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

論文題目: GearsOS のメタ計算

氏　　名: 清水 隆博

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

アプリケーションの信頼性を保証するには、土台となる OS の信頼性は高く保証されていなければならない。信頼性を保証する方法としてテストコードを使う手法が広く使われている。OS のソースコードは巨大であり、並列処理など実際に動かさないと発見できないバグが存在する。OS の機能をテストですべて検証するのは不可能である。

テストに頼らず定理証明やモデル検査などの形式手法を使用して、OS の信頼性を保証したい。証明を利用して信頼性を保証する定理証明は、 Agda や Coq などの定理証明支援系を利用することになる。支援系を利用する場合、各支援系で OS を実装しなければならない。証明そのものは可能であるが、 支援系で証明されたソースコードがそのまま OS として動作する訳ではない。このためには定理証明されたコードを等価な C 言語などに変換する処理系が必要となる。

信頼性を保証するほかの方法として、プログラムの可能な実行をすべて数え上げて仕様を満たしているかを確認するモデル検査がある。モデル検査は実際に動作しているプログラムに対して実行することが可能である。すでに実装したプログラムのコードに変化を加えずモデル検査を行いたい。

プログラムは本来やりたい計算であるノーマルレベルの計算と、その計算をするのに必要なメタレベルの計算に別けられる。メタレベルの計算では資源管理などを行うが、 モデル検査などの証明をメタレベルの計算で行いたい。

この実現にはノーマルレベル、メタレベルの計算の処理の切り分けと、メタレベルの計算をより柔軟に扱う OS、言語処理系が必要となる。両レベルを記述できる言語に Continuation Based (CbC) がある。CbC はスタック、あるいは環境を持たず継続によって次の処理を行う特徴がある。CbC を用いて、拡張性と信頼性を両立する OS である GearsOS を開発している。

GearsOS の開発ではノーマルレベルのコードとメタレベルのコードの両方が必要であり、メタレベルの計算の数は多岐にわたる。GearsOS の開発を進めていくには、メタレベルの計算を柔軟に扱う API や、 自動でメタレベルの計算を作製する GearsOS のビルドシステムが必須となる。本研究では GearsOS の信頼性と拡張性の保証につながる、メタ計算に関する API について考察し、言語機能などの拡張を行った。また、メタ計算を自動生成しているトランスパイラを改良し、従来の GearsOS のシステムよりさらに柔軟性が高いものを考案した。

Abstract

In order to guarantee the reliability of an application, the reliability of the underlying OS must be highly guaranteed. The source code of an OS is huge, and there are bugs such as parallel processing that can only be discovered by actually running the OS. It is impossible to verify all the functions of an OS by testing.

Instead of relying on tests, we want to use formal methods such as theorem proving and model checking to guarantee the reliability of the OS. For theorem proving to guarantee reliability using proofs, we can use theorem proving support systems such as Agda and Coq. Another method of guaranteeing reliability is model checking, in which all possible executions of a program are counted to verify that it meets the specifications.

A program can be divided into normal-level computation, which is the computation we want to do, and meta-level computation, which is the computation necessary to do the computation. In meta-level computation, we want to perform resource management, etc., but we also want to perform proofs such as model checking in meta-level computation.

In order to achieve this, it is necessary to separate the processing of normal-level and meta-level computation, and to have an OS and language processing system that can handle meta-level computation more flexibly. A language that can describe both levels is Continuation Based C (CbC). CbC is characterized by the fact that it does not have a stack or an environment, and the next process is performed by continuation. Using CbC, we are developing GearsOS, an OS that is both scalable and reliable.

The development of GearsOS requires both normal-level code and meta-level code, and the number of meta-level computations varies widely. In order to proceed with the development of GearsOS, an API that can flexibly handle meta-level computations and a build system for GearsOS that can automatically create meta-level computations are essential. In this study, we discussed the API for meta-calculus, which will guarantee the reliability and scalability of GearsOS, and extended the language functions. We also improved the trans-compiler that automatically generates meta-calculus, and devised a system that is more flexible than the conventional GearsOS system.

研究関連業績

- CbC を用いた Perl6 処理系 清水 隆博, 河野真治 第 60 回プログラミング・シンポジウム, Jan, 2019
- How to build traditional Perl interpreters. Takahiro SHIMIZU PerlCon2019 , Aug, 2019
- Perl6 Rakudo の内部構造について 清水 隆博 オープンソースカンファレンス 2019 Okinawa, Apr, 2020
- 継続を基本とした OS Gears OS 清水 隆博, 河野真治 第 61 回プログラミング・シンポジウム, Jan, 2020
- Perl6 のサーバを使った実行 福田 光希, 清水 隆博, 河野真治 第 61 回プログラミング・シンポジウム, Jan, 2020
- xv6 の構成要素の継続の分析 清水 隆博, 河野 真治 (琉球大学), 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, 2020

目 次

図 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]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[4]やCoq[5]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。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。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、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。モデル検査はプログラムの可能な実行をすべて数え上げて要求している使用を満たしているかどうかを調べる手法である。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。また、モデル検査ができるモデル検査器は特定のプログラム形式でないと動かないものがある。例えば Spin は Promela 形式でないとモデル検査ができない。モデル検査ができる場合も、 モデル検査したコードと実際に動くコードを同一にしたい。また、モデル検査をする場合としない場合の切り替えを、より手軽に行いたい。

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。[6] 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。この場合は検証を実行するメタ計算と、 検証をしないメタ計算を手軽に切り替える必要がある。さらに検証用とそうでない用で、動作させたいアルゴリズムの実装そのもののコードを変更したくない。これも検証をメタレベルで行い、実装をノーマルレベルで行い、各レベルを切り離すことで実現可能である。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。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`[7][8] あるいは `LLVM`[9][10] 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 [11] 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。

CbC を用いてノーマルレベルとメタレベルの分離を行い、信頼性と拡張性を両立させることを目的として GearsOS を開発している。GeasOS では、 CbC の実行単位である CodeGear とデータの単位である DataGear を基本単位としている。GearsOS のメタ計算には MetaCodeGear と MetaDataGear を用いる。信頼性の保証は MetaCodeGear で行いたい。その為には GearsOS が柔軟にメタ計算を切り替えることが必要となる。また、GearsOS で実行されるメタ計算の数は膨大である。すべてをプログラミングするのではなく、いくつかのメタ計算は自動で生成されてほしい。GearsOS では拡張性の保証も重要な課題である。拡張性を保証するにはすべて純粋な CbC で実装すると、実装がきわめて煩雑である。その為には CbC とセマンティックが等しいより簡潔な GearsOS 独自のシンタックスなどが必要である。独自のシンタックスは Perl スクリプトによって等価な CbC のソースコードに変換していた。

従来の Perl スクリプトによるソースコードの変換では、 CodeGear が出力を DataGear に書き出す際に、 手でメタ計算を書かなければならない問題があった。また、 GearsOS のモジュール化の仕組みである Interface の実装である Implement の型定義ファイルが存在していなかった。GearsOS ではノーマルレベルで宣言した DataGear は、構造体の形で表現される。従来のシステムではこの構造体も手で実装しなければならず、メタレベルの計算のうち大半を手で実装する必要があった。これらのメタレベルの計算はコンパイル時に決定するために、自動化を行いたい。

本研究では GearsOS の信頼性と拡張性の保証につながる、メタ計算に関する API について考察する。GearsOS がメタ計算を自動生成している Perl トランスパイラで従来の GearsOS のシステムよりさらに拡張性の充実と、信頼性の保証を図る。

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、GCC10[12] 及び LLVM10/clang 上 [13] 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 に遷移すると元の処理に戻ってこれない。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 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。環境付き goto を使うと、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

DataGear は CbC でのデータの単位である。CbC 上では構造体の形で表現される。各 CodeGear の入力として受ける DataGear を InputDataGear と呼ぶ。逆に次の継続に渡す DataGear を OutputDataGear と呼ぶ。

メタレベルでは DataGear はポインタを扱っているが、 ノーマルレベルの DataGear はポインタを扱っていないと仮定している。例えばリストの DataGear を考えると、 C の実装の場合はポインタを使った単方向リストが考えられる。リストのそれぞれの要素には、メタレベルでは次の要素を指し示すポインタが含まれている。ノーマルレベルの DataGear として見る場合は、 リストそのものや、 リストの中の値そのものとして判断するために、より抽象化された単位として見える。これは関数型プログラミングにおける末尾再起呼び出し時の値のやりとりに似た概念である。

2.3 CbC を使った例題

ソースコード 2.1 に CbC を使った例題を、 ソースコード 2.2 に通常の C で実装した例題を示す。この例では構造体 `struct test` を codegear1 に渡し、その次に codegear2 に継続している。codegear2 からは codegear3 に goto し、 最後に exit する。

ソースコード 2.1: CbC の例題

```
1  extern int printf(const char*,...);
2
3  typedef struct test {
4      int number;
5      char* string;
6  } TEST;
7
8
9  __code codegear1(TEST);
10 __code codegear2(TEST);
11 __code codegear3(TEST);
12
13 __code codegear1(TEST testin){
14     TEST testout;
15     testout.number = testin.number + 1;
16     testout.string = testin.string;
17     goto codegear2(testout);
18 }
19
20 __code codegear2(TEST testin){
21     TEST testout;
22     testout.number = testin.number;
23     testout.string = "Hello";
24     goto codegear3(testout);
25 }
```

```
26
27 __code codegear3(TEST testin){
28     printf("number = %d\t string= %s\n",testin.number,testin.string);
29     goto exit(0);
30 }
31
32 int main(){
33     TEST test = {0,0};
34     goto codegear1(test);
35 }
```

