

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

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

研究関連論文業績	iii
第1章 OSとアプリケーションの信頼性	5
第2章 Continuation Based C	8
第3章 GearsOS	9
3.1 pmake	9
第4章 GearsOSのトランスコンパイラ	11
4.1 トランスコンパイラ	11
4.2 GearsCbCのInterfaceの実装時の問題	14
4.3 Interfaceを満たすコード生成の他言語の対応状況	14
4.4 GearsOSでのInterfaceを満たすCbCの雛形生成	15
4.5 GearsOSのInterfaceの構文の改良	17
4.6 メタ計算部分の入れ替え	17
第5章 まとめ	20
5.1 総括	20
5.2 今後の課題	20
5.2.1 hogehoge	20
謝辞	20
謝辞	21
参考文献	22
付録	23
付録A 研究会業績	24
A-1 研究会発表資料	24

目 次

3.1	pmake.pl の処理フロー	10
4.1	generate_sub.pl を使ったトランスコンパイル	13
4.2	generate_context.pl を使ったファイル生成	13
4.3	impl2cbc の処理の流れ	16

表 目 次

ソースコード目次

4.1	CMakeList.txt 内での Perl の実行部分	12
4.2	golang の interface 宣言	17
4.3	meta.pm	18

第1章 OSとアプリケーションの信頼性

コンピュータ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 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られることができる。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。これはプログラマー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を行う為には、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とは、C言語から関数呼び出しとループ処理を取り除いた言語である。Cの下位言語と言え、C言語とアセンブラの中間のような言語として利用することが出来る。

第3章 GearsOS

3.1 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[13] 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 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クスクリプトを使う必

要がある。これらのリンクスクリプトも 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 3.1 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。

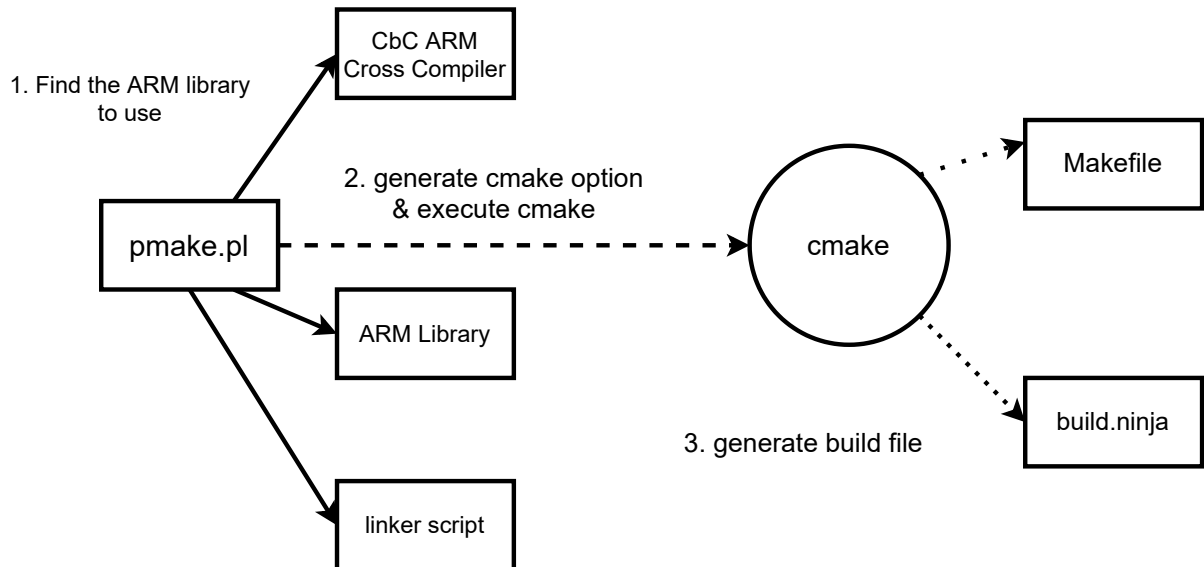


図 3.1: `pmake.pl` の処理フロー

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[14]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。GearsOS ではビルドツールに CMake を利用している。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。

