2022年度 卒業論文

Bachelor's Thesis

Gears Debuggerの開発

Development of Gears Debugger

琉球大学工学部工学科知能情報コース

185703H　松岡 隆斗

指導教員　河野 真治

要旨

プログラミングにおいてエラー・バグは付き物であり、その発見・修正が重要である。エラー・バグの発見を支援するツールとしてデバッガがある。デバッガを使うことで変数の値の確認や、スタックトレースによる関数呼び出しの確認を行うことでエラー・バグの発見を行う。

本研究室では、Continuation Based C(CbC) を用いて信頼性と拡張性を両立させた OS である GearsOS の開発を行っている。GearsOS の開発においてもエラー・バグの発見のためにデバッガを用いてデバッグを行っている。しかし、GearsOS は変数の格納方法が複雑なため煩雑な記述を行わなければならない特性や、スタックを持たないためにスタックトレースが使えないなどの特性を持つ。これらの特性により既存デバッガを用いたデバッグが難しく、エラー・バグの特定が難しいという問題を抱えている。

本研究では GearsOS 特化のデバッガを開発することでエラー・バグの発見および修正のコストを下げるとともにさらなる GearsOS の信頼性の向上を目指す。

Abstract

Errors and bugs are inevitable in programming, and it is important to find and fix them. A debugger is a tool that supports the detection of errors and bugs. By using a debugger, you can check the value of variables and check function calls with stack traces.

In this laboratory, we are developing GearsOS, which is an OS with both reliability and scalability using Continuation Based C (CbC). The debugger is also used in the development of GearsOS to find errors and bugs. However, GearsOS has some characteristics such as a complicated method of storing variables and the inability to use stack tracing.

This research aims to develop a GearsOS-specific debugger to reduce the cost of finding and fixing errors and bugs, and to further improve the reliability of GearsOS.

目次

第 1 章 GearsOS におけるデバッグ 1

第 2 章 Continuation Based C 3

2.1 CodeGear 3

2.2 DataGear 4

2.3 メタ計算 4

2.4 MetaCodeGear 4

2.5 MetaDataGear 5

第 3 章 GearsOS 6

3.1 Context 6

3.2 Worker 8

3.3 TaskManager 9

3.4 TaskQueue 9

3.5 Interface 9

第 4 章 GearsOS におけるモデル検査 10

4.1 stateDB 10

第 5 章 既存のデバッガについて 11

5.1 代表的なデバッガの機能 11

5.2 ステップ実行 11

5.3 変数の値の確認 11

5.4 ソースコードの表示 12

5.5 ブレークポイント 12

5.6 スタックトレース 12

第 6 章 GearsDebugger 13
6.1 GearsDebugger の持つ機能 13
6.1.1 ユーザーインプット機能 13
6.1.2 CodeGear 単位でのデバッグ 14
6.1.3 DataGear の表示 14
6.2 DebugWorker 14
6.3 DebugTaskManager 15
6.4 debugMeta 17
6.5 meta.pm 17

第 7 章 結論 19
7.1 今後の課題 19

参考文献 21
7.2 Mindmap 22

図目次

2.1 gotoによる軽量継続を用いたCodeGear遷移 3
2.2 MetaCodeGearを含めたCodeGearの遷移 4

7.1 研究遂行および論文執筆時に作成したMindmap 22

第 1 章

GearsOS におけるデバッグ

OS は CPU の割り当てやスケジュール、記憶容量へのアクセスなどアプリケーションが動作する上で土台となる重要なソフトウェアである。そのため OS の信頼性は高く保証されている必要がある。信頼性を保証する方法として形式検証が挙げられる。形式検証には定理証明とモデル検査があり、定理証明は Agda[1] などの定理証明支援系を用いて数学的にプログラムの信頼性を保証する。モデル検査はプログラムが特定の状態から遷移しうる全ての状態を数え上げ、仕様を満たしているかどうかを検査することでプログラムの信頼性を保証する。