ソースコード 2.2: ソースコード 2.1 の C での実装

```
1  extern int printf(const char*,...);
2
3  typedef struct test {
4      int number;
5      char* string;
6  } TEST;
7
8
9  void codegear1(TEST);
10 void codegear2(TEST);
11 void codegear3(TEST);
12
13 void codegear1(TEST testin){
14     TEST testout;
15     testout.number = testin.number + 1;
16     testout.string = testin.string;
17     codegear2(testout);
18 }
19
20 void codegear2(TEST testin){
21     TEST testout;
22     testout.number = testin.number;
23     testout.string = "Hello";
24     codegear3(testout);
25 }
26
27 void codegear3(TEST testin){
28     printf("number = %d\t string= %s\n",testin.number,testin.string);
29     exit(0);
30 }
31
32 int main(){
33     TEST test = {0,0};
34     codegear1(test);
35 }
```

CbC の場合は継続で進んでいくが、 C 言語での実装は void 型の返り値を持つ関数呼び出しで表現される。codegear3 に遷移したタイミングで、 CbC は main 関数のスタックしか持たないが、 C 言語では codegear1、codegear2 のスタックをそれぞれ持つ違いがある。

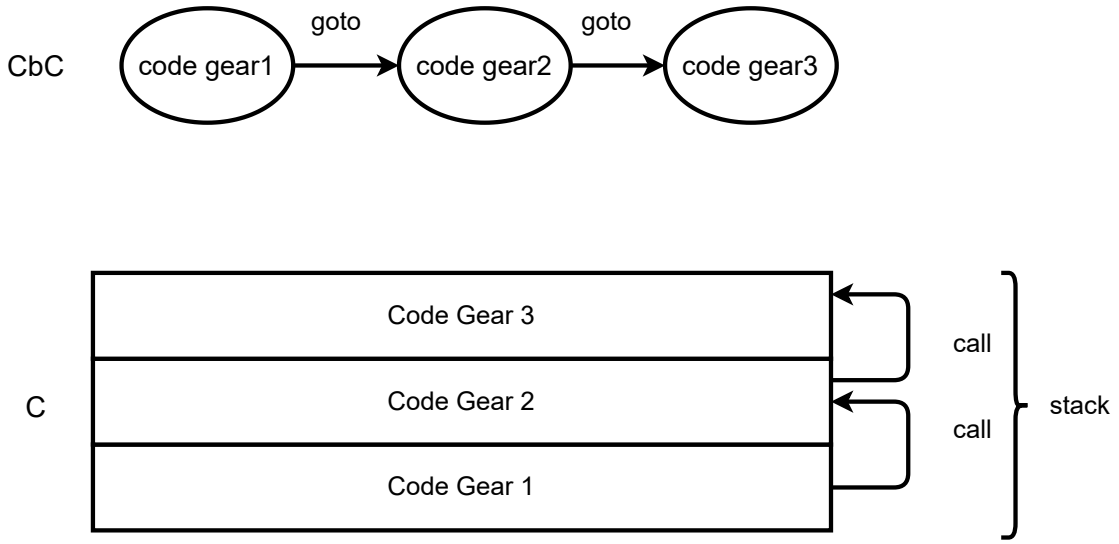

図 2.1: CbC と C の処理の差

(図 2.1)

実際に軽量継続になっているかを、この例題をアセンブラに変換した結果を見比べて確認する。

ソースコード 2.3: CbC の例題をコンパイルしたアセンブラの一部

```
codegear1:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, %eax
    movq    %rsi, %rcx
    movq    %rcx, %rdx
    movq    %rax, -32(%rbp)
    movq    %rdx, -24(%rbp)
    movl    -32(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -16(%rbp)
    movq    -24(%rbp), %rax
    movq    %rax, -8(%rbp)
    movl    -16(%rbp), %edx
    movq    -8(%rbp), %rax
    movl    %edx, %edi
    movq    %rax, %rsi
    popq    %rbp
    .cfi_def_cfa 7, 8
```

```
25      jmp codegear2
26      .cfi_endproc
27  .LFE0:
28      .size   codegear1, .-codegear1
29      .section    .rodata
30  .LC0:
31      .string "Hello"
32      .text
33      .globl  codegear2
34      .type   codegear2, @function
```

ソースコード 2.4: C 言語の例題をコンパイルしたアセンブラの一部

```
1       pushq   %rbp
2       .cfi_def_cfa_offset 16
3       .cfi_offset 6, -16
4       movq    %rsp, %rbp
5       .cfi_def_cfa_register 6
6       subq    $32, %rsp
7       movl    %edi, %eax
8       movq    %rsi, %rcx
9       movq    %rcx, %rdx
10      movq    %rax, -32(%rbp)
11      movq    %rdx, -24(%rbp)
12      movl    -32(%rbp), %eax
13      addl    $1, %eax
14      movl    %eax, -16(%rbp)
15      movq    -24(%rbp), %rax
16      movq    %rax, -8(%rbp)
17      movl    -16(%rbp), %edx
18      movq    -8(%rbp), %rax
19      movl    %edx, %edi
20      movq    %rax, %rsi
21      call    codegear2
22      nop
23      leave
24      .cfi_def_cfa 7, 8
25      ret
26      .cfi_endproc
27  .LFE0:
28      .size   codegear1, .-codegear1
29      .section    .rodata
30  .LC0:
31      .string "Hello"
32      .text
33      .globl  codegear2
34      .type   codegear2, @function
```

codegear1 から codegear2 への移動の際に、CbC と C で発行されるアセンブラの命令を比較する。CbC の例題の場合のアセンブラのソースコード 2.3 は codegear2 へ 25 行目で jmp 命令を使って遷移している。対して C 言語での実装の場合 (ソースコード 2.4) は 21 行目で callq を使っている。jmp 命令はプログラムカウンタを切り替えるのみの命令であり、

call は関数呼び出しの命令であるためにスタックの操作が行われる。CbC での goto 文はすべてこの jmp 命令に変換されるため、 関数呼び出しより軽量に実行することが可能である。

