

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 継続を中心としたプログラミングスタイル	4
第2章 GearsOS のトランスコンパイラ	5
2.1 トランスコンパイラ	5
2.2 GearsCbC の雛形生成	6
2.3 GearsOS の Interface の構文の改良	8
第3章 まとめ	9
3.1 総括	9
3.2 今後の課題	9
3.2.1 hogehoge	9
謝辞	9
謝辞	10
参考文献	11
付録	11
付録 A 研究会業績	12
A-1 研究会発表資料	12

# 图 目 次

# 表 目 次

# 第1章 継続を中心としたプログラミングスタイル

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。

アプリケーションは通常特定のプログラミング言語で実装されている。このプログラミング言語自身の信頼性は高く保証される必要がある。また、実際にアプリケーションを動作させる OS も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 や Coq 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、型と命題が、プログラムと証明が対応する。

## 第2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。

### 2.1 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[1]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。

## 2.2 GearsCbC の雛形生成

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。

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[2] 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。[3] 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[4] 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[5]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。

GearsOSでも同様のプログラマ支援ツールを導入したい。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スクリプトをGearsOSに導入した。

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。

雛形生成では他にコンストラクタの生成も行う。コンストラクタはデフォルトでは変数は0、またはNULLで初期化される。このスクリプトで生成されたコンストラクタはgenerate\_stub.pl内でも自動的に生成することが可能である。そのため、オーバーライ

ドする必要がない場合は削除しても問題なく動作するようになっている。

## 2.3 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 2.1: golang の interface 宣言

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

## 第3章 まとめ

### 3.1 総括

### 3.2 今後の課題

#### 3.2.1 hogehoge

# 謝辞

ホゲ様，フガ様ありがとうございます

## 参考文献

- [1] Babel. <https://babeljs.io/>.
- [2] Eclipse jdt language server. <https://github.com/eclipse/eclipse.jdt.ls>.
- [3] yaohaizh. Add unimplemented methods code action.
- [4] josharian/impl. <https://github.com/josharian/impl>.
- [5] golang. golang/vscode-go.
- [6] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.
- [7] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.
- [8] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 99–110, New York, NY, USA, 2010. ACM.
- [9] 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. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. ACM.
- [10] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 1–16, Berkeley, CA, USA, 2016. USENIX Association.

# 付 録 A 研究会業績

## A-1 研究会発表資料