本研究室では定理証明やモデル検査を用いて信頼性の保証を行う GearsOS の開発を行っている。GearsOS は Continuation Based C(CbC) を用いて記述した OS で、CbC は本研究室で開発している言語で、CodeGear というプログラム単位で記述・遷移する C 言語の下位言語である。CodeGear 間の遷移は通常の関数呼び出しではなく、goto 文によって行われるため、呼び出し元へ戻らない。そのためプログラムの記述をそのまま状態遷移として落とし込むことができ、これにより定理証明やモデル検査が可能である。

GearsOS は定理証明やモデル検査により信頼性の保証を行っているが、これらの定理証明やモデル検査自体を自分がどう証明したのかを確かめたり、GearsOS 上の例題のデバッグしたいということがある。現状このような場合には GDB[2] や LLDB[3] などの既存のデバッガを用いて変数の値の確認や、スタックトレースによる関数呼び出しの確認を行うことでデバッグを行っている。しかし、GearsOS は変数の格納方法が複雑なため煩雑な記述を行わなければならない特性や、スタックを持たないためにスタックトレースが使えないなどの特性を持つ。これらの特性によりプログラムの遷移の流れや、変数の値の変化など既存デバッガを用いたデバッグが難しいという問題がある。

本研究では GearsOS 特化のデバッガを開発することでエラー・バグの発見および修正

のコストを下げるとともにさらなる GearsOS の信頼性の向上を目指す。

第 2 章

Continuation Based C

Continuation Based C(CbC)[4] とは、本研究室で開発している軽量継続を導入した C 言語の下位言語である。軽量継続とは Scheme の call/cc などの環境を持つ継続とは異なり、スタックが無く環境を持たない継続である。そのため call/cc よりも軽量であることから軽量継続と呼ばれる。

2.1 CodeGear

CbC は従来のプログラミング言語における関数ではなく CodeGear という単位でプログラムを記述する。プログラムの遷移は CodeGear から CodeGear への遷移によって実現され、軽量継続である goto 文を用いて行われる。CbC は軽量継続により遷移を行うため、次の CodeGear へ継続すると継続前の CodeGear に戻ることができない。goto による軽量継続を用いた CodeGear の継続の流れが図 2.1 である。

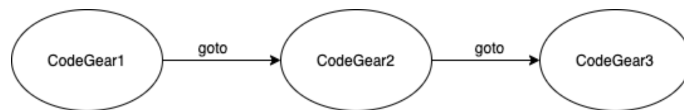

図 2.1 goto による軽量継続を用いた CodeGear 遷移

TODO: 簡単な CodeGear 遷移の例を作って、記述する。

2.2 DataGear

CodeGear と同じく CbC におけるプログラミング単位の一つである。CodeGear から次の CodeGear への継続の際に、データの受け渡しに用いられるのが DataGear である。特に、CodeGear からの入力を受け取る DataGear を InputDataGear、CodeGear 実行後に CodeGear からの出力を受け取る DataGear を OutputDataGear という。

2.3 メタ計算

記述したプログラムの計算を行うためには、メモリ管理や、スレッド管理、GPU の資源管理などの OS レベルの計算を行う必要がある。この OS レベルの計算をメタ計算と呼び、ユーザーが記述したプログラムをノーマルレベルの計算と呼ぶことで 2 つの計算を区別している。

2.4 MetaCodeGear

CbC においてメタ計算を行う CodeGear を MetaCodeGear といい、特に実行したい CodeGear の直前で実行される MetaCodeGear を StubCodeGear という。ノーマルレベルから見ると CodeGear が直接 InputDataGear からデータを受け取り、プログラム実行、その後 OutputDataGear にデータを書き込んでいるように見えるが、実際には CodeGear 実行の前に MetaCodeGear が実行され、MetaCodeGear を経由して DataGear とデータの受け渡しを行っている。メタレベルの計算を MetaCodeGear が行うことで、ノーマルレベルの計算とメタレベルの計算の分離を実現している。MetaCodeGear、DataGear を含めた CodeGear の遷移を図 2.2 に示す。