2.4 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[14] 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 2.5 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 2.5: 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     }
```

軽量継続を持つ CbC を利用して、 証明可能な OS を実装したい。その為には証明に使用される定理証明支援系や、 モデル検査機での表現に適した状態遷移単位での記述が求められる。CbC で使用する CodeGear は、 状態遷移モデルにおける状態そのものとして捉えることが可能である。CodeGear を元にプログラミングをするにつれて、 CodeGear の入出力の Data も重要であることが解ってきた。

2.5 メタ計算

メタ計算のメタとは、高次元などの意味を持つ言葉であり、特定の物の上位に位置するものである。メタ計算の場合は計算に必要な計算や、 計算を行うのに必要な計算を指す。GearsOS でのメタ計算は、 通常の計算を管理している OS レベルの計算などを指す。OS から見たメタ計算は、自分自身を検証する計算などになる。

ノーマルレベルの計算からすると、メタ計算は通常隠蔽される。これは UNIX のプログラムを実行する際に、 OS のスケジューラーのことを意識せずに実行可能であることな

どから分かる。新しいプロセスを作製する場合は `fork` システムコールを実行する必要がある。システムコールの先は OS が処理を行う。fork システムコールの処理を OS が計算するが、 この計算はユーザープログラムから見るとメタ計算である。システムコールの中で何が起こっているかはユーザーレベルでは判断できず、 返り値などの API を経由して情報を取得する。現状の UNIX ではメタ計算はこの様なシステムコールの形としても表現される。

メタデータなどはデータのデータであり、 データを扱う上で必要なデータ情報を意味する。プログラムの中でプログラムを生成するものをメタプログラミングなどと呼ぶ。メタ計算やメタプログラムは、プログラム自身の検証などにとって重要な機能である。しかしメタレベルの計算をノーマルレベルで自在に記述してしまうと、 ノーマルレベルでの信頼性に問題が発生する。メタレベルではポインタ操作や資源管理を行うため、メモリオーバーフローなどの問題を簡単に引き起こしてしまう。メタレベルの計算とノーマルレベルの計算を適切に分離しつつ、 ノーマルレベルから安全にメタレベルの計算を呼び出す手法が必要となる。

プログラミング言語からメタ計算を取り扱う場合、 言語の特性に応じて様々な手法が使われてきた。関数型プログラミングの見方では、 メタ計算はモナドの形で表現されていた。[15] OS の研究ではメタ計算の記述に型付きアセンブラを用いることもある。[16]

通常ユーザーがメタレベルのコードを扱う場合は特定の API を経由することになる。プログラム実行中のスタックの中には、 プログラムが現在実行している関数までのフレームや、各関数でアロケートされた変数などの情報が入る。これらを環境と呼ぶ。現状のプログラミング言語や OS では、 この環境を常に持ち歩かなければならない。ノーマルレベルとメタレベルを分離しようとすると、 環境の保存について考慮しなければならず、 結果的にシステムコールなどの API を使わなければならない。システムコールを利用しても、 保存されている環境が常に存在する。また関数単位での分離を行っても、 呼び出す関数の数が細かくなってしまい、スタックの容量を容易に消費してしまう。既存言語ではメタ計算の分離が困難である。

CbC では goto 文による軽量継続によって、 スタックを goto の度に捨てていく。そもそもスタックが存在しないため、 暗黙の環境も存在せずに自由にプログラミングが可能となる。また CodeGear をどれだけ呼び出しても、関数呼び出し時に伴うスタックの消費も存在しない。メタ計算の単位で細かく CodeGear を切り分けても、実行の問題が生じない。その為従来のプログラミング言語ではできなかった、ノーマルレベルとメタレベルのコードの分離が容易に行える。

CbC でのメタ計算は CodeGear、 DataGear の単位がそのまま使用できる。メタ計算を行う CodeGear や、 メタな情報を持つ DataGear が存在する。これらの単位はそれぞれ、MetaCodeGear、 MetaDataGear と呼ばれる。

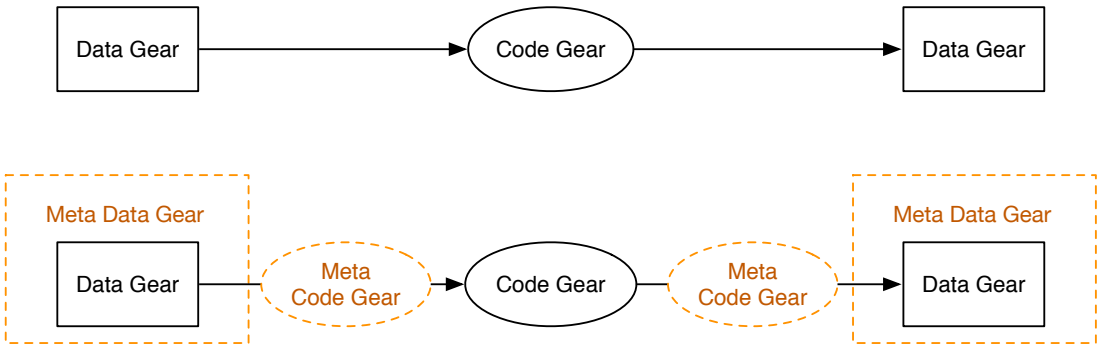

図 2.2: CodeGear と MetaCodeGear

2.6 MetaCodeGear

遷移する各 CodeGear の実行に必要なデータの整合性の確認などはメタ計算である。この計算は MetaCodeGear と呼ばれる各 CodeGear ごと実装されたメタな CodeGear で計算を行う。

特に対象の CodeGear の直前で実行される MetaCodeGear を StubCodeGear と呼ぶ。ユーザーからするとノーマルレベルの CodeGear 間の移動に見えるが、実際には StubCodeGear が挿入される。MetaCodeGear や MetaDataGear は、プログラマが直接実装せず、 Perl スクリプトによって GearsOS のビルド時に生成される。ただし Perl スクリプトはすでに書かれていた StubCodeGear は上書きしない。スクリプトに問題がある場合や、 細かな調整をしたい場合はプログラマが直接実装可能である。CodeGear から別の CodeGear に遷移する際の DataGear などの関係性を、図 2.2 に示す。

通常のコード中では入力の DataGear を受け取り CodeGear を実行、 結果を DataGear に書き込んだ上で別の CodeGear に継続する様に見える。この流れを図 2.2 の上段に示す。しかし実際は CodeGear の実行の前後に実行される MetaCodeGear や入出力の DataGear を MetaDataGear から取り出すなどのメタ計算が加わる。これは図 2.2 の下段に対応する。

2.7 MetaDataGear

基本は C 言語の構造体そのものであるが、 DataGear の場合はデータに付随するメタ情報も取り扱う。これはデータ自身がどういう型を持っているかなどの情報である。ほかに計算を実行する CPU、 GPU の情報や、 計算に必要なすべての DataGear の管理などの実行環境のメタデータも DataGear の形で表現される。このメタデータを扱う DataGear を MetaDataGear と呼ぶ。

また CbC はスタックを持たないため、 データを保存したい場合はスタック以外の場所に値を書き込む必要がある。このスタック以外の場所は DataGear であり、 メタなデータを扱っているために MetaDataGear と言える。具体的に MetaDataGear がどのように構成されているかは、CbC を扱うプロジェクトによって異なる。

第3章　GearsOS

GearsOS とは Continuation Based C を用いて信頼性と拡張性の両立を目指して実装している OS プロジェクトである。[17] CodeGear と DataGear を基本単位として実行する。CodeGear を基本単位としているため、 各 CodeGear は割り込みされず実行される必要がある。割り込みを完全に制御することは一般的には不可能であるが、 GearsOS のメタ計算部分でこれを保証したい。DataGear も基本単位であるため、各 CodeGear が DataGear をどのように扱っているか、書き込みをしたかは GearsOS 側で保証するとしている。

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。GearsOS はノーマルレベルとメタレベルの分離を目指して構築している OS である。すべてをプログラマが純粋な CbC で記述してしまうと、 メタレベルの情報を実装しなければならず、ノーマルレベルとメタレベルの分離をした意味がなくなってしまう。GesrsOS ではユーザーが書いたノーマルレベルのコードの特定の記述や、シンタックスをもとに、メタレベルの情報を含む等価な CbC へとコンパイル時にコードを変換する。コード変換は Perl スクリプトで行われている。

現在の GearsOS は Unix システム上のアプリケーションとして実装されているものと、xv6 の置き換えとして実装されているもの [18] がある。

3.1　GearsOS の構成

GearsOS は様々な役割を持つ CodeGear と DataGear で構成されている。また CodeGear と DataGear のモジュール化の仕組みとして Interface が導入されている。GearsOS の構成図を図 3.1 に示す。中心となる MetaDataGear は以下の要素である。

- Context
- TaskManager
- TaskQueue
- Worker

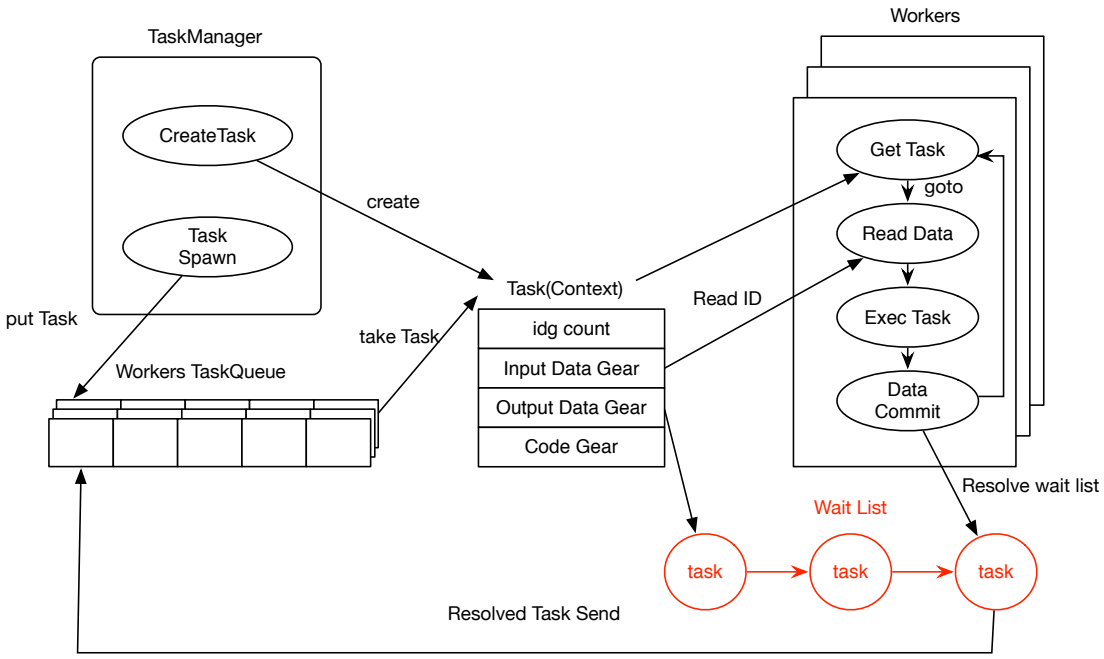

図 3.1: GearsOS の構成

3.2 Context

Context とは従来の OS のプロセスに相当する概念である。GearsOS でのデータの単位から見ると、 MetaDataGear に相当する。Context の概要図を図 3.2 に、実際の CbC 上での定義をソースコード 3.1 に示す。

Context は MetaDataGear である為に、 ノーマルレベルの CodeGear からは context は直接参照しない。context の操作をしてしまうと、メタレベルとノーマルレベルの分離をした意味がなくなってしまう為である。

Context はプロセスに相当するので、ユーザープログラムごとに Cotnext が存在する。この Context を User Context と呼ぶ。さらに実行している GPU や CPU ごとに Context が必要となる。これらは CPU Context と呼ばれる。GearsOS は OS であるため、全体を管理する Kernel の Context も必要となる。これは KernelContext や KContext と呼ばれる。KContext はすべての Context を参照する必要がある。OS が持たなければならない割り込みのフラグなどは KContext に置かれている。GearsOS のメタレベルのプログラミングでは、 今処理をしている Context が誰の Context であるかを強く意識する必要がある。

ソースコード 3.1: context の定義

