとして特定の状況を作成する必要がある。実際にOSが動作する中でバグやエラーを発生する条件を、並列処理の状況などを踏まえてテストコードで表現するのは困難である。非決定的な処理を持つOSの信頼性を保証するには、 テストコード以外の手法を用いる必要がある。

テストコード以外の方法として、 形式手法的と呼ばれるアプローチがある。形式手法の具体的な検証方法の中で、 証明を用いる方法[1][2][3][4]とモデル検査を用いる方法がある。証明を用いる方法ではAgda[5]やCoq[6]などの定理証明支援系を利用し、 数式的にアルゴリズムを記述する。Curry-Howard同型対応則により、型と論理式の命題が対応する。この型を導出するプログラムと実際の証明が対応する。証明には特定の型を入力として受け取り、証明したい型を生成する関数を作成する。整合性の確認は、記述した関数を元に定理証明支援系が検証する。証明を使う手法の場合、 実際の証明を行うのは定理証明支援系であるため、 定理証明支援系が理解できるプログラムで実装する必要がある。AgdaやCoqの場合はAgda、 Cow自身のプログラムで記述する必要がある。しかしAgdaで証明ができてもAgdaのコードを直接OSのソースコードとしてコンパイルすることはできない。Agda側でCのソースコードを吐き出せれば可能ではあるが、 現状は検証したコードと実際に動作するコードは分離されている。検証されたアルゴリズムをもとにCで実装することは可能であるが、 この場合移植時にバグが入る可能性がある。検証ができているソースコードそのものを使ってOSを動作させたい。

他の形式手法にモデル検査がある。モデル検査は実際に動作するコードですべての可

能な実行の組み合わせを実行し検証する方法である。例えば Java のソースコードに対してモデル検査をする JavaPathFinder などがある。モデル検査を利用する場合は、実際に動作するコード上で検証を行うことが出来る。OS のソースコードそのものをモデル検査すると、実際に検証された OS が動作可能となる。しかし OS の処理は膨大である。すべての存在可能な状態を数え上げるモデル検査では状態爆発が問題となる。状態を有限に制限したり抽象化を行う必要がある。

OS のシステムコールは、ユーザーから API 経由で呼び出され、 いくつかの処理を行う。その処理に着目すると OS は様々な状態を遷移して処理を行っていると考えることができる。OS を巨大な状態遷移マシンと考えると、 OS の処理の特定の状態の遷移まで範囲を絞ることができる。範囲が限られているため、有限時間でモデル検査などで検証することが可能である。この為には OS の処理を証明しやすくする表現で実装する必要がある。[7] 証明しやすい表現の例として、 状態遷移ベースでの実装がある。

証明を行う対象の計算は、 その意味が大きく別けられる。OS やプログラムの動作においては本来したい計算がまず存在する。これはプログラマが通常プログラミングするものである。これら本来行いたい処理のほかに、 CPU、メモリ、スレッドなどの資源管理なども必要となる。前者の計算をノーマルレベルの計算と呼び、後者をメタレベルの計算と呼ぶ。OS はメタ計算を担当していると言える。ユーザーレベルから見ると、データの読み込みなどは資源へのアクセスが必要であるため、システムコールを呼ぶ必要がある。システムコールを呼び出すと OS が管理する資源に対して何らかの副作用が発生するメタ計算と言える。副作用は関数型プログラムの見方からするとモナドと言え、 モナドもメタ計算ととらえることができる。OS 上で動くプログラムは CPU により並行実行される。この際の他のプロセスとの干渉もメタレベルの処理である。実装のソースコードはノーマルレベルであり検証用のソースコードはメタ計算だと考えると、OS そのものが検証を行ない、システム全体の信頼を高める機能を持つべきだと考える。ノーマルレベルの計算を確実に行う為には、メタレベルの計算が重要となる。

プログラムの整合性の検証はメタレベルの計算で行いたい。ユーザーが実装したノーマルレベルの計算に対応するメタレベルの計算を、自由にメタレベルの計算で証明したい。またメタレベルで検証ががすでにされたプログラムがあった場合、都度実行ユーザーの環境で検証が行われるとパフォーマンスに問題が発生する。この場合はメタレベルの計算を検証をするもの、しないものと切り替えられる柔軟な API が必要となる。メタレベルの計算をノーマルレベルの計算と同等にプログラミングできると、動作するコードに対して様々なアプローチが掛けられる。この為にはノーマルレベル、メタレベル共にプログラミングできる言語と環境が必要となる。

プログラムのノーマルレベルの計算とメタレベルの計算を一貫して行う言語として、Continuation Based C(CbC) を用いる。CbC は基本 `goto` 文で `CodeGaar` というコードの単位を遷移する言語である。通常の関数呼び出しと異なり、スタックあるいは環境と呼ば

れる隠れた状態を持たない。このため、計算のための情報は CodeGear の入力にすべてそろっている。そのうちのいくつかはメタ計算、つまり、OS が管理する資源であり、その他はアプリケーションを実行するためのデータ (DataGear) である。メタ計算とノーマルレベルの区別は入力のどこを扱うかの差に帰着される。CbC は C と互換性のある C の下位言語である。CbC は GCC[8][9] あるいは LLVM[10][11] 上で実装されていて、通常の C のアプリケーションやシステムプログラムをそのまま包含できる。C のコンパイルシステムを使える為に、 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 [12] 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。

第2章　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 の call/cc 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て `for` 文などのループの代わりに再起呼び出しを行う。これは関数型プログラミングでの Tail call スタイルでプログラミングすることに相当する。Agda よる関数型の CbC の記述も用意されている。実際の OS やアプリケーションを記述する場合には、GCC 及び LLVM/clang 上の CbC 実装を用いる。

2.1　CodeGear

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。各 CodeGear は DataGear と呼ばれるデータの単位で入力を受け取り、 その結果を別の DataGear に書き込む。入力の DataGear を InputDataGear と呼び、 出力の DataGear を OutputDataGear と呼ぶ。CodeGear がアクセスできる DataGear は、 InputDataGear と OutputDataGear に限定される。

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。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。

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。これは呼び出し元の関数を次の CodeGear の継続対象として設定するものである。これは GCC では内部コードを生成を行う。LLVM/clang では `setjmp` と `longjmp` を使い実装している。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。