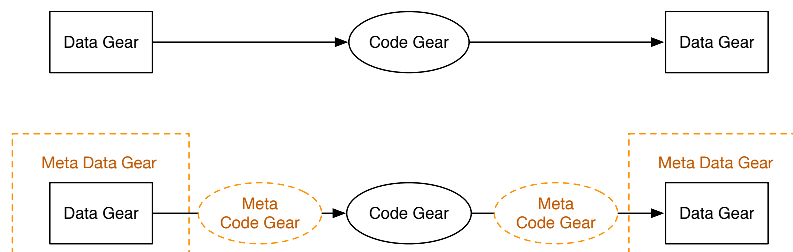

図 2.2　MetaCodeGear を含めた CodeGear の遷移

2.5 MetaDataGear

CPU や GPU の情報、計算に必要な全ての DataGear などメタ計算を行うために必要な情報を持つのが MetaDataGear である。

第 3 章

GearsOS

GearsOS は本研究室で開発している CbC を用いて信頼性と拡張性の両立を目指した OS[8] である。CbC と同様に CodeGear と DataGear を基本単位として実行し、ノーマルレベルの計算とメタレベルの計算の分離により信頼性を担保している。

GearsOS は様々な役割を持つ CodeGear と DataGear で構成されている。本章では構成の中心となっている MetaDataGear や、モジュール化の仕組みとして導入されている Interface などについて説明を行う。

3.1 Context

Context とは GearsOS の計算に必要な CodeGear や DataGear を持つ MetaDataGear であり、従来の OS におけるプロセスやスレッドに対応する。Context は使用可能な CodeGear と DataGear のリストや、TaskQueue へのポインタ、DataGear を格納するためのメモリ空間などを持っている。ソースコード 3.1 が Context の定義である。

ソースコード 3.1　Context の定義

```
struct Context {
    enum Code next;
    struct Worker* worker;
    struct TaskManager* taskManager;
    int codeNum;
    __code (**code) (struct Context*);
    union Data **data;
    struct Meta **metaDataStart;
    struct Meta **metaData;
    void* heapStart;
    void* heap;
```

```
12     long heapLimit;
13     int dataNum;
14     // task parameter
15     int idgCount; //number of waiting dataGear
16     int idg;
17     int maxIdg;
18     int odg;
19     int maxOdg;
20     int gpu; // GPU task
21     struct Context* task;
22     struct Element* taskList;
23 #ifdef USE_CUDAWorker
24     int num_exec;
25     CUmodule module;
26     CUfunction function;
27 #endif
28     /* multi dimension parameter */
29     int iterate;
30     struct Iterator* iterator;
31 };
```

ソースコード 3.16 行目の code が CodeGear を格納するための配列である。code 配列へアクセスするための index は、ソースコード 3.2 で定義される enum を用いる。この enum は GearsOS で用いる CodeGear を全て列挙しており、コンパイル時に一意な番号へと変換される。この番号と配列へ格納される CodeGear のポインタが対応しているため、特定の CodeGear の取り出しは対応した番号を index に指定することで実現する。

ソースコード 3.2　CodeGear の enum

```
1 enum Code {
2     C_checkAndSetAtomicReference,
3     C_clearSingleLinkedStack,
4     C_clearSynchronizedQueue,
5     C_code1,
6     C_code2,
7     ...
8 };
```

ソースコード 3.17 行目の data が DataGear を格納するための配列である。data 配列は union Data 型であり、これは共用体によって定義されている。ソースコード 3.3 に union Data の定義を示す。union Data 共用体の中に Atomic や CPUWorker などの DataGear を構造体として定義を行っている。これは通常の C 言語においては、struct Atomic と struct CPUWorker は当然別の型として判別される。しかし GearsOS の

Context においては Atomic も CPUWorker も DataGear として等しく扱う必要がある。そのため共用体を用いて汎用的な DataGear 型である union Data 型を定義することで任意の DataGear を一律に扱うことができる。