```
1 struct Context {
2     enum Code next;
```

```
3     struct Worker* worker;
4     struct TaskManager* taskManager;
5     int codeNum;
6     __code (**code) (struct Context*);
7     union Data **data;
8     struct Meta **metaDataStart;
9     struct Meta **metaData;
10    void* heapStart;
11    void* heap;
12    long heapLimit;
13    int dataNum;
14
15    // task parameter
16    int idgCount; //number of waiting dataGear
17    int idg;
18    int maxIdg;
19    int odg;
20    int maxOdg;
21    int gpu; // GPU task
22    struct Context* task;
23    struct Element* taskList;
24 #ifdef USE_CUDAWorker
25    int num_exec;
26    CUmodule module;
27    CUfunction function;
28 #endif
29    /* multi dimension parameter */
30    int iterate;
31    struct Iterator* iterator;
32 };
```

Context は GearsOS の計算で使用されるすべての DataGear と CodeGear を持つ。つまり GearsOS で使われる CodeGear と DataGear は、誰かの Context に必ず書き込まれている。各 CodeGear、DataGear は Context はそれぞれ配列形式で Context にデータを格納する場所が用意されている。CodeGear が保存されている配列はソースコード 3.1 の 6 行目で定義している `code` である。StubCodeGear は Context のみを引数で持つため、 `__code stub(struct Context*)` の様な CodeGear の関数ポインタのポインタ、つまり CodeGear の配列としての定義されている。これは前述した StubCodeGear の関数ポインタが格納されており、 `__code meta` でのディスパッチに利用される。

DataGear が保存されている配列は 7 行目で定義している `data` である 。すべての DataGear は GearsOS 上では `union Data` 型として取り扱えるので、 `union Data` のポインタの配列として宣言されている。ただし GearsOS で使うすべての DataGear がこの Context に保存されている訳ではない。Interface を利用した goto 時の値の保存場所として、この配列に DataGear ごと割り振られた場所に DtaGear を保存する用途で利用している。CodeGear で利用している配列と同様に、 この配列の添え字も DataGear の番号に対応している。

DataGear は配列形式のデータ格納場所のほかに、Context が持つヒープに保存することも可能である。計算で必要な DataGear は、 CbC の中でアロケーションした場合は

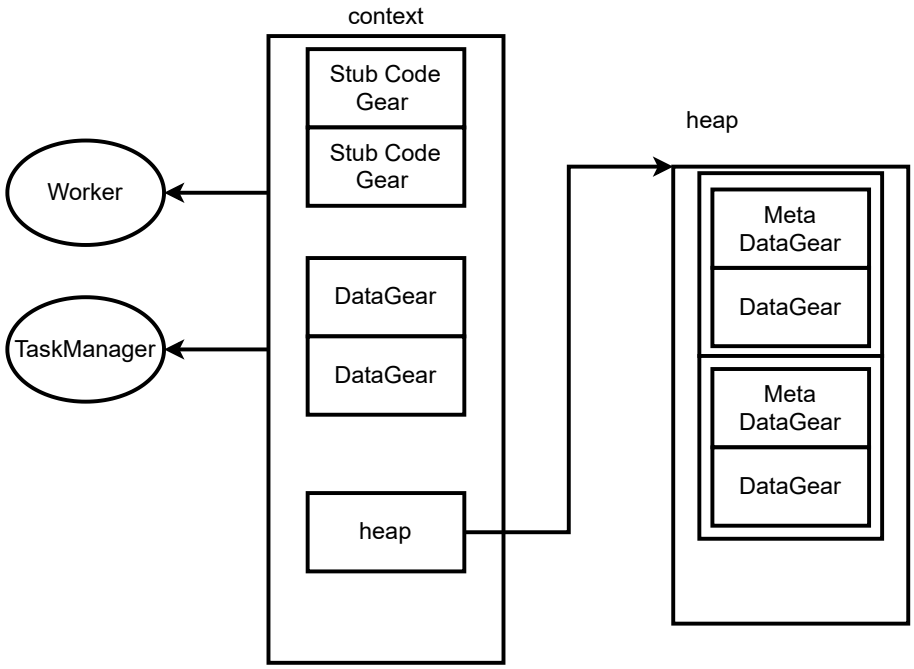

図 3.2: Context の概要図

Context にヒープに書き込まれる。ヒープには DataGear と、書き込んだ DataGear のメタ情報が記載されている MetaDataGear で構成されている。

3.3　Stub Code Gear

次の CodeGear に継続する際、ノーマルレベルから見ると次の CodeGear を直接指定しているように見える。さらに次の CodeGear に引数などを直接渡しているようにも見える。しかしノーマルレベルから次の CodeGear に継続する場合は関数ポインタなどが必要になるが、 これらはメタ計算に含まれる。その為純粋にノーマルレベルから CodeGear 間を自由に継続させてしまうと、 ノーマルレベルとメタレベルの分離ができなくなってしまう。ノーマルレベルとメタレベルの分離の為に、 次の CodeGear には直接継続させず、間に MetaCodeGear をはさむようにする必要がある。またポインタをノーマルレベルには持たせず、 継続先の CodeGear は番号を使って指定する。CodeGear 間の継続は GearsOS のビルド時に Perl スクリプトによって書き換えが行われ、MetaCodeGear を経由するように変更される。

GearsOS では DataGear はすべて Context を経由してやり取りをする。次の継続に DataGear を渡す場合、 継続する前に一度 Context に DataGear を書き込み、 継続先で Context

から DataGear を取り出す。Context は MetaDataGear であるために、 ノーマルレベルの CodeGear ではなく MetaCodeGear で扱う必要がある。各 CodeGear の計算で必要な DataGear を Context から取り出す MetaCodeGear は、 実行したい CodeGear の直前で実行される必要がある。この CodeGear を特に StubCodeGear と呼ぶ。StubCodeGear はすべての CodeGear に対して実装しなければならず、 手で実装するのは煩雑である。StubCodeGear も GearsOS のビルド時に Perl スクリプトによって自動生成される。

ソースコード 3.4 に示すノーマルレベルで記述した CodeGear を、Perl スクリプトによって変換した結果をソースコード 3.5 に示す。常に自分自身の Context を CodeGear は入力の形で受け取る為、変換後の `pushSingleLinkedStack` は、 第 1 引数に Context が加わっている。pushSingleLinkedStack は引数は 3 つ要求していた。これらの引数は生成された `pushSingleLinkedStack_stub` が Context の特定の場所から取り出す。この CodeGear は GearsOS の Interface を利用しており、Stack Interace の実装となっている。マクロ `Gearef` は、 context の Interface 用の DataGear の置き場所にアクセスするマクロであり、 Stack Interface の置き場所から、引数情報を取得している。マクロ `Gearef` の定義をソースコード 3.2 に示す。マクロ `Gearef` では引数で与えられた DataGear の名前を、enum を利用した番号に変換し、context から値を取り出している。DataGear は `enum Data` 型で各 DataGear の型ごとに番号が割り振られている。(ソースコード 3.3)

ソースコード 3.2: Gearef マクロ

```
1 #define Gearef(context, t) (&(context)->data[D_##t]->t)
```

ソースコード 3.3: enumData の定義

```
1 enum DataType {
2     D_Code,
3     D_Atomic,
4     D_AtomicReference,
5     D_CPUWorker,
6     D_Context,
7     D_Element,
8     ...
9 };
```

すべての引数を取得したのちに、`goto pushSingleLinkedStack` で、CodeGear に継続する。

ソースコード 3.4: Stack に Push する CodeGear

```
1 __code pushSingleLinkedStack(struct SingleLinkedStack* stack, union Data*
      data, __code next(...)) {
2     Element* element = new Element();
3     element->next = stack->top;
4     element->data = data;
5     stack->top = element;
6     goto next(...);
```

```
7 }
```

ソースコード 3.5: 3.4 の StubCodeGear

```
1  __code pushSingleLinkedStack(struct Context *context,struct
       SingleLinkedStack* stack, union Data* data, enum Code next) {
2      Element* element = &ALLOCATE(context, Element)->Element;
3      element->next = stack->top;
4      element->data = data;
5      stack->top = element;
6      goto meta(context, next);
7  }
8
9  __code pushSingleLinkedStack_stub(struct Context* context) {
10     SingleLinkedStack* stack = (SingleLinkedStack*)GearImpl(context,
       Stack, stack);
11     Data* data = Gearef(context, Stack)->data;
12     enum Code next = Gearef(context, Stack)->next;
13     goto pushSingleLinkedStack(context, stack, data, next);
14 }
```

Context と継続の関係性を図 3.3 に示す。StubCodeGear は GearsOS で定義されているノーマルレベルの CodeGear のすべてに生成される。

__code meta の定義をソースコード 3.6 に示す。

ソースコード 3.6: __code meta

```
1  __code meta(struct Context* context, enum Code next) {
2      goto (context->code[next])(context);
3  }
```

__code meta は Context に格納されている CodeGear の配列から CodeGear のアドレスを取得し継続する。この際に配列の要素を特定する際に使われる添え字は、各 CodeGear に割り振られた番号を利用している。この番号は C 言語の列挙体を使用した enum Code 型で定義されている。enum Code 型の定義をソースコード 3.7 に示す。命名規則は C_CodeGearName となっている。

ソースコード 3.7: CodeGear の番号である enumCode の定義