2.2 DataGear

2.3 CbC を使ったシステムコールディスパッチの例題

CbC を用いて MIT が開発した教育用の OS である xv6[13] の書き換えを行った。CbC を利用したシステムコールのディスパッチ部分をソースコード 2.1 に示す。この例題では特定のシステムコールの場合、CbC で実装された処理に goto 文をつかって継続する。例題では CodeGear へのアドレスが配列 cbccodes に格納されている。引数として渡している cbc_ret は、 システムコールの返り値の数値をレジスタに代入する CodeGear である。実際に cbc_ret に継続が行われるのは、 read などのシステムコールの一連の処理の継続が終わったタイミングである。

ソースコード 2.1: CbC を利用したシステムコールのディスパッチ

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

第3章　GearsOS

GearsOS とは Continuation Based C を用いて実装している OS プロジェクトである。CodeGear と DataGear を基本単位として実行する。GearsOS は OS として実行する側面と、 CbC のシンタックスを拡張した言語フレームワークとしての側面がある。

3.1　GearsOSのビルドシステム

GearsOS ではビルドツールに CMake を利用している。ビルドフローを図 3.1 に示す。CMake は automake などの Make ファイルを作成するツールに相当するものである。GearsOS でプログラミングする際は、ビルドしたいプロジェクトを CMakeLists.txt に記述する。CMake は自身がコンパイルをすることはなく、ビルドツールである make や ninja-build に処理を移譲している。CMake は make や ninja-build が実行可能な Makefile、 build.ninja の生成までを担当する。

GearsOS のビルドでは直接 CbC コンパイラがソースコードをコンパイルすることはなく、 間に Perl スクリプトが 2 種類実行される。Perl スクリプトはビルド対象の GearsOS で拡張された CbC ファイルを、純粋な CbC ファイルに変換する。ほかに GearsOS で動作する例題ごとに必要な初期化関数なども生成する。Perl スクリプトで変換された CbC ファイルなどをもとに CbC コンパイラがコンパイルを行う。ビルドの処理は自動化されており、 CMake 経由で make や ninja コマンドを用いてビルドする。

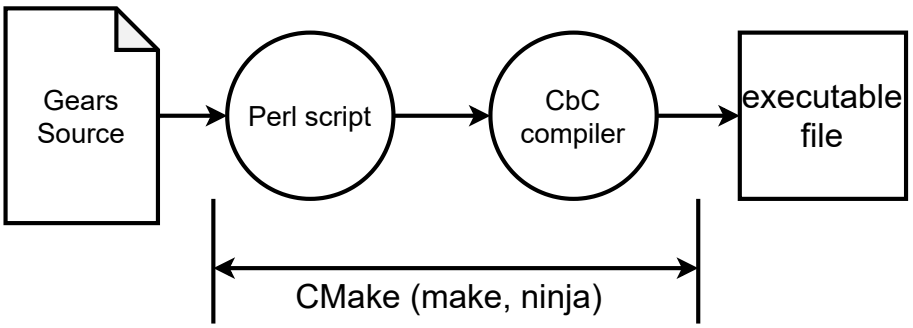

図 3.1: GearsOS のビルドフロー

3.2　pmake

GearsOS をビルドする場合は、x86 アーキテクチャのマシンからビルドするのが殆どである。この場合ビルドしたバイナリは x86 向けのバイナリとなる。これはビルドをするホストマシンに導入されている CbC コンパイラが x86 アーキテクチャ向けにビルドされたものである為である。

CbC コンパイラは GCC と llvm/clang 上に構築した 2 種類が主力な処理系である。LVM/-clang の場合は LLVM 側でターゲットアーキテクチャを選択することが可能である。GCC の場合は最初から j ターゲットアーキテクチャを指定してコンパイラをビルドする必要がある。

時にマシンスペックの問題などから、別のアーキテクチャ向けのバイナリを生成したいケースがある。教育用マイコンボードである Raspberry Pi[14] は ARM アーキテクチャが搭載されている。Raspberry Pi 上で GearsOS のビルドをする場合、ARM 用にビルドされた CbC コンパイラが必要となる。Raspberry Pi 自体は非力なマシンであるため、GearsOS のビルドはもとより CbC コンパイラの構築を Raspberry Pi 上でするのは困難である。マシンスペックが高めの x86 マシンから ARM 用のバイナリをビルドして、 Raspberry Pi に転送し実行したい。ホストマシンのアーキテクチャ以外のアーキテクチャ向けにコンパイルすることをクロスコンパイルと呼ぶ。

GearsOS はビルドツールに CMake を利用しているので、 CMake でクロスコンパイル出来るように工夫をする必要がある。ビルドに使用するコンパイラやリンカは CMake が自動探索し、決定した上で Makefile や build.ninja ファイルを生成する。しかし CMake は今ビルドしようとしている対象が、自分が動作しているアーキテクチャかそうでないか、クロスコンパイラとして使えるかなどはチェックしない。つまり CMake が自動でクロスコンパイル対応の GCC コンパイラを探すことはない。その為そのままビルドすると x86 用のバイナリが生成されてしまう。

CMake を利用してクロスコンパイルする場合、CMake の実行時に引数でクロスコンパイラを明示的に指定する必要がある。この場合 x86 のマシンから ARM のバイナリを出力する必要があり、 コンパイラやリンカーなどを ARM のクロスコンパイル対応のものに指定する必要がある。また、 xv6 の場合はリンク時に特定のリンカスクリプトを使う必要がある。これらのリンカスクリプトも CMake 側に、 CMake が提供しているリンカ用の特殊変数を使って自分で組み立てて渡す必要がある。このような CMake の処理を手打ちで行うことは難しいので、 `pmake.pl` を作成した。`pmake.pl` の処理の概要を図 3.2 に示す。`pmake.pl` は Perl スクリプトで、 シェルコマンドを内部で実行しクロスコンパイル用のオプションを組み立てる。`pmake.pl` を経由して CMake を実行すると、 make コマンドに対応する Makefile、 ninja-build に対応する build.ninja が生成される。以降は cmake ではなく make などのビルドツールがビルドを行う。