ソースコード 3.3 union Data の定義

```
union Data {
    struct Atomic {
        union Data* atomic;
        union Data** ptr;
        union Data* oldData;
        union Data* newData;
        enum Code checkAndSet;
        enum Code next;
        enum Code fail;
    } Atomic;

    struct AtomicReference {
    } AtomicReference;

    struct CPUWorker {
        pthread_mutex_t mutex;
        pthread_cond_t cond;
        struct Context* context;
        int id;
        int loopCounter;
    } CPUWorker;
    ...
}
```

3.2 Worker

Worker は TaskManager から Task を取り出し、Task の CodeGear の実行を行う。実行後は OutputDataGear へ書き込みを行う。Worker はまず生成時にスレッドを作成する。スレッド生成後は TaskManager の TaskQueue から Task を取得する。Task は Context の形で表現されているため、Worker の Context を Task に入れ替えて Task の次の CodeGear へと継続する。Task 実行後は OutputDataGear の書き出しを行う。Worker は CodeGear の前後で呼び出されるため、CodeGear の前後の状態を記録することが可能である。また、Worker 自体が Interface によって定義されているためコードを変更せずに Worker の切り替えが可能である。そのため GearsDebugger ではデバッグ用 Worker を定義し、デバッグ時に通常の Worker からデバッグ用 Worker へ切り替えるこ

とによって CodeGear の実行前後でデバッグ用 Worker を呼び出してデバッグを行う。

3.3 TaskManager

TaskManager は Task を実行する Worker の生成、管理、Task の送信などを行う。GearsOS における Task は Context の形で表現されており、実行する CodeGear、計算に必要な InputDataGear、計算後に書き出す OutputDataGear の格納場所などの情報を持っている。TaskManager は、CodeGear の実行に必要な InputDataGear が揃っているか確認し、揃っていなければ待ち合わせを行い、揃った場合は Task を送信し実行させる。

3.4 TaskQueue

Worker が利用する Queue であり SynchronizedQueue によって表現されている。SynchronizedQueue はマルチスレッドでもデータの一貫性を保証する Queue となっており、データ更新時に CAS(Check And Set) を行う。CAS は更新前のデータと更新後のデータを比較し、値が同じであればデータの競合がないとして更新に成功し、値が異なる場合は更新に失敗する。

3.5 Interface

GearsOS におけるモジュール化の仕組みとして Interface が導入されている。Interface は DataGear の定義と、その DataGear に対しての操作 (API) を行う CodeGear の集合である。Interface は仕様 (Interface) と実装 (Implement, Impl) の 2 つに分けて記述する。これによって API となる CodeGear の実装を変えることで、同じ CodeGear(API) だが別の処理を行うというようなことが実現できる。CodeGear としては同じであるため、呼び出し元のコードを変更することなく、処理だけ変えることが可能である。

第 4 章

GearsOS におけるモデル検査

先行研究 [5] にて GearsOS においてモデル検査を行う手法が提案されている。このモデル検査手法を確かめるための GearsOS の例題として DPPMC がある。DPPMC は食事する哲学者の問題 (Dining Philosophers Probrem) のモデル検査 (Model Checking) を行う例題である。食事する哲学者の問題は並列処理に関する問題であり、リソース共有により起こるデッドロックを抽象化・一般化した例である。モデル検査により網羅的なプログラムの実行を行うことでプログラムの状態を展開しデッドロックを調べる。

4.1 stateDB

モデル検査を行うためにプログラムで生じうる全ての状態を数え上げ記憶しておく必要があり、GearsOS においては Context の状態を数え上げることでモデル検査を実現できる。しかし Context の状態は有限状態となるとは限らず、また有限であったとしても巨大になる懸念があったため Context を十分に抽象化する必要があった。そこでメモリ領域の集合を 1 つの状態と定義することで抽象化を実現している。この状態を格納するためのデータベースが stateDB である。GearsDebugger においてもトレース状態の保存に stateDB を用いている。

第 5 章

既存のデバッガについて

デバッガとはプログラムのデバッグを行う際に用いられるソフトウェアであり、変数の値を確認したり、プログラムを 1 行単位で実行したりしながらプログラムのバグの発見・修正を行う。