```
1  enum Code {
2      C_checkAndSetAtomicReference,
3      C_clearSingleLinkedStack,
4      C_clearSynchronizedQueue,
5      C_createTask,
6      C_decrementTaskCountTaskManagerImpl,
7      C_exit_code,
8      C_get2SingleLinkedStack,
9      ...
10 };
```

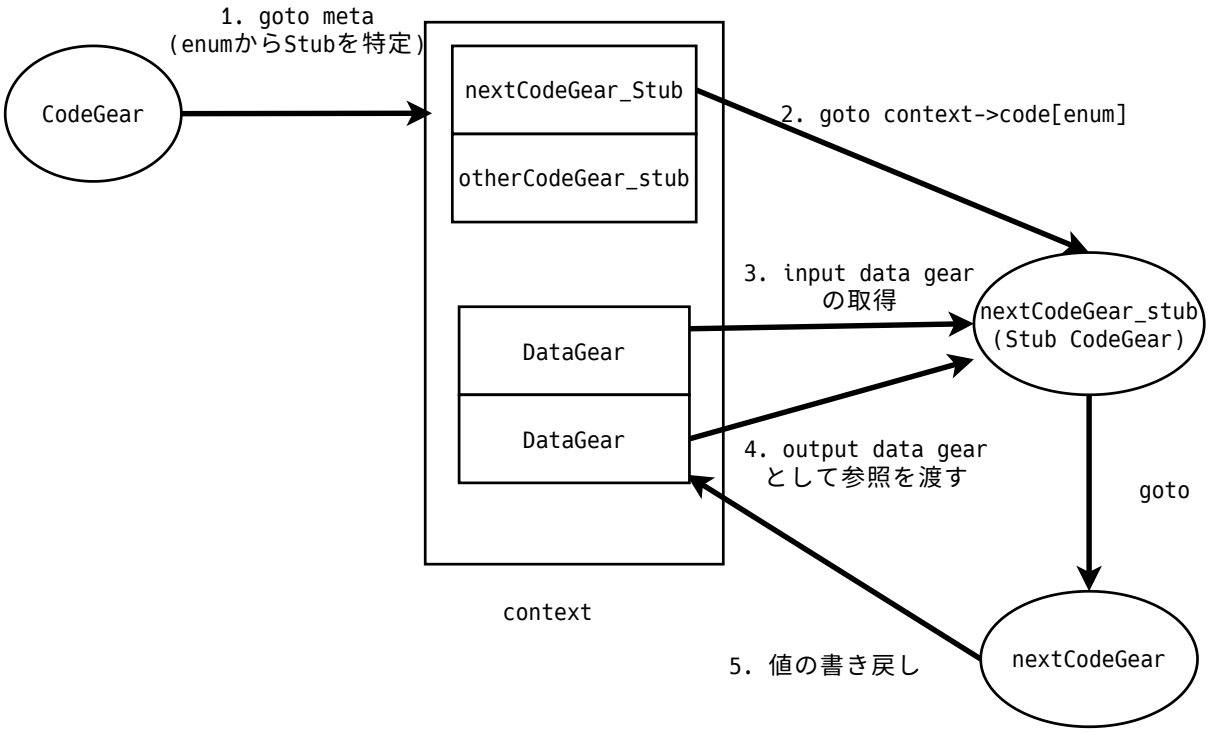

図 3.3: Context を参照した CodeGear のデータアクセス

`enum Code` 型は GearsOS のコンパイル時に利用されている CodeGear を数え上げて生成される。Context の code 配列には、各 CodeGear の StubCodeGear の関数ポインタが配置されている。よって`__code meta`から継続する先の CodeGear は、呼び出し先の CodeGear の直前に実行される StubCodeGear になる。

CodeGear から CodeGear への継続は、関数型プログラミングの継続先に渡す Data と Code の組の Closure となっている。シンタックスでは継続の際に引数 (...) を渡す。これは処理系では特に使用していないキーワードであるが、 この Closure を持ち歩いていることを意識するために導入されている。

3.4　TaskManager

TaskManager は GearsOS 上で実行される Task の管理を行う。GearsOS 上の Task とは Context のことであり、各 Context には自分の計算に必要な DataGear のカウンタなどが含まれている。TaskManager は、CodeGear の計算に必要な入力の DataGear(InputDataGear) が揃っているかの確認、揃っていなかったら待ち合わせを行う処理がある。すべての

DataGear が揃った場合、 Task を Worker の Queue に通知し Task を実行させる。この処理は GearsOS を並列実行させる場合に必要な機能となっている。

TaskManager は性質上シングルトンである。その為、複数 Worker を走らせた場合でも 1 全体で 1 つのみの値を持っていたいものは TaskManager が握っている必要がある。例えばモデル検査用の状態保存用のデータベース情報は、 TaskManager が所有している。

3.5　TaskQueue

GearsOS の TaskQueue は SynchronizedQueue で表現されている。TaskQueue は Worker が利用する Queue となっている。

Worker の Queue は、TaskManager に接続して Task を送信するスレッドと、 Task を実行する Worker 自身のスレッドで扱われる。さらに Worker が複数走る可能性もある。その為 SynchronizedQueue は、マルチスレッドでもデータの一貫性を保証する必要がある。GearsOS では CAS(Check and Set, Compare and Swap) を利用して実装が行われている。

3.6　Worker

Worker は Worker の初期化にスレッドを作る。GearsOS ではスレッドごとにそれぞれ Context が生成される。Worker はスレッド作製後に Context の初期化 API を呼び出し、自分のフィールドに Context のアドレスを書き込む。

スレッド作製後は TaskManager から Task を取得する。Task は Context の形で表現されているために、Worker の Context を Task に切り替え、 Task の次の継続に実行する。OutputDataGear がある場合は、Task 実行後に DataGear の書き出しが行われる。

Worker は CodeGear の前後で確実に呼び出される。この性質を利用すると、CodeGear の実行の前後での状態を記録することが可能である。つまりモデル検査が可能である為、モデル検査用の Worker を定義して入れ替えるとコードに変更を与えずに実行できる。Worker 自体は Interface で表現されているために、入れ替えは容易となっている。GearsOS では通常の Worker として CPUWorker を、GPU に関連した処理をする CUDAWorker、合間にモデル検査関連のメタ計算をはさむ MCWorker が定義されている。

3.7　union Data型

CbC/GearsOS では DataGear は構造体の形で表現されていた。すべての DataGear を管理する、Context は計算で使うすべての DataGear の型定義を持っている。各 DataGear は当然ではあるが別の型である。例えば Stack DataGear と Queue DataGear は、それぞ

れ struct Stack と struct Queue で表現されるが、 C 言語のシステム上別の型とみなされる。メタレベルで見れば、 この型定義は `union Data` 型にすべて書かれている。しかし Context はこれらの型をすべて DataGear として等しく扱う必要がある。この為に C 言語の共用体を使用し、汎用的な DataGear の型である union Data 型を定義している。共用体とは、構成するメンバ変数で最大の型のメモリサイズと同じメモリサイズになる特徴があり、複数の異なる型をまとめて管理することができる。構造体と違い、1 度に一つの型しか使うことができない。

実際にどの型が書き込まれているかは、 Context のどこに書き込まれているかによって確認の方法が異なる。Interface の入出力で利用している data 配列の場合は、enum の番号と data 配列の添え字が対応している。このため enum で指定した場所に入っている union Data の具体的な型は、 enum と対応する DataGear になる。context のヒープにアロケートされた DataGear の場合は、 型情報を取得できる MetaDataGear にアクセスすると、なんの型であったかが分かる。

Context から取り出してきた union Data から DataGear の型への変換はメタ計算で行われる。GearsOS の場合は、計算したい CodeGear の直前で実行される StubCodeGear で値のキャストが行われる。

3.8 Interface

GearsOS のモジュール化の仕組みとして Interface がある。Interface は CodeGear と各 CodeGear で使う入出力の DataGear の集合である。Interface に定義されている CodeGear は、各 Interface が満たすこと期待する API である。

Inteface は仕様 (Interface) と、実装 (Implement, Impl) を別けて記述する。Interface を呼び出す場合は、Interface に定義された API に沿ってプログラミングをすることで、Impl の内容を隠蔽することができる。これによってメタ計算部分で実装を入れ替えつつ Interface を使用したり、 ふるまいを変更することなく具体的な処理の内容のみを変更することが容易にできる。これは Java の Interface、Haskell の型クラスに相当する機能である。

3.8.1 Interface の定義

GearsOS に実装されている Queue の Interface の定義をソースコード 3.8 に示す。

ソースコード 3.8: Queue の Interface

```
1 typedef struct Queue<Impl>{
2         union Data* queue;
3         union Data* data;
4         __code whenEmpty(...);
5         __code clear(Impl* queue, __code next(...));
```

```
6         __code put(Impl* queue, union Data* data, __code next(...));
7         __code take(Impl* queue, __code next(union Data*, ...));
8         __code isEmpty(Impl* queue, __code next(...), __code whenEmpty
      (...));
9         __code next(...);
10 } Queue;
```

Interface の定義は、前半に入出力で利用する DataGear を列挙する。ここでは queue と data を利用する。4 行目からは API の宣言である。各 API は CodeGear であるので__code で宣言する。各 CodeGear の第 1 引数は Impl*型の変数になっている。これは Interface と対応する Implement の DataGear のポインタである。Java などのオブジェクト指向言語では self や this のキーワードで参照できるものとほぼ等しい。Interface 宣言時には具体的にどの型が来るかは不定であるため、 キーワードを利用している。Impl は Interface の API 呼び出し時に、メタレベルの処理である StubCodeGear で自動で入力される。その為ユーザーは Interface の API を呼び出す際は、 この Impl に対応する引数は設定しない。すなわち実際にいれるべき引数は、 Impl を抜いたものになる。

第 1 引数に Impl が来ない CodeGear として whenEmpty と next が Queue の例で存在している。これらは API の呼び出し時に継続として渡される CodeGear であるため、Interface の定義時には不定である。その為... を用いて、不定な CodeGear と DataGear の Closure が来ると仮定している。8 行目で定義している whenEmpty は Queue の状態を確認し、空でなければ next、空であれば whenEmpty に継続する。これらは呼び出し時に CodeGear を入力して与えることになる。

3.8.2 Interface の呼び出し

Interface で定義した API は interface->method の記法で呼びだすことが可能である。ソースコード 3.9 では、 Queue Interface の take API を呼び出している。take は__code next を要求しているので、 CodeGear の名前を引数として渡している。これはノーマルレベルでは enum の番号として処理される。take は出力を 1 つ出す CodeGear である為、継続で渡された odgCommitCUDAWorker4 は Stub でこの出力を受け取る。

ソースコード 3.9: Interface の API の呼び出し

```
1 __code odgCommitCUDAWorker3(struct CUDAWorker* worker, struct Context*
      task) {
2     int i = worker->loopCounter;
3     struct Queue* queue = GET_WAIT_LIST(task->data[task->odg+i]);
4     goto queue->take(odgCommitCUDAWorker4);
5 }
```

また、Interface を利用する場合はソースコード中に#interface "interfaceName.h"と記述する必要がある。例えば Queue を利用する場合は#interface "Queue.h"と記述しな

ければならない。#interface 構文は、一見すると C 言語のマクロの様に見える。実際にはマクロではなく、 Perl スクリプトによってメタレベルの情報を含む CbC ファイルに変換する際に、 Perl スクリプトに使っている Interface を教えるアノテーションの様な役割である。Perl スクリプトによって変換時に、 #interface の宣言は削除される。

3.8.3 Interface のメタレベルの実装

Interface 自身も DataGear であり、実際の定義は context の union Data 型に記述されている。メタレベルでは Inteface の DataGear の GearsOS 上の実装である構造体自身にアクセス可能である。Queue Interface に対応する構造体の定義をソースコード 3.10 に示す。

ソースコード 3.10: Queue の Interface に対応する構造体