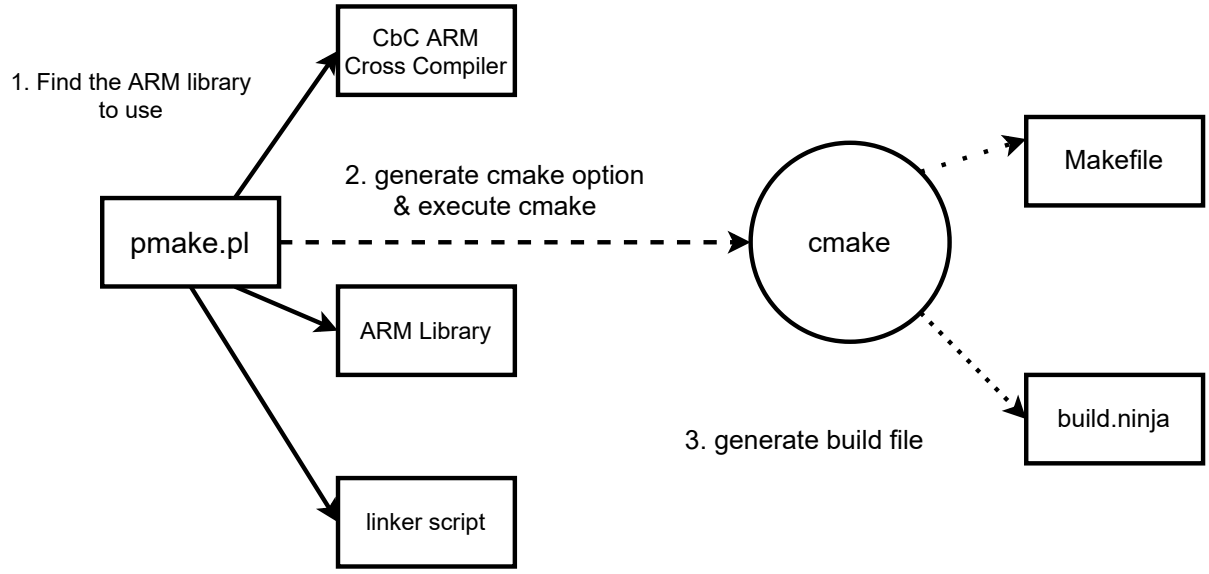

図 3.2: pmake.pl の処理フロー

3.3　Interface の取り扱い方法の検討

GearsOS の Interface はモジュール化の仕組みと goto 文での引数の一時保管場所としての機能を持っている。Interface の Implement のヘッダーファイルを実装したことで、GearsOS 上で Interface を実装する際に新たな方法での実装を検討した。Implement の CodeGear は今までは Interface で定義した CodeGear と 1 対 1 対応していた。Implement の CodeGear から goto する先は、入力として与えられた CodeGear か、Implement 内で独自に定義した CodeGear に goto するケースとなっていた。後者の独自に定義した CodeGear に goto するケースも、 実装の CbC ファイルの中に記述されている CodeGear に遷移していた。

GearsOS を用いて xv6 OS を再実装した際に、 実装側の CodeGear を細かく別けて記述した。細分化によって 1 つの CbC ファイルあたりの CodeGear の記述量が増えてしまうという問題が発生した。見通しをよくする為に、 Interface で定義した CodeGear と直接対応する CodeGear の実装と、 それらから goto する CodeGear で実装ファイルを分離することを試みた。

第4章　GearsOSのトランスコンパイラ

GearsOS は CbC で実装を行う。CbC は C 言語よりアセンブラに近い言語であるため、すべてを純粋な CbC で記述しようとすると記述量が膨大になってしまう。またノーマルレベルの計算とメタレベルの計算を、全てプログラマが記述する必要が発生してしまう。メタ計算では値の取り出しなどを行うが、 これはノーマルレベルの CodeGear の API が決まれば一意に決定される。したがってノーマルレベルのみ記述すれば、 機械的にメタ部分の処理は概ね生成可能となる。また、メタレベルのみ切り替えたいなどの状況が存在する。ノーマルレベル、メタレベル共に同じコードの場合は記述の変更量が膨大であるが、メタレベルの作成を分離するとこの問題は解消される。

GearsOS ではメタレベルの処理の作成に Perl スクリプトを用いており、ノーマルレベルで記述された CbC から、メタ部分を含む CbC へと変換する。変換前の CbC を GearsCbC と呼ぶ。

4.1　トランスコンパイラ

プログラミング言語から実行可能ファイルやアセンブラを生成する処理系のことを、一般的にコンパイラと呼ぶ。特定のプログラミング言語から別のプログラミング言語に変換するコンパイラのことを、 トランスコンパイラと呼ぶ。トランスコンパイラとしては JavaScript を古い規格の JavaScript に変換する Babel[15] がある。

またトランスコンパイラは、変換先の言語を拡張した言語の実装としても使われる。JavaScript に強い型制約をつけた拡張言語である TypeScript は、 TypeScript から純粋な JavaScript に変換を行うトランスコンパイラである。すべての TypeScript のコードは JavaScript にコンパイル可能である。JavaScript に静的型の機能を取り込みたい場合に使われる言語であり、 JavaScript の上位の言語と言える。

GearsOS は CbC を拡張した言語となっている。ただしこの拡張自体は CbC コンパイラである gcc、 llvm/clang には搭載されていない。その為 GearsOS の拡張部分を、等価な純粋な CbC の記述に変換する必要がある。現在の GearsOS では、 CMake によるコンパイル時に Perl で記述された `generate_stub.pl` と `generate_context.pl` の 2 種類のスクリプトで変換される。

- generate_stub.pl
  - 各 CbC ファイルごとに呼び出されるスクリプト
  - 対応するメタ計算を導入した CbC ファイル (拡張子は c) に変換する
    - 図 4.1 に処理の概要を示す
- generate_context.pl
  - 生成した CbC ファイルを解析し、使われている CodeGear を確定する
  - context.h を読み込み、使われている DataGear を確定する
  - Context 関係の初期化ルーチンや CodeGear、 DataGear の番号である enum を生成する
    - 図 4.2 に処理の概要を示す

これらの Perl スクリプトはプログラマが自分で動かすことはない。Perl スクリプトの実行手順は CMakeLists.txt に記述しており、 make や ninja-build でのビルド時に呼び出される。(ソースコード 4.1)