5.1　代表的なデバッガの機能

既存のデバッガとして GDB[2] や LLDB[3] などが挙げられる。またこれらのデバッガの持つ代表的な機能として以下が挙げられる。

5.2　ステップ実行

プログラムを 1 ステップずつ実行させる機能である。これによりプログラムの動作を 1 つずつ確認することができる。ステップ実行には種類がありステップイン、ステップオーバー、ステップアウトなどがある。

5.3　変数の値の確認

現在の状態における変数の確認ができる機能である。変数だけでなく、構造体やクラスなどの値を確認することもできる。これを用いて任意の状態における変数の値を確認することで、プログラマーが意図した処理が正常に実行されているかどうかを確かめることができる。

5.4　ソースコードの表示

プログラムをステップ実行する際にどの行を実行しているかを表示する。

5.5　ブレークポイント

ブレークポイントはプログラムの任意の行や関数を指定することでその箇所でプログラムの実行を一時停止する機能である。ブレークポイントを設定して、変数の値の確認など行うことで変数に正しい値が格納されているかどうかなどを確かめることができる。

5.6　スタックトレース

スタックトレースとはプログラムの実行手順を確認できる機能で、関数がどのように呼び出されているのかを確認できる。これを使うことでプログラムの実行順を追いやすくなりデバッグ作業を行う上で重要な機能である。

第 6 章

GearsDebugger

本研究では、GearsOS に特化したデバッガとして GearsDebugger の開発を行った。本章では GearsDebugger の持つ機能や仕組みについて記述する。

6.1 GearsDebugger の持つ機能

GearsDebugger の持つ機能として以下が挙げられる。

6.1.1 ユーザーインプット機能

ユーザーインプット機能はユーザーが行いたいデバッグ処理を指定するための機能で、プロンプトにより対話形式でデバッグコマンドを入力することでデバッグが可能になった。デバッグコマンドは後述する next や pd コマンドなどが存在する。ソースコード 6.1 の 7 行目に (Gears Debugger) があるが、これがユーザーへ入力を促すプロンプトである。

ソースコード 6.1 ユーザーへ入力を促すプロンプト

```
1 $ ./Debug_hello_world
2 cpus:       1
3 gpus:       0
4 length:     102400
5 length/task:    12800
6
7 (Gears Debugger)
```

6.1.2 CodeGear 単位でのデバッグ

GearsOS は CodeGear と DataGear を単位として実行される。そのためデバッグ時においては 1 つの CodeGear を実行した後の情報を見ることで、その CodeGear によってどの DataGear がどのように変化したのかを見ることができる。そのため 1 つの CodeGear 実行後に debugMeta を呼ぶことにより CodeGear 単位でのデバッグを実現している。また、次の CodeGear を実行したい場合は next もしくは n コマンドを入力することで次の CodeGear へと遷移する。

6.1.3 DataGear の表示

CbC のデータ単位であり、GearsOS の基本単位である DataGear の表示が可能となった。表示する際には print DataGear を意味する pd コマンドを入力する。また複数存在する DataGear の指定に対応するため、オプションで DataGear 名を指定することで特定の DataGear の表示が出来る。表示の際には全ての DataGear の参照ができる context を経由して参照を行う。実際に DPPMC で使用される DataGear である Phils の表示結果をソースコード 6.2 に示す。

ソースコード 6.2 pd コマンドを用いた Phils の表示結果

```
1  (Gears Debugger) pd Phils
2  DataGear Name: Phils
3  DataGear Address: 0x7fcdb4c00000
4  Phils Address: 0x7fccb4c00614
5  putdown_rfork: 0
6  thinking: 0
7  pickup_lfork: 0
8  pickup_rfork: 0
9  eating: 0
10 next: 10
```

6.2 DebugWorker

Debug 用の Worker として DebugWorker の作成を行った。通常の Worker と異なる部分として、debugMeta の実装が挙げられる。ソースコード 6.3 が debugMeta の定義の一部である。ソースコード 6.3 の 1 行目の__ncode は、ユーザーが定義できる MetaCodeGear である。通常の CodeGear の定義はソースコード 6.4 の 1 行目のよう