```
1  struct Queue {
2      union Data* queue;
3      union Data* data;
4      enum Code whenEmpty;
5      enum Code clear;
6      enum Code put;
7      enum Code take;
8      enum Code isEmpty;
9      enum Code next;
10 } Queue;
```

Interface の実装は、この構造体に代入されている値で表現される。Interface の定義 (ソースコード 3.8) と、 実際の構造体 (ソースコード 3.10) を見比べると、 CodeGear は enum Code として表現し直されている。enum Code は GearsOS で使うすべての CodeGear に割り振られた番号である。Interface は API に対応する enum Code に、Impl 側の enumCode を代入することで、実装を表現している。Interface の Impl 側の DataGear は、各 Interface に存在する、Interface 名の最初の一文字が小文字になった union Data 型のポインタ経由で取得可能である。

3.8.4 Interface の Impl の実装

実際に Interface の初期化をしている箇所を確認する。Queue Interface に対応する SingleLinkedQueue の実装を 3.11 に示す。

ソースコード 3.11: SingleLinkedQueue の実装

```
1  #include "../context.h"
2  #include <stdio.h>
3  #interface "Queue.h"
4
5  Queue* createSingleLinkedQueue(struct Context* context) {
```

```
6       struct Queue* queue = new Queue();
7       struct SingleLinkedQueue* singleLinkedQueue = new SingleLinkedQueue()
        ;
8       queue->queue = (union Data*)singleLinkedQueue;
9       singleLinkedQueue->top  = new Element();
10      singleLinkedQueue->last = singleLinkedQueue->top;
11      queue->take  = C_takeSingleLinkedQueue;
12      queue->put   = C_putSingleLinkedQueue;
13      queue->isEmpty = C_isEmptySingleLinkedQueue;
14      queue->clear = C_clearSingleLinkedQueue;
15      return queue;
16  }
17
18  __code clearSingleLinkedQueue(struct SingleLinkedQueue* queue, __code
        next(...)) {
19      queue->top = NULL;
20      goto next(...);
21  }
22
23  __code putSingleLinkedQueue(struct SingleLinkedQueue* queue, union Data*
        data, __code next(...)) {
24      Element* element = new Element();
25      element->data = data;
26      element->next = NULL;
27      queue->last->next  = element;
28      queue->last = element;
29      goto next(...);
30  }
```

Interface の実装の場合も、Interface 呼び出しの API である`#interface "Queue.h"`を記述する必要がある。`createSingleLinkedQueue` は SingleLinkedQueue で実装した Queue Interface のコンストラクタである。これは関数呼び出しで実装されており、 返り値は Interface のポインタである。コンストラクタ内では Queue および SingleLinkedQueue のアロケーションを行っている。`new` 演算子が使われているが、 これは GearsOS で拡張されたシンタックスの 1 つである。new は GearsOS のビルド時に Perl スクリプトによって、context が持つ DataGear のヒープ領域の操作のマクロに切り替わる。ノーマルレベルでは context にアクセスできないので、 Java の様なアロケーションのシンタックスを導入している。

3.8.5 goto 時の Context と Interface の関係

Interface はモジュール化の仕組みとしてでなく、メタレベルでは一時的な変数の置き場所としても利用している。ソースコード 3.9 で呼び出している take は、 OutputDataGear がある API である。この OutputDataGear は、 context 内の DataGear の置き場所である data 配列の、 Interface のデータ格納場所に書き込まれる。OutputDataGear を取得する場合は、 継続先でなく、API の Interface から取得しないといけない。

また、goto 文で別の CodeGear に遷移する際も、引数情報を継続先の context の data 配列の場所に書き込む必要がある。この処理はメタレベルの計算であるため、 GearsOS のビルド時に Perl で変換される。ソースコード 3.12 にソースコード 3.9 の変換結果を示す。この例では StubCodeGear のディスパッチを行う __code meta への goto の前に、Gearef マクロを使った context への書き戻しが行われている。GearsOS は CbC を用いて実装している為、スタックを持っていない。その為都度データは Context に書き戻す必要があるが、データの一時保管場所としても利用されている。

ソースコード 3.12: take を呼び出す部分の変換後

```
1 __code odgCommitCPUWorker3(struct Context *context,struct CPUWorker*
     worker, struct Context* task) {
2     int i = worker->loopCounter;
3     struct Queue* queue = GET_WAIT_LIST(task->data[task->odg+i]);
4     Gearef(context, Queue)->queue = (union Data*) queue;
5     Gearef(context, Queue)->next = C_odgCommitCPUWorker4;
6     goto meta(context, queue->take);
7 }
```

この性質から Interface には、Interface を実装する DataGear が持っておきたい変数はいれてはいけない。例えば Queue の実装では先頭要素を指し示す情報が必要であるが、 これを Interface 側の DataGear にしてしまうと、 呼び出し時に毎回更新されてしまう。常に持っておきたい値は、 Impl 側の DataGear の要素として定義する必要がある。

3.8.6 Stack.Stack->Stack.stack

Context のデータ保管場所について確認する。ここではソースコード 3.13 を実行する際に、 Context が持つ引数保管場所がどのような値になるかを見る。

ソースコード 3.13: Context の引数格納用の場所の確認の例題

```
1 Stack* stack = createSingleLinkedStack(context);
2 StackTest* stackTest = createStackTestImpl3(context);
3 Gearef(context, StackTest)->stackTest = (union Data*) stackTest;
4 Gearef(context, StackTest)->stack = stack;
5 Gearef(context, StackTest)->next = C_shutdown;
6 goto meta(context, stackTest->insertTest1);
```

ソースコード 3.14 の 1 行目で作製した StackTest Interface のアドレスは、 3 行目の出力である 0x7fff2f806b50 である。Interface が持つ実装へのポインタは、 5 行目の先頭の stackTest に代入されている値である 0x7fff2f806bb8 となっている。

ソースコード 3.14: gdb を使って作製した Interface のアドレスを確認する

```
1 50          StackTest* stackTest = createStackTestImpl3(context);
2 (gdb) p stackTest
```

```
3 $24 = (StackTest *) 0x7fff2f806b50
4 (gdb) p *stackTest
5 $25 = {stackTest = 0x7fff2f806bb8, stack = 0x0, data = 0x0, data1 = 0x0,
     insertTest1 = C_insertTest1StackTestImpl3,
6   insertTest2 = C_insertTest2StackTestImpl3, insertTest3 =
     C_insertTest3StackTestImpl3, insertTest4 = C_insertTest4StackTestImpl3
     ,
7   pop2Test = C_pop2TestStackTestImpl3, pop2Test1 =
     C_pop2Test1StackTestImpl3, next = C_checkAndSetAtomicReference}
```

ソースコード 3.15 の行を実行し、 context の引数格納用の場所に Interface のアドレスを書き込む。

ソースコード 3.15: context の引数格納用の場所に代入

```
1 (gdb)
2 51          Gearef(context, StackTest)->stackTest = (union Data*)
     stackTest;
```

ソースコード 3.16 では、 引数格納用の場所の値を確認する。これは union Data 型であるので、StackTest 型を指定して確認する。(4 行目)

出力結果の stackTest の値は 0x7fff2f806b50 である。(4 行目、8 行目) この値は Interface のアドレスと一致している。

ソースコード 3.16: context の引数格納用の場所の値を確認

```
1 (gdb) p context->data[D_StackTest]
2 $26 = (union Data *) 0x7fff2f806444
3 (gdb) p context->data[D_StackTest]->StackTest
4 $27 = {stackTest = 0x7fff2f806b50, stack = 0x0, data = 0x0, data1 = 0x0,
     insertTest1 = C_checkAndSetAtomicReference,
5   insertTest2 = C_checkAndSetAtomicReference, insertTest3 =
     C_checkAndSetAtomicReference, insertTest4 =
     C_checkAndSetAtomicReference,
6   pop2Test = C_checkAndSetAtomicReference, pop2Test1 =
     C_checkAndSetAtomicReference, next = C_checkAndSetAtomicReference}
7 (gdb) p context->data[D_StackTest]->StackTest.stackTest
8 $28 = (union Data *) 0x7fff2f806b50
```

ソースコード 3.17 で続けて stackTest フィールドが指す union Data のポインタを StackTest 型として指定し値を確認する。(1 行目) 2 行目で表示されている stackTest フィールドの値は、StackInterface が持つ実装へのアドレスと一致している。CbC 上では 5 行目のように、ドットでフィールドを指定すると値が取れる。

ソースコード 3.17: context の data 配列から Interface を取り出す

```
1 (gdb) p context->data[D_StackTest]->StackTest.stackTest->StackTest
2 $29 = {stackTest = 0x7fff2f806bb8, stack = 0x0, data = 0x0, data1 = 0x0,
     insertTest1 = C_insertTest1StackTestImpl3,
3   insertTest2 = C_insertTest2StackTestImpl3, insertTest3 =
     C_insertTest3StackTestImpl3, insertTest4 = C_insertTest4StackTestImpl3
     ,
```