ソースコード 4.1: CMakeList.txt 内での Perl の実行部分

```
1  macro( GearsCommand )
2      set( _OPTIONS_ARGS )
3      set( _ONE_VALUE_ARGS TARGET )
4      set( _MULTI_VALUE_ARGS SOURCES )
5      cmake_parse_arguments( _Gears "${_OPTIONS_ARGS}" "${_ONE_VALUE_ARGS}"
       "${_MULTI_VALUE_ARGS}" ${ARGN} )
6
7      set (_Gears_CSOURCES)
8      foreach(i ${_Gears_SOURCES})
9          if (${i} MATCHES "\\.cbc")
10             string(REGEX REPLACE "(.*).cbc" "c/\\1.c" j ${i})
11             add_custom_command (
12                 OUTPUT    ${j}
13                 DEPENDS   ${i}
14                 COMMAND  "perl" "generate_stub.pl" "-o" ${j} ${i}
15             )
16         elseif (${i} MATCHES "\\.cu")
17             string(REGEX REPLACE "(.*).cu" "c/\\1.ptx" j ${i})
18             add_custom_command (
19                 OUTPUT    ${j}
20                 DEPENDS   ${i}
21                 COMMAND  nvcc ${NVCCFLAG} -c -ptx -o ${j} ${i}
22             )
23         else()
24             set(j ${i})
25         endif()
```

```
26          list(APPEND _Gears_CSOURCES ${j})
27      endforeach(i)
```

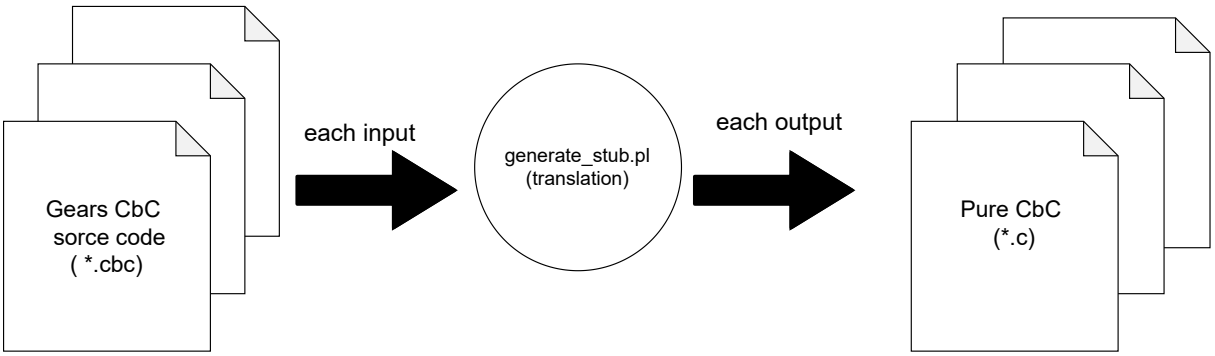

図 4.1: generate_sub.pl を使ったトランスコンパイル

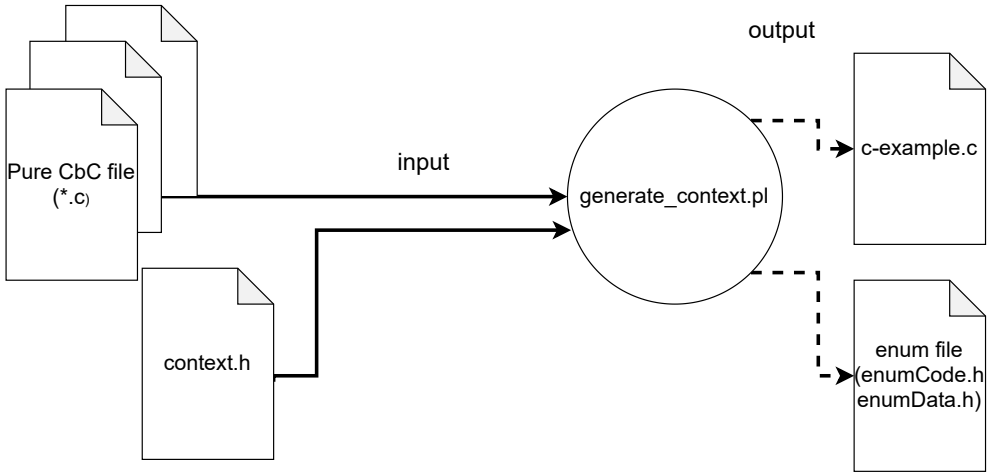

図 4.2: generate_context.pl を使ったファイル生成

4.2　GearsCbC の Interface の実装時の問題

Interface とそれを実装する Impl の型が決定すると、最低限満たすべき CodeGear の API は一意に決定する。ここで満たすべき CodeGear は、Interface で定義した CodeGear と、Impl 側で定義した private な CodeGear となる。例えば Stack Interface の実装を考えると、各 Impl で pop, push, shift, isEmpty などを実装する必要がある。

従来はプログラマが手作業でヘッダーファイルの定義を参照しながら.cbc ファイルを作成していた。手作業での実装のため、 コンパイル時に次のような問題点が多発した。

- CodeGear の入力のフォーマットの不一致
- Interface の実装の CodeGear の命名規則の不一致
- 実装を忘れている CodeGear の発生

特に GearsOS の場合は Perl スクリプトによって純粋な CbC に一度変換されてからコンパイルが行われる。実装の状況とトランスコンパイラの組み合わせによっては、 CbC コンパイラレベルでコンパイルエラーを発生させないケースがある。この場合は実際に動作させながら、gdb, lldb などの C デバッガを用いてデバッグをする必要がある。また CbC コンパイラレベルで検知できても、すでに変換されたコード側でエラーが出てしまうので、トランスコンパイラの挙動をトレースしながらデバッグをする必要がある。Interface の実装が不十分であることのエラーは、 GearsOS レベル、最低でも CbC コンパイラのレベルで完全に検知したい。

4.3　Interface を満たすコード生成の他言語の対応状況

Interface を機能として所持している言語の場合、Interface を完全に見たいしているかどうかはコンパイルレベルか実行時レベルで検知される。例えば Java の場合は Interface を満たしていない場合はコンパイルエラーになる。

Interface の API を完全に実装するのを促す仕組みとして、Interface の定義からエディタやツールが満たすべき関数と引数の組を自動生成するツールがある。

