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

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。

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。

- `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
- `generate_context.pl`
  - 生成した CbC ファイルを解析し、使われている CodeGear、DataGear を確定する
  - これらの情報をもとに Context 及び Context 関係の初期化ルーチン、API を作成する

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 に記述し、 移行は CMake のビルドに移譲する。CMake は Makefile や Ninja file を生成し実際にビルドを行うのは make や ninja-build となっている。

2.2　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。

2.3　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、Interfce 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。

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

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。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。

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。

2.5 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 2.1: golang の interface 宣言

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

2.6 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。この CodeGear 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。

このメタ計算部分を独自で定義した CodeGear に差し替えたいケースが存在する。

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