に__code を用いて定義を行う。この__code を使った定義の場合はトランスパイラによる変換時にノーマルレベルの CodeGear だと判断され、メタ計算を行う StubCodeGear がトランスパイラによって生成される。しかし__ncode を使った CodeGear の定義はトランスパイラによって MetaCodeGear と判断され、Stub の生成などがされない。

GearsOS はノーマルレベルとメタレベルの分離を実現するため、ノーマルレベルの CodeGear から MetaDataGear である Context への参照を行わない。GearsDebugger においてはデバッグ情報を Context から取得するため、デバッグ処理を行う CodeGear はノーマルレベルの CodeGear ではなく MetaCodeGear である必要がある。そのため__ncode を用いてデバッグ用 MetaCodeGear である debugMeta を作成している。

ソースコード 6.3　__ncode を用いた debugMeta の定義

```
1  __ncode debugMeta(struct Context* context, enum Code next) {
2      context->next = next; // remember next Code Gear
3      struct DebugWorker* debugWorker = (struct DebugWorker*) context->
       worker->worker;
4      StateNode st;
5      StateDB out = &st;
6      struct Element* list = NULL;
7      struct DebugTaskManagerImpl* debugTaskManagerImpl = (struct
       DebugTaskManagerImpl *)debugWorker->taskManager->taskManager;
8
9      ...
10
11     if (strcmp(command_arr[0], "next") == 0 || strcmp(command_arr[0],
       "n") == 0) {
12         goto meta(context, context->next);
13     }
14 }
```

ソースコード 6.4　通常の__code を用いた CodeGear の定義

```
1  __code hello(struct HelloImpl* hello, __code next(...)) {
2      printf("Hello, World;");
3      goto next(...);
4  }
```

6.3 DebugTaskManager

Debug 用の TaskManager として DebugTaskManager の作成を行った。通常は 3.3 で触れた TaskManager を用いるが、GearsDebugger においてデバッグする場合は Debug-

TaskManager を用いる。通常の TaskManager との違いとしてはまず Worker 生成部が挙げられる。デバッグ時には通常用いられる Worker ではなく、DebugWorker を用いる。この Worker の生成は TaskManager で行うため、DebugTaskManager においては Worker 生成時に DebugWorker を生成する。ソースコード 6.5 が通常の TaskManager 内の Worker 生成部、ソースコード 6.6 が DebugTaskManager 内の Worker 生成部である。これを見ると、ソースコード 6.5 の 6, 11 行目の通常の Worker 生成を行う関数である createCPUWorker が、ソースコード 6.6 の 6, 11 行目のように createDebugWorker に変わっていることがわかる。

ソースコード 6.5　TaskManager における Worker 生成

```
1  void createWorkers(struct Context* context, TaskManagerImpl*
       taskManager) {
2      ...
3  #ifdef USE_CUDAWorker
4          taskManager->workers[i] = (Worker*)createCUDAWorker(context, i
       , queue,0);
5  #else
6          taskManager->workers[i] = (Worker*)createCPUWorker(context, i,
        queue);
7  #endif
8      }
9      for (;i<taskManager->maxCPU;i++) {
10         Queue* queue = createSynchronizedQueue(context);
11         taskManager->workers[i] = (Worker*)createCPUWorker(context, i,
        queue);
12     }
13 }
```

ソースコード 6.6　DebugTaskManager における Worker 生成

```
1  void createWorkers(struct Context* context, DebugTaskManagerImpl*
       taskManager) {
2      ...
3  #ifdef USE_CUDAWorker
4          taskManager->workers[i] = (Worker*)createCUDAWorker(context, i
       , queue,0);
5  #else
6          taskManager->workers[i] = (Worker*)createDebugWorker(context,
       i, queue);
7  #endif
8      }
9      for (;i<taskManager->maxCPU;i++) {
10         Queue* queue = createSynchronizedQueue(context);
```

```
11          taskManager->workers[i] = (Worker*)createDebugWorker(context,
        i, queue);
12      }
13  }
```