Java では様々な手法でこのツールを実装している。Microsoft が提唱している IDE とプログラミング言語のコンパイラをつなぐプロトコルに Language Server がある。Language Server はコーディング中のソースコードをコンパイラ自身でパースし、 型推論やエラーの内容などを IDE 側に通知するプロトコルである。主要な Java の Language Server の実装である eclipse.jdt.ls[16] では、 LanguageServer の機能として未実装のメソッドを検知する機能が実装されている。[17] この機能を応用して vscode 上から未実装のメソッドを特定し、 雛形を生成する機能がある。他にも IntelliJ IDE などの商用 IDE では、 IDE が独自に未実装のメソッドを検知、雛形を生成する機能を実装している。

golang の場合は主に `josharian/impl`[18] が使われている。これはインストールすると `impl` コマンドが使用可能になり、 実装したい Interface の型と、 Interface を実装する Impl の型 (レシーバ) を与えることで雛形が生成される。主要なエディタである vscode の golang の公式パッケージである `vscode-go`[19] でも導入されており、 vscode から呼び出すことが可能である。vscode 以外にも vim などのエディタから呼び出すことや、 シェル上で呼び出して標準出力の結果を利用することが可能である。

4.4　GearsOS での Interface を満たす CbC の雛形生成

GearsOS でも同様の Interface の定義から実装する CodeGear の雛形を生成したい。LanguageServer の導入も考えられるが、 今回の場合は C 言語の LanguageServer を CbC 用にまず改良し、 さらに GearsOS 用に書き換える必要がある。現状の GearsOS が持つシンタックスは CbC のシンタックスを拡張しているものではあるが、これは CbC コンパイラ側には組み込まれていない。LanguageServer を GearsOS に対応する場合、CbC コンパイラ側に GearsOS の拡張シンタックスを導入する必要がある。CbC コンパイラ側への機能の実装は、 比較的難易度が高いと考えらる。CbC コンパイラ側に手をつけず、 Interface の入出力の検査は既存の GearsOS のビルドシステム上に組み込みたい。

対して golang の `impl` コマンドのように、 シェルから呼び出し標準出力に結果を書き込む形式も考えられる。この場合は実装が比較的容易かつ、 コマンドを呼び出して標準出力の結果を使えるシェルやエディタなどの各プラットフォームで使用可能となる。先行事例を参考に、コマンドを実行して雛形ファイルを生成するコマンド `impl2cbc.pl` を GearsOS に導入した。`impl2cbc.pl` の処理の概要を図 4.3 に示す。

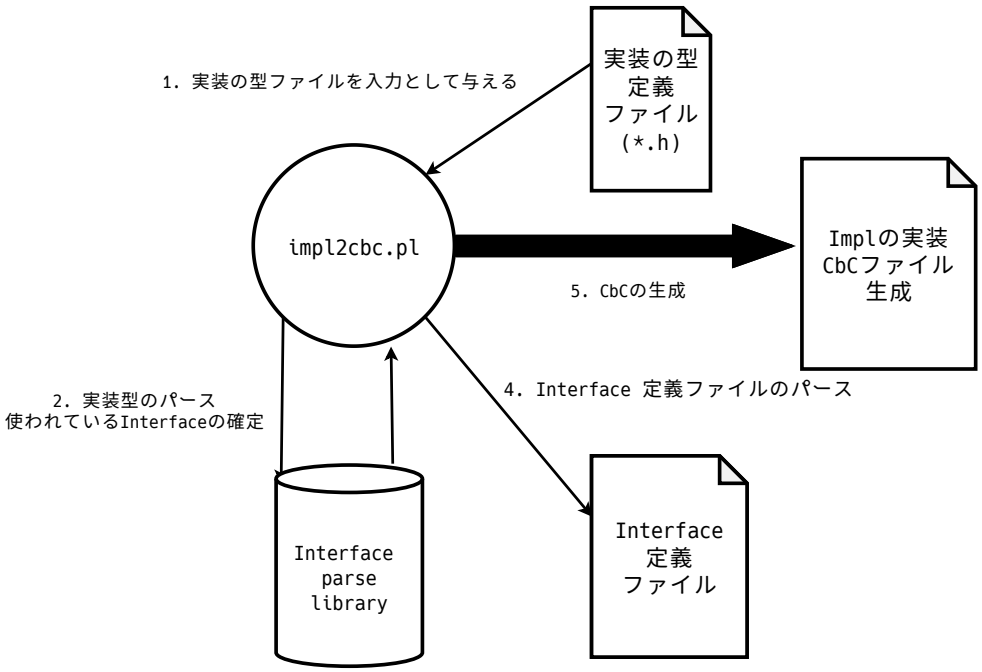

図 4.3: impl2cbc の処理の流れ

4.4.1　雛形生成の手法

Interface では入力の引数が Impl と揃っている必要があるが、 第一引数は実装自身のインスタンスがくる制約となっている。実装自身の型は、Interface 定義時には不定である。その為、 GearsOS では Interface の API の宣言時にデフォルト型変数 `Impl` を実装の型として利用する。デフォルト型 `Impl` を各実装の型に置換することで自動生成が可能となる。

実装すべき CodeGear は Interface と Impl 側の型を見れば定義されている。`__code` で宣言されているものを逐次生成すればよいが、 継続として呼び出される CodeGear は具体的な実装を持たない。GearsOS で使われている Interface には概ね次の継続である `next` が登録されている。`next` そのものは Interface を呼び出す際に、入力として与える。その為各 Interface に入力として与えられた `next` を保存する場所は存在するが、 next そのものの独自実装は各 Interface は所持しない。したがってこれを Interface の実装側で明示的に実装することはできない。雛形生成の際に、入力として与えられる CodeGear を生成してしまうと、プログラマに混乱をもたらしてしまう。

入力として与えられている CodeGear は、Interface に定義されている CodeGear の引数として表現されている。コードに示す例では、`whenEmpty` は入力して与えられている CodeGear である。雛形を生成する場合は、入力として与えられた CodeGear を除外して出力を行う。順序は Interface をまず出力した後に、 Impl 側を出力する。

4.4.2　コンストラクタの自動生成

雛形生成では他にコンストラクタの生成も行う。GearsOS の Interface のコンストラクタは、 メモリの確保及び各変数の初期化を行う。メモリ上に確保するのは主に Interface と Impl のそれぞれが基本となっている。Interface によっては別の DataGear を内包しているものがある。その場合は別の DataGear の初期化もコンストラクタ内で行う必要があるが、 自動生成コマンドではそこまでの解析は行わない。