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}"
6     "${_MULTI_VALUE_ARGS}" ${ARGN} )
7
8   set ( _Gears_CSOURCES)
9   foreach(i ${_Gears_SOURCES})
10    if (${i} MATCHES "\\\.cbc")
11      string(REGEX REPLACE "(.*)\.cbc" "c/\\1.c" j ${i})
12      add_custom_command (
13        OUTPUT    ${j}
14        DEPENDS   ${i}
15        COMMAND  "perl" "generate_stub.pl" "-o" ${j} ${i}
16      )
17    elseif (${i} MATCHES "\\\.cu")
18      string(REGEX REPLACE "(.*)\.cu" "c/\\1.ptx" j ${i})
19      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)

```

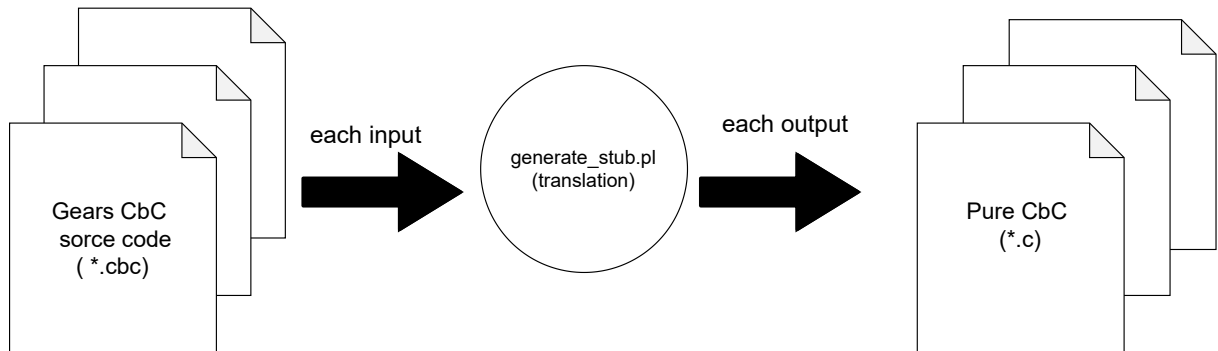


図 4.1: generate_stub.pl を使ったトランスコンパイル

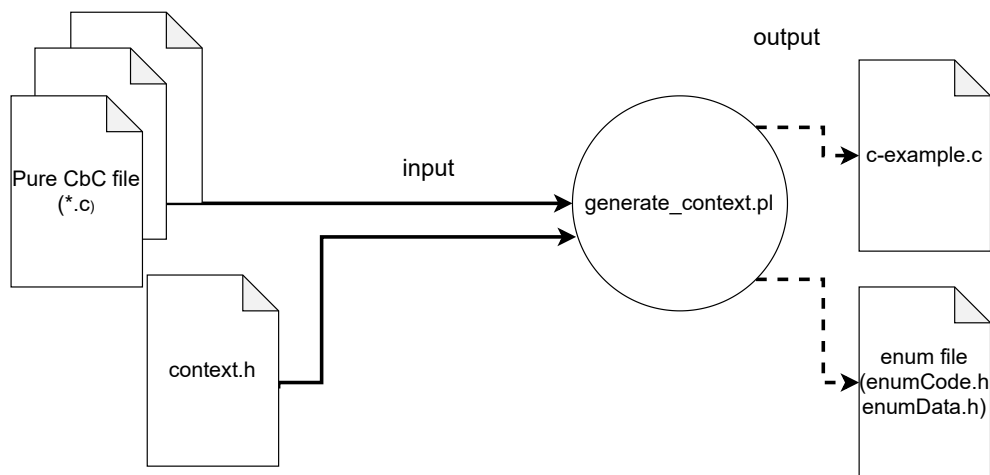


図 4.2: generate_context.pl を使ったファイル生成

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
- 実装を忘れていた 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、これら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[15] 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。[16] 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[17] 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`[18] 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れる。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。

Interface では入力引数が 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。

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。

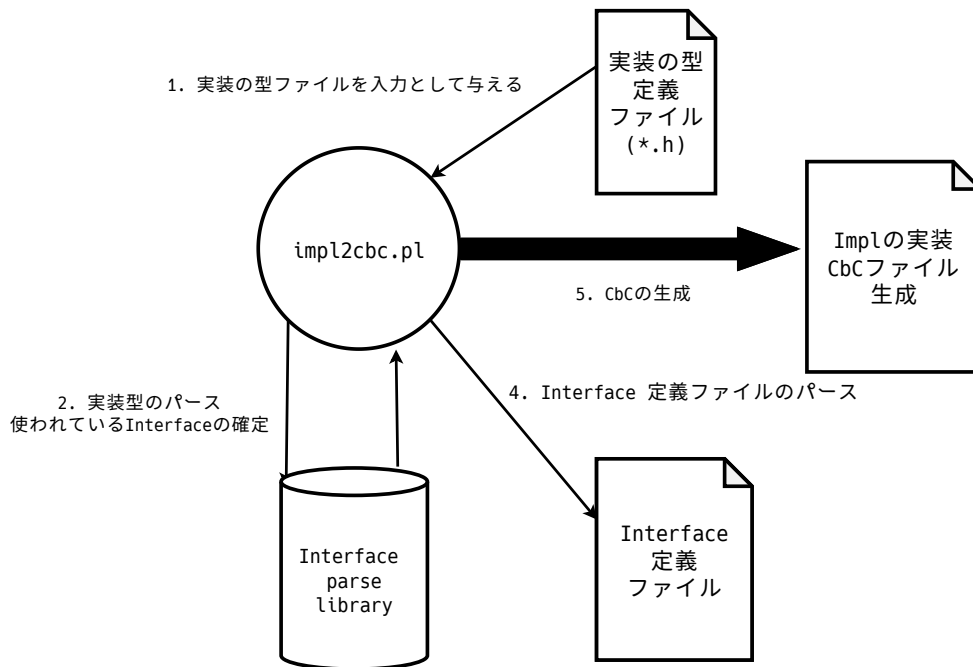


図 4.3: impl2cbc の処理の流れ

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 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` 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 Code 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。

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 Nikolai 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] Raspberry Pi. <https://www.raspberrypi.org>.
- [14] Babel. <https://babeljs.io/>.
- [15] Eclipse jdt language server. <https://github.com/eclipse/eclipse.jdt.ls>.
- [16] yaohaizh. Add unimplemented methods code action.
- [17] josharian/impl. <https://github.com/josharian/impl>.
- [18] golang. golang/vscode-go.
- [19] Zhiyi Wang. xv6-rpi. <https://code.google.com/archive/p/xv6-rpi/>, 2013.
- [20] 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.
- [21] 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.
- [22] 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>.
- [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