2 つ目の違いとしては、DebugTaskManager はメモリ状態を保存するための DB である stateDB の情報を持つ。

6.4 debugMeta

6.2 で DebugWorker 内に debugMeta を作成したが、その debugMeta の行う処理について記述していく。debugMeta はデバッグ処理におけるユーザーインプット部分を担う MetaCodeGear である。GearsDebugger は 1 つの CodeGear の実行後に DataGear の表示などデバッグ情報を見たい。そのため、各 CodeGear の呼び出し後に debugMeta を呼び出すことでユーザーに入力を促す。これによって各 CodeGear 実行後にデバッグを行うことができる。

6.5 meta.pm

6.4 にて各 CodeGear の呼び出し後に debugMeta を呼び出すと説明したが、その呼び出し方法として meta.pm[12] という GearsOS のビルドシステムの API を用いている。通常は CodeGear 実行後、次の CodeGear の StubCodeGear へと遷移するが、meta.pm を使うことで CodeGear 実行後の遷移先を特定の MetaCodeGear へと変更することができる。ソースコード 6.7 に meta.pm を示す。ソースコード 6.7 の 7 行目に qr/HelloImpl/ と正規表現がある。トランスパイラはこの正規表現にマッチした CodeGear の goto 先を切り替えている。例でいうと、HelloImpl という文字列を含む CodeGear があった場合、ソースコード 6.7 13 行目のサブルーチン generatedebugMeta を呼び出し、goto 先を debugMeta へと切り替えている。

ソースコード 6.7 meta.pm

```
1  package meta;
2  use strict;
3  use warnings;
4
5  sub replaceMeta {
6    return (
7      [qr/HelloImpl/ => \&generatedebugMeta],
```

```
8    );
9  }
10
11 #my ($currentCodeGearName, $context, $next) = @_;
12
13 sub generatedebugMeta {
14   my ($context, $next) = @_;
15   return "goto debugMeta($context, $next);";
16 }
17 1;
```

第 7 章

結論

本研究では、GearsOS に特化したデバッガの作成を行った。CodeGear 単位で遷移を行う GearsOS のデバッグを行うために、CodeGear 単位でプログラムを一時停止させ、デバッグコマンドを入力することで DataGear の値を見ることが可能となった。また、デバッグ用の Worker と TaskManager を作成し、通常の Worker と TaskManager と変更することで、様々な例題を対象として DataGear の値を確認することが可能となった。

7.1 今後の課題

今後の課題としてデバッガ呼び出し手順の煩雑さが挙げられる。現状でデバッガを呼び出すには通常の Worker と TaskManager からデバッグ用の Worker、TaskManager への入れ替えや、meta.pm の記述・配置、debugMeta のプロトタイプ宣言の記述を行い、再ビルドを行う必要がある。Worker や TaskManager に関しては、通常の Worker や TaskManager をデバッグ用に拡張させ、例題実行時にオプションとしてデバッグオプションを指定することで振る舞いを通常の振る舞いからデバッグ用へと分岐させることが可能だと考えられる。また meta.pm や debugMeta に関しては GearsOS のビルドシステムである generate_stub.pl や generate_context.pl 側でデバッグオプションの有無により継続を先を変更することが出来れば、通常の例題を再ビルドさせることなく、オプションの有無によって通常実行とデバッグ実行を切り替えられると考える。

謝辞

本研究の遂行、本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に心より感謝致します。そして、 共に研究を行い暖かな気遣いと励ましをもって支えてくれた並列信頼研究室の全てのメンバーに感謝致します。最後に、有意義な時間を共に過ごした知能情報コースの学友、並びに物心両面で支えてくれた家族に深く感謝致します。

参考文献

[1] Ulf Norell. Dependently typed programming in agda. pp. 1
UTF20132, 2009

[2] https://www.gnu.org/software/gdb/

[3] https://lldb.llvm.org/

[4] 河野真治. 継続を持つ c の下位言語によるシステム記述. 日本ソフトウェア科学会第 17 回大会, 2000.

[5] 東恩納琢偉. GearsOS におけるモデル検査を実現する手法について. 琉球大学大学院理工学研究科情報工学専攻修士論文, March2021.