コンストラクタのメンバ変数はデフォルトでは変数は 0、ポインタの場合は NULL で初期化するように生成する。このスクリプトで生成されたコンストラクタを使う場合、CbC ファイルから該当する部分を削除すると、`generate_stub.pl` 内でも自動的に生成される。自動生成機能を作成すると 1CbC ファイルあたりの記述量が減る利点がある。

明示的にコンストラクタが書かれていた場合は、Perl スクリプト内での自動生成は実行しないように実装した。これはオブジェクト指向言語のオーバーライドに相当する機能と言える。現状の GearsOS で使われているコンストラクタは、 基本は `struct Context*` 型の変数のみを引数で要求している。しかしオブジェクトを識別するために ID を実装側に埋め込みたい場合など、 コンストラクタ経由で値を代入したいケースが存在する。この場合はコンストラクタの引数を増やす必要や、 受け取った値をインスタンスのメンバに書き込む必要がある。具体的にどの値を書き込めば良いのかまでは Perl スクリプトでは

判定することができない。このような細かな調整をする場合は、 generate_stub.pl 側での自動生成はせずに、 雛形生成されたコンストラクタを変更すれば良い。あくまで雛形生成スクリプトはプログラマ支援であるため、 いくつかの手動での実装は許容している。

4.5 GearsOS の Interface の構文の改良

GearsOS の Interface では、 従来は DataGear と CodeGear を分離して記述していた。CodeGear の入出力を DataGear として列挙する必要があった。CodeGear の入出力として__code() の間に記述した DataGear の一覧と、Interface 上部で記述した DataGear の集合が一致している必要がある。

従来の分離している記法の場合、 この DataGear の宣言が一致していないケースが多々発生した。また Interface の入力としての DataGear ではなく、 フィールド変数として DataGear を使うようなプログラミングスタイルを取ってしまうケースも見られた。GearsOS では、 DataGear やフィールド変数をオブジェクトに格納したい場合、 Interface 側ではなく Impl 側に変数を保存する必要がある。Interface 側に記述してしまう原因は複数考えられる。GearsOS のプログラミングスタイルに慣れていないことも考えられるが、構文によるところも考えられる。CodeGear と DataGear は Interface の場合は密接な関係性にあるが、分離して記述してしまうと「DataGear の集合」と「CodeGear の集合」を別個で捉えてしまう。あくまで Interface で定義する CodeGear と DataGear は Interface の API である。これをユーザーに強く意識させる必要がある。

golang にも Interface の機能が実装されている。golang の場合は Interface は関数の宣言部分のみを記述するルールになっている。変数名は含まれていても含まなくても問題ない。

ソースコード 4.2: golang の interface 宣言

```
1 type geometry interface {
2     area() float64
3     perim() float64
4 }
```

4.6 Implement の型をいれたことによる間違った Gears プログラミング

Implement の型を導入したが、 GearsOS のプログラミングをするにつれていくつかの間違ったパターンがあることがわかった。自動生成される StubCodeGear は、 goto meta から遷移するのが前提であるため、引数を Context から取り出す必要がある。Context か

ら取り出す場合は、 実装している Interface に対応している置き場所からデータを取り出す。この置き場所は data 配列であり、 配列の添え字は enum Data と対応している。また各 CodeGear から goto する際に、 遷移先の Interface に値を書き込みに行く。

Interface で定義した CodeGear と対応している Implement の CodeGear の場合はこのデータの取り出し方で問題はない。しかし Implement の CodeGear から内部で goto する CodeGear の場合は事情が異なる。内部で goto する CodeGear は、Java などのプライベートメソッドのように使うことを想定している。この CodeGear のことを private CodeGear と呼ぶ。privateCodeGear に goto する場合、 goto 元の CodeGear からは goto meta 経由で遷移する。goto meta が発行されると Stub Code Gear に遷移するが、現在のシステムでは Interface から値をとってくることになってしまう。

4.7 メタ計算部分の入れ替え

GearsOS では次の CodeGear に移行する前の MetaCodeGear として、 デフォルトでは__code meta が使われている。__code meta は context に含まれている CodeGear の関数ポインタを、 enum からディスパッチして次の Stub CodeGear に継続するものである。

例えばモデル検査を GearsOS で実行する場合、 通常の Stub CodeGear のほかに状態の保存などを行う必要がある。この状態の保存に関する一連の処理は明らかにメタ計算であるので、 ノーマルレベルの CodeGear ではない箇所で行いたい。ノーマルレベル以外の CodeGear で実行する場合は、 通常のコード生成だと StubCodeGear の中で行うことになる。StubCodeGear は自動生成されてしまうため、 値の取り出し以外のことを行う場合は自分で実装する必要がある。しかしモデル検査に関する処理は様々な CodeGear の後に行う必要があるため、 すべての CodeGear の Stub を静的に実装するのは煩雑である。

ノーマルレベルの CodeGear の処理の後に、StubCodeGear 以外の Meta Code Gear を実行したい。Stub Code Gear に直ちに遷移してしまう__code meta 以外の Meta CodeGear に、 特定の CodeGear の計算が終わったら遷移したい。このためには、特定の CodeGear の遷移先の MetaCodeGear をユーザーが定義できる API が必要となる。この API を実装すると、ユーザーが柔軟にメタ計算を選択することが可能となる。

GearsOS のビルドシステムの API として meta.pm を作製した。これは Perl のモジュールファイルとして実装した。meta.pm は Perl で実装された GearsOS のトランスコンパイラである generate_stub.pl から呼び出される。meta.pm の中のサブルーチンである replaceMeta に変更対象の CodeGear と変更先の MetaCodeGear への goto を記述する。ユーザーは meta.pm の Perl ファイルを API として GearsOS のトランスコンパイラにアクセスすることが可能となる。

具体的な使用例をコード 4.3 に示す。meta.pm はサブルーチン replaceMeta が返すリストの中に、特定のパターンで配列を設定する。各配列の 0 番目には、goto meta を置

換したい CodeGear の名前を示す Perl 正規表現リテラルを入れる。コード 4.3 の例では、PhilsImpl が名前に含まれる CodeGear を指定している。すべての CodeGear の goto の先を切り替える場合は qr/.*/などの正規表現を指定する。