```
4   pop2Test = C_pop2TestStackTestImpl3, pop2Test1 =
      C_pop2Test1StackTestImpl3, next = C_checkAndSetAtomicReference}
5 (gdb) p context->data[D_StackTest]->StackTest.stackTest->StackTest.
      stackTest
6 $30 = (union Data *) 0x7fff2f806bb8
```

この状況を纏めると、図 3.4 のようになる。Context の引数格納用の場所は、最初はその型の Interface へのポインタが、Interface が通常実装へのポインタを指すフィールドに書き込まれる。これは GearsOS では Stack.stack->Stack.stack 問題と言われている。

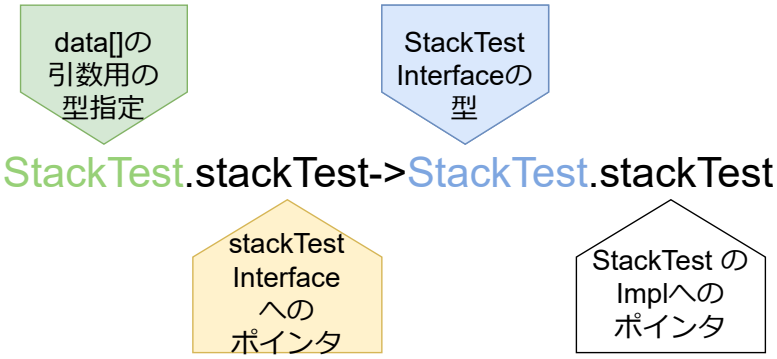

図 3.4: Stack.stack->Stack.stack

3.9 par goto

par goto とは GearsOS の並列処理用の構文である。ソースコード 3.18 では、 配列を初期化する createArray、 配列要素を 2 倍する twice、 配列の状況を出力する printArray を並列で動作させている。par goto した処理がそれぞれ終了すると、 code2 に継続する。

ソースコード 3.18: par goto の呼び出し

```
1  __code createTask1(struct LoopCounter* loopCounter, struct TaskManager*
       taskManager) {
2      Array* array1 = new Array();
3      Array* array2 = new Array();
4      Timer* timer = createTimerImpl(context);
5
6      par goto createArray(array1, timer, __exit);
7      par goto twice(array1, array2, iterate(split), __exit);
8      par goto printArray(array2, timer, __exit);
9      goto code2();
10 }
```

GearsOS で並列処理をする場合、 Context の複製などのメタレベルの計算を多数行わなければならない。ノーマルレベルからは通常の goto 文のように記述可能な構文を導入し、メタレベルとの分離を行っている。メタレベルに変換された結果をソースコード 3.19 に示す。`par goto` で CodeGear への継続を指定すると、`par goto` の数に応じて Context が生成される。CodeGear の実行は、割り当てられた Context が担当する。作製された Context は一度大本の Context の task フィールドにアドレスが書き込まれ、初期化が行われる。InputDataGear と OutputDataGear の Queue の用意を行った後に、context の TaskList に Element 各 DataGear は構造体の形で表現されている。Element はリスト構造を作製する際に使われるデータ構造で、 次の Element を取得できる。

各 CodeGear は入力の DataGear である InputDataGear の依存関係が解決したら実行され、OutputDataGear の状況を TaskManager に通知するようになっている。par goto の場合は TaskManager 関係のメタ計算を挟む必要があるため、 通常の goto meta ではなく、初回は goto parGotoMeta が実行される。

ソースコード 3.19: par goto のメタレベル計算

```
1  __code createTask1(struct Context *context,struct LoopCounter*
       loopCounter, struct TaskManager* taskManager) {
2      Array* array1 = &ALLOCATE(context, Array)->Array;
3      Array* array2 = &ALLOCATE(context, Array)->Array;
4      Timer* timer = createTimerImpl(context);
5
6      struct Element* element;
7                      context->task = NEW(struct Context);
8                      initContext(context->task);
9                      context->task->next = C_createArray;
10                     context->task->idgCount = 0;
11                     context->task->idg = context->task->dataNum;
12                     context->task->maxIdg = context->task->idg + 0;
13                     context->task->odg = context->task->maxIdg;
14                     context->task->maxOdg = context->task->odg + 2;
15     GET_META(array1)->wait = createSynchronizedQueue(context);
16     GET_META(timer)->wait = createSynchronizedQueue(context);
17     context->task->data[context->task->odg+0] = (union Data*)array1;
18     context->task->data[context->task->odg+1] = (union Data*)timer;
19                     element = &ALLOCATE(context, Element)->Element;
20                     element->data = (union Data*)context->task;
21                     element->next = context->taskList;
22                     context->taskList = element;
23                     context->task = NEW(struct Context);
24                     initContext(context->task);
25                     context->task->next = C_twice;
26                     context->task->idgCount = 1;
27                     context->task->idg = context->task->dataNum;
28                     context->task->maxIdg = context->task->idg + 1;
29                     context->task->odg = context->task->maxIdg;
30                     context->task->maxOdg = context->task->odg + 1;
31     context->task->iterate = 0;
32     context->task->iterator = createMultiDimIterator(context, split, 1,
```

```
    1);
33      GET_META(array1)->wait = createSynchronizedQueue(context);
34      GET_META(array2)->wait = createSynchronizedQueue(context);
35      context->task->data[context->task->idg+0] = (union Data*)array1;
36      context->task->data[context->task->odg+0] = (union Data*)array2;
37                      element = &ALLOCATE(context, Element)->Element;
38                      element->data = (union Data*)context->task;
39                      element->next = context->taskList;
40                      context->taskList = element;
41                      context->task = NEW(struct Context);
42                      initContext(context->task);
43                      context->task->next = C_printArray;
44                      context->task->idgCount = 2;
45                      context->task->idg = context->task->dataNum;
46                      context->task->maxIdg = context->task->idg + 2;
47                      context->task->odg = context->task->maxIdg;
48                      context->task->maxOdg = context->task->odg + 0;
49      GET_META(array2)->wait = createSynchronizedQueue(context);
50      GET_META(timer)->wait = createSynchronizedQueue(context);
51      context->task->data[context->task->idg+0] = (union Data*)array2;
52      context->task->data[context->task->idg+1] = (union Data*)timer;
53                      element = &ALLOCATE(context, Element)->Element;
54                      element->data = (union Data*)context->task;
55                      element->next = context->taskList;
56                      context->taskList = element;
57      Gearef(context, TaskManager)->taskList = context->taskList;
58      Gearef(context, TaskManager)->next1 = C_code2;
59      goto parGotoMeta(context, C_code2);
60  }
```

3.10 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 3.5 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で利用するソースコード群を CMake の設定ファイルである CMakeLists.txt に記述する。CMakeLists.txt では GearsOS のビルドに必要な一連の処理をマクロ `GearsCommand` で制御している。このマクロにプロジェクト名を `TARGET` として、 コンパイルしたいファイルを `SOURCES` に記述する。ソースコード 3.20 の例では `pop_and_push` が `TARGET` に指定されている。なおヘッダファイルは `SOURCES` に指定する必要はなく、 自動で解決される。

CMake 自身はコンパイルに必要なコマンドを実行することはなく、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 が実行時に必要とするファイルである Makefile、 build.ninja の生成までを担当する。

ソースコード 3.20: CMakeList.txt 内でのプロジェクト定義

```
GearsCommand(
  TARGET
  pop_and_push
  SOURCES
  examples/pop_and_push/main.cbc  examples/pop_and_push/StackTestImpl.cbc
    TaskManagerImpl.cbc CPUWorker.cbc SynchronizedQueue.cbc
   AtomicReference.cbc SingleLinkedStack.cbc examples/pop_and_push/
   StackTest2Impl.cbc examples/pop_and_push/StackTestImpl3.cbc
)
```

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。

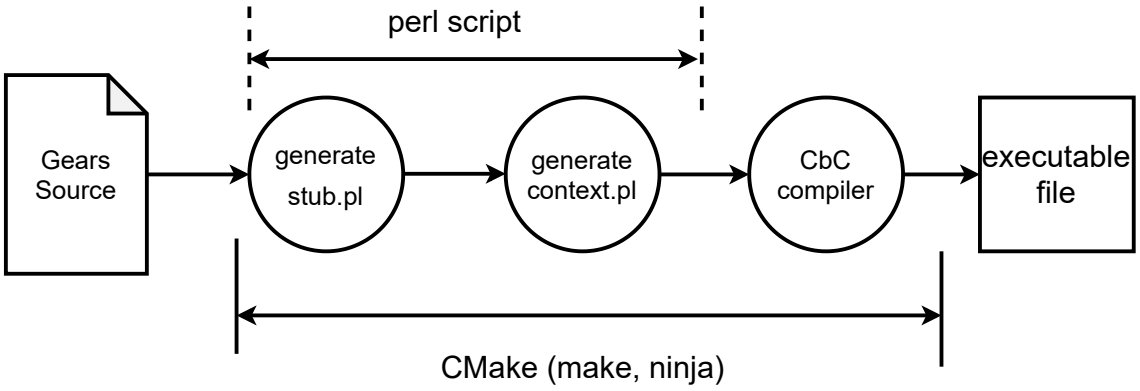

図 3.5: GearsOS のビルドフロー

3.11　GearsOS の CbC から純粋な CbC への変換

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。

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 3.21)

ソースコード 3.21: CMakeList.txt 内での Perl の実行部分