[6] 東恩納琢偉. GearsOS におけるモデル検査を実現する手法について. 琉球大学大学院理工学研究科情報工学専攻修士論文, pp.13, March2021.

[7] 東恩納琢偉. GearsOS におけるモデル検査を実現する手法について. 琉球大学大学院理工学研究科情報工学専攻修士論文, pp26-27, March 2021.

[8] 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.

[9] https://cmake.org/

[10] 伊波 立樹. Gears OS の並列処理. 琉球大学大学院理工学研究科情報工学専攻修士論文, March 2018.

[11] 清水隆博. GearsOS のメタ計算. 琉球大学大学院理工学研究科情報工学専攻修士論文, March 2021.

[12] 清水隆博. GearsOS のメタ計算. 琉球大学大学院理工学研究科情報工学専攻修士論文, pp.77, March 2021.

付録

7.2　Mindmap

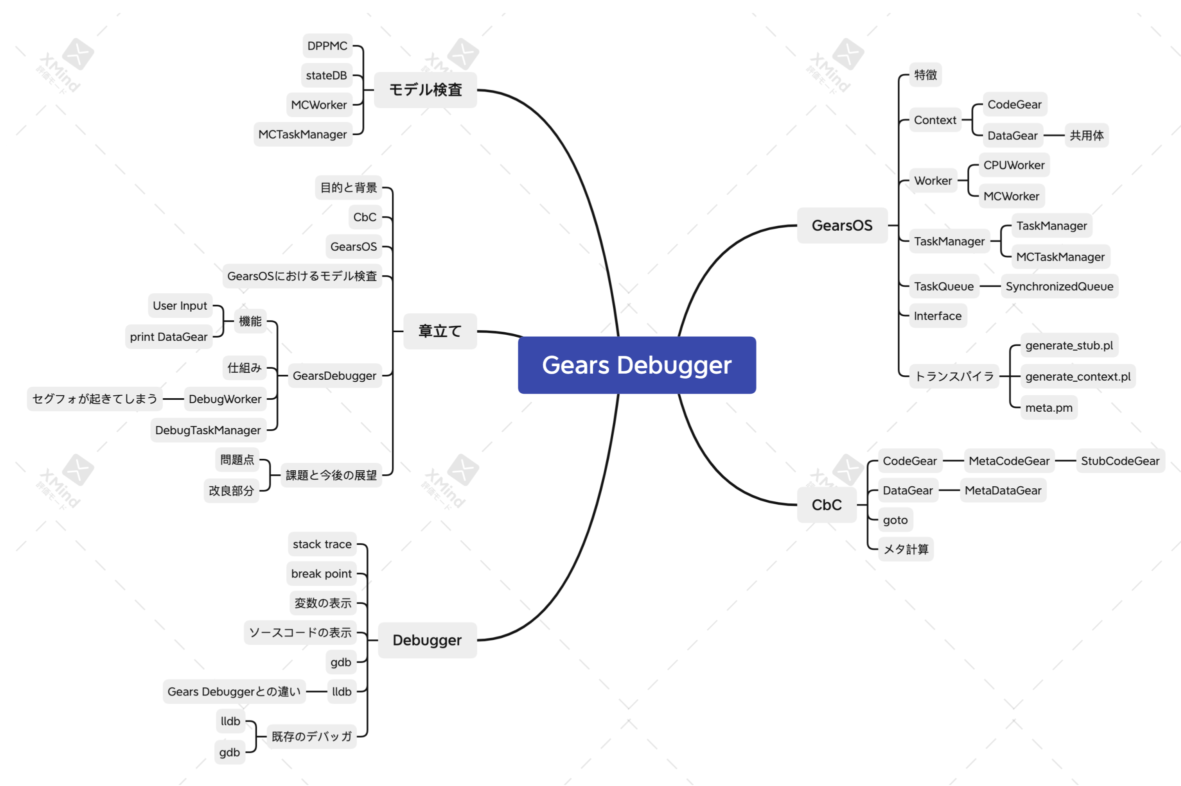

図 7.1　研究遂行および論文執筆時に作成した Mindmap