ソースコード 4.3: meta.pm

```
1  package meta;
2  use strict;
3  use warnings;
4
5  sub replaceMeta {
6    return (
7      [qr/PhilsImpl/ => \&generateMcMeta],
8    );
9  }
10
11 sub generateMcMeta {
12   my ($context, $next) = @_;
13   return "goto mcMeta($context, $next);";
14 }
15
16 1;
```

generate_stub.pl は Gears CbC ファイルの変換時に、 CbC ファイルがあるディレクトリに meta.pm があるかを確認する。meta.pm がある場合はモジュールロードを行う。meta.pm がない場合は meta Code Gear に goto するものをデフォルト設定として使う。各 Gode Gear が goto 文を呼び出したタイミングで replaceMeta を呼び出し、 ルールにしたがって goto 文を書き換える。変換する CodeGear がルールになかった場合は、 デフォルト設定が呼び出される。

4.8　別Interfaceからの書き出しを取得する必要がある CodeGear

従来の MetaCodeGear の生成では、 別の Interface からの入力を受け取る CodeGear の Stub の生成に問題があった。具体的なこの問題が発生する例題をソースコード 4.4 に示す。

ソースコード 4.4: 別 Interface からの書き出しを取得する CodeGear の例

```
1  #interface "String.h"
2  #interface "Stack.h"
3
4  #impl "StackTest.h" for "StackTestImpl3.h"
5
6  /* 略 */
7
8  __code pop2Test(struct StackTestImpl3* stackTest, struct Stack* stack,
       __code next(...)) {
9      goto stack->pop2(pop2Test1);
10 }
```

```
11
12
13 __code pop2Test1(struct StackTestImpl3* stackTest, union Data* data,
       union Data* data1, struct Stack* stack, __code next(...)) {
14     String* str = (String*)data;
15     String* str2 = (String*)data1;
16
17     printf("%d\n", str->size);
18     printf("%d\n", str2->size);
19     goto next(...);
20 }
```

この例では pop2TestCode Gear から stack->pop2 を呼び出し、 継続として pop2Test1 を渡している。pop2Test 自体は StackTest Interface であり、 stack->pop2 の stack は Stack Interface である。例題では Stack Interface の実装は SingleLinkedStack である。SingleLinkedStack の pop2 の実装をソースコード 4.5 に示す。

ソースコード 4.5: SingleLinkedStack の pop2

```
1  __code pop2SingleLinkedStack(struct SingleLinkedStack* stack, __code next
       (union Data* data, union Data* data1, ...)) {
2      if (stack->top) {
3          data = stack->top->data;
4          stack->top = stack->top->next;
5      } else {
6          data = NULL;
7      }
8      if (stack->top) {
9          data1 = stack->top->data;
10         stack->top = stack->top->next;
11     } else {
12         data1 = NULL;
13     }
14     goto next(data, data1, ...);
15 }
```

pop2 はスタックから値を 2 つ取得する API である。pop2 の継続は next であり、 継続先に data と data1 を渡している。data、 data1 は引数で受けている union Data*型の変数であり、 それぞれ stack の中の値のポインタを代入している。この操作で stack から値を 2 つ取得している。

このコードを generate_stub.pl 経由でメタ計算を含むコードに変換する。変換した先のコードを 4.6 に示す。

ソースコード 4.6: SingleLinkedStack の pop2 のメタ計算

```
1  __code pop2SingleLinkedStack(struct Context *context,struct
       SingleLinkedStack* stack, enum Code next,union Data **O_data,union
       Data **O_data1) {
2    Data* data  __attribute__((unused))  = *O_data;
3    Data* data1  __attribute__((unused))  = *O_data1;
```

```
4       if (stack->top) {
5           data = stack->top->data;
6           stack->top = stack->top->next;
7       } else {
8           data = NULL;
9       }
10      if (stack->top) {
11          data1 = stack->top->data;
12          stack->top = stack->top->next;
13      } else {
14          data1 = NULL;
15      }
16    *O_data = data;
17    *O_data1 =  data1;
18      goto meta(context, next);
19  }
20
21
22  __code pop2SingleLinkedStack_stub(struct Context* context) {
23    SingleLinkedStack* stack = (SingleLinkedStack*)GearImpl(context, Stack,
        stack);
24    enum Code next = Gearef(context, Stack)->next;
25    Data** O_data = &Gearef(context, Stack)->data;
26    Data** O_data1 = &Gearef(context, Stack)->data1;
27    goto pop2SingleLinkedStack(context, stack, next, O_data, O_data1);
28  }
```

実際は next は goto meta に変換されてしまう。data、data1 は goto meta の前にポインタ変数 O_data が指す値にそれぞれ書き込まれる。O_data は pop2 の Stub CodeGear である pop2SingleLinkedStack_stub を見るとなんであるかが分かる。つまり O_data は context 中に含まれている Stack Interface のデータ保管場所にある変数 data のアドレスである。

当初 Perl スクリプトが生成した pop2Test1 の stub CodeGear はソースコード 4.7 のものである。CodeGear 間で処理されるデータの流れの概要図を図 4.4 に示す。

ソースコード 4.7: 生成された Stub

```
1   __code pop2Test1StackTestImpl3_stub(struct Context* context) {
2     StackTestImpl3* stackTest = (StackTestImpl3*)GearImpl(context,
        StackTest, stackTest);
3     Data* data = Gearef(context, StackTest)->data;
4     Data* data1 = Gearef(context, StackTest)->data1;
5     Stack* stack = Gearef(context, StackTest)->stack;
6     enum Code next = Gearef(context, StackTest)->next;
7     goto pop2Test1StackTestImpl3(context, stackTest, data, data1, stack,
        next);
8   }
```

__code pop2Test で遷移する先の CodeGear は StackInterface であり、 呼び出している API は pop2 である。取得した API は GearsOS の Interface の処理ルールにより、Context 中の Stack Interface のデータ格納場所に書き込まれる。しかしソースコード 4.7 の例で

は `Gearef(context, StackTest)` で Context 中の `StackTest` Interface の data の置き場所から値を取得している。これでは pop2 でせっかく取り出した値を取得できない。

ここで必要となってくるのは、 呼び出し元の Stack Interface からの値の取得である。どの Interface から呼び出されているかは、 コンパイルタイムには確定できるので Perl のトランスコンパイラで Stub Code を生成したい。

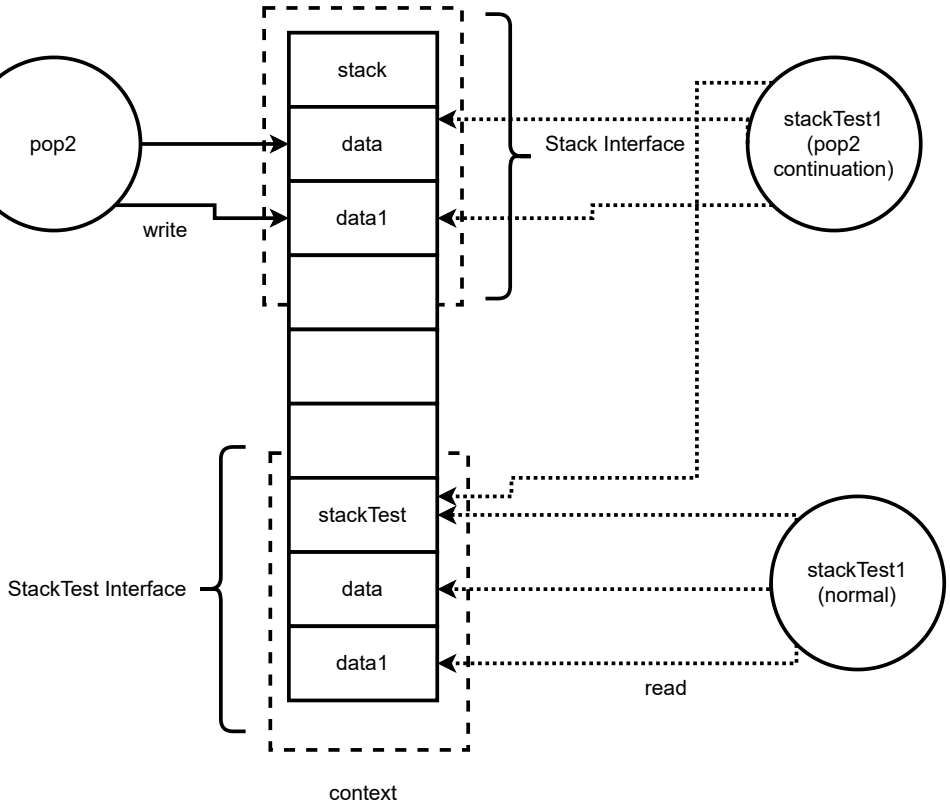

図 4.4: stackTest1 の stub の概要

別 Interface から値を取得するには別の出力がある CodeGear の継続で渡された CodeGear をまず確定させる。今回の例では `pop2Test1` が該当する。この CodeGear の入力の値と、出力がある CodeGear の出力を見比べ、 出力をマッピングすれば良い。Stack Interface の pop2 は data と data1 に値を書き込む。pop2Test1 の引数は data, data1, stack であるので、前 2 つに pop2 の出力を代入したい。

Context から値を取り出すのはメタ計算である Stub CodeGear で行われる。別 Interface から値を取り出そうとする場合、 すでに Perl トランスコンパイラが生成している Stub を書き換えてしまう方法も取れる。しかし StubCodeGear そのものを、 別 Interface から値を取り出すように書き換えてはいけない。これは別 Interface の継続として渡されるケー

スと、 次の goto 先として遷移するケースがあるためである。前者のみの場合は書き換えで問題ないが、 後者のケースで書き換えを行ってしまうと Stub で値を取り出す先が異なってしまう。どのような呼び出し方をしても対応できるようにするには工夫が必要となる。

GearsOS では継続として渡す場合や、 次の goto 文で遷移する先の CodeGear はノーマルレベルでは enum の番号として表現されていた。enum が降られる CodeGear は、厳密には CodeGear そのものではなく Stub CodeGear に対して降られる。StubCodeGear を実装した分だけ enum の番号が降られるため、 `goto meta` で遷移する際に enum の番号さえ合わせれば独自定義の Stub に継続させることが可能である。別 Interface から値を取り出したいケースの場合、 取り出してくる先の Interface と呼び出し元の CodeGear が確定したタイミングで別の StubCodeGear を生成する。呼び出し元の CodeGear が継続として渡す StubCodeGear の enum を、独自定義した enum に差し替えることでこの問題は解決する。この機能を Perl のトランスコンパイラである `generate_stub.pl` に導入した。

4.8.1　実装の手法

第5章　まとめ

5.1　総括

5.2　今後の課題

5.2.1　hogehoge

謝辞

ホゲ様，フガ様ありがとうございます

参考文献

[1] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system, 2010.

[2] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel, 2009.

[3] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. pp. 1–16, 2016.

[4] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. pp. 18–37, 2015.

[5] Ulf Norell. Dependently typed programming in agda. pp. 1–2, 2009.

[6] the coq proof assistant. *https://coq.inria.fr/*.

[7] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[8] GNU Compiler Collection (GCC) Internals. *http://gcc.gnu.org/onlinedocs/gccint/*.

[9] 大城信康, 河野真治. Continuationbasedc の gcc4.6 上の実装について. 第 53 回プログラミング・シンポジウム予稿集, Vol. 2012, pp. 69–78, jan 2012.

[10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[12] 外間政尊, 河野真治. 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, Jan 2019.

[13] 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`.

[14] Raspberry Pi. https://www.raspberrypi.org.

[15] Babel. `https://babeljs.io/`.

[16] Eclipse jdt language server. `https://github.com/eclipse/eclipse.jdt.ls`.

[17] yaohaizh. Add unimplemented methods code action.

[18] josharian/impl. `https://github.com/josharian/impl`.

[19] golang. golang/vscode-go.

[20] Zhiyi Wang. xv6-rpi. `https://code.google.com/archive/p/xv6-rpi/`, 2013.

[21] 坂本昂弘, 桃原優, 河野真治. 継続を用いた x.v6 kernel の書き換え. 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), No. 4, may 2019.

[22] 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), May 2016.

[23] J. Lions. *Lions' Commentary on UNIX 6th Edition with Source Code*. Computer classics revisited. Peer-to-Peer Communications, 1996.

[24] Eugenio Moggi. Notions of computation and monads, July 1991.

付 録A　研究会業績

A-1　研究会発表資料