```
macro( GearsCommand )
    set( _OPTIONS_ARGS )
    set( _ONE_VALUE_ARGS TARGET )
    set( _MULTI_VALUE_ARGS SOURCES )
    cmake_parse_arguments( _Gears "${_OPTIONS_ARGS}" "${_ONE_VALUE_ARGS}"
    "${_MULTI_VALUE_ARGS}" ${ARGN} )

    set (_Gears_CSOURCES)
    foreach(i ${_Gears_SOURCES})
        if (${i} MATCHES "\\.cbc")
            string(REGEX REPLACE "(.*).cbc" "c/\\1.c" j ${i})
            add_custom_command (
                OUTPUT    ${j}
                DEPENDS   ${i}
                COMMAND  "perl" "generate_stub.pl" "-o" ${j} ${i}
            )
        elseif (${i} MATCHES "\\.cu")
            string(REGEX REPLACE "(.*).cu" "c/\\1.ptx" j ${i})
            add_custom_command (
                OUTPUT    ${j}
                DEPENDS   ${i}
                COMMAND  nvcc ${NVCCFLAG} -c -ptx -o ${j} ${i}
            )
        else()
            set(j ${i})
        endif()
        list(APPEND _Gears_CSOURCES ${j})
    endforeach(i)
```

3.12 generate_stub.pl

generate_stub.pl は各 CbC ファイルごとに呼び出される。図 3.6 に generate_stub.pl を使った処理の概要を示す。ユーザーが記述した GearsOS の CbC ファイルは、 ノーマルレベルのコードである。generate_stub.pl は、 CbC ファイルにメタレベルの情報を付け加え、GearsOS の拡張構文を取り除いた結果の CbC ファイルを新たに生成する。返還前の GearsOS のファイルの拡張子は.cbc であるが、 generate_stub.pl によって変換されると、同名で拡張子のみ.c に切り替わったファイルが生成される。拡張子は.c であるが、中身は CbC で記述されている。generate_stub.pl はあるプログラムのソースコードから別のプログラムのソースコードを生成するトランスパイラとして見ることができる。

generate_stub.pl は GearsOS のソースコードを 2 回読む。1 度目の読み込みは関数 getDataGear が担当する。(図 3.7) ソースコード中に登場する CodeGear と、CodeGear の入出力を検知する。この際に`#interface` 構文で Interface の利用が確認された場合、 Interface の定義ファイルを開き、getDataGear をさらに呼び出す。これらの処理で、 1 つの

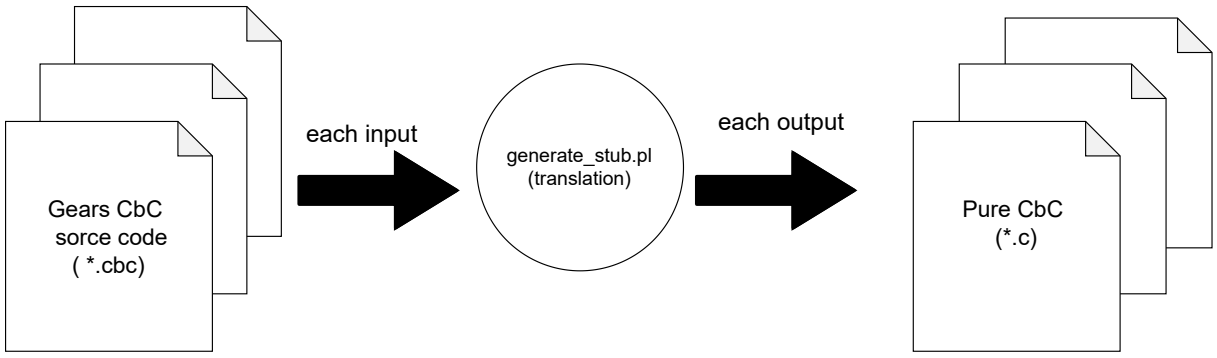

図 3.6: generate_sub.pl を使ったトランスコンパイル

GearsOS のファイル自身と、 呼び出している Interface の定義ファイルに実装されている CodeGear と DataGear の組を取得する。データの収集は、入力で与えられたファイルを 1 行ずつ読み込み、 あらかじめ設定した正規表現にパターンマッチすることで処理を行っている。ソースコード 3.22 は、GearsOS のソースコード中で Interface の使用宣言部分のパターンマッチ処理である。

ソースコード 3.22: CMakeList.txt 内での Perl の実行部分

```
1 } elsif(/^#interface "(.*)"/) {
2     debug_print("getDataGear",__LINE__, $_) if $opt_debug;
3     # use interface
4     my $interfaceHeader = $1;
5     next if ($interfaceHeader =~ /context.h/);
6     $interfaceHeader =~ m|(\w+)\.\w+$|; #remove filename extension
7     my $interfaceName = $1;
8     includeInterface(\%call_interfaces, $filename, $interfaceName,
    $headerNameToInfo);
```

ここでは使用している Interface の名前を変数`$interfaceHeader`にキャプチャし、Interface の読み込み処理を 8 行目の `includeInterface` 関数で実行している。includeInterface 関数の中では getDataGear を呼び出し、CodeGear、DataGear の情報を取得している。

取得した情報は generate_stub.pl が持つ変数に書き込まれる。この変数は主に Perl のハッシュ (連想配列) が利用される。

1 度ファイルを完全に読み込み、 CodeGear、DataGear の情報を取得し終わると、以降はその情報をもとに変換したファイルを書き出す。この書き出しは generateDataGear 関数で行われる。(図 3.8) ファイルを書き出す際は、 元の CbC ファイルを再読み込みし、変換する必要があるキーワードが出現するまでは、変換後のファイルに転記を行う。例えば各 CodeGear の最後に実行される `goto` 文は、 GearsOS の場合は MetaCodeGear に継続するように、 対象を切り替える必要がある。この為に generate_stub.pl は、goto 文を検知すると context 経由で引数のやりとりをするメタ処理を付け加える。また、すべての

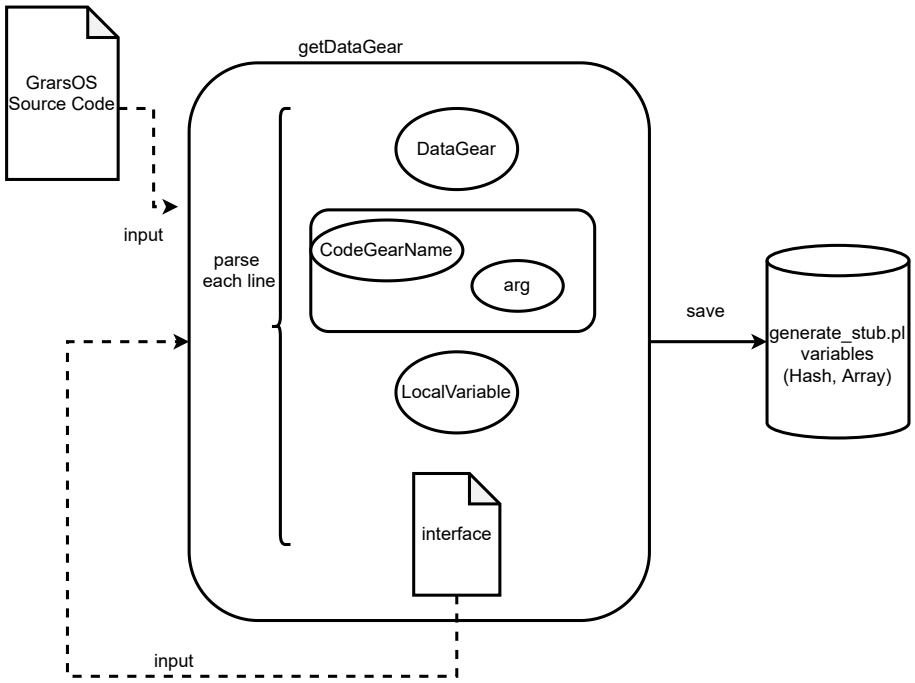

図 3.7: getDataGear の処理の概要

CodeGear は context を入力として受け取る必要があるため、 引数を書き換えて Context を付け加えている。

generate_stub.pl は Perl で書かれたトランスパイラであり、 C 言語のコンパイラのように文字列を字句解析、構文解析をする訳ではない。いくつかあらかじめ定義した正規表現パターンに読み込んでいる CbC ファイルの行がパターンマッチされたら、 特定の処理をする様に実装されている。

CodeGear の入力を context から取り出す StubCodeGear の生成も generate_stub.pl で行う。なおすでに StubCodeGear が実装されていた場合は、 generate_stub.pl は StubCodeGear は生成しない。

3.13　generate_context.pl

generate_context.pl は、Context の初期化関連のファイルを生成する Perl スクリプトである。Context を初期化するためには、下記の処理をしなければならない。

- CodeGear のリストに StubCodeGear のアドレスの代入
- goto meta 時に引数を格納する data 配列のアロケーション

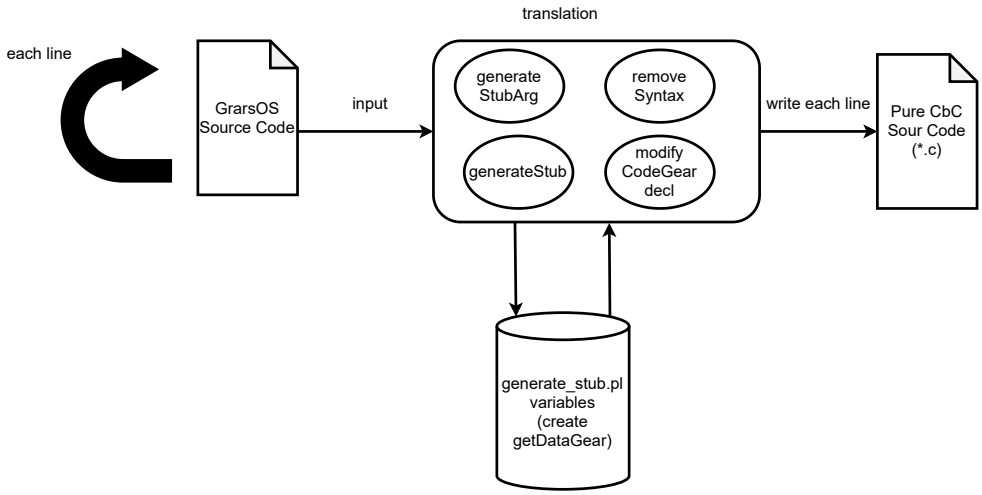

図 3.8: generateDataGear の処理の概要

- 計算で使用するすべての DataGear、CodeGear に対して番号を割り振り、 enum を作製する
- コンストラクタ関数の extern の生成

これらの記述は煩雑であるものの、CbC ファイルと DataGear の情報が纏められた context.h を見れば、記述すべき内容は一意に決定でき為、自動化が可能である。generate_context.pl は、 context.h を読み、まず DataGear の取得を行う。CodeGear は、generate_stub.pl で変換された CbC ファイルを読み込み、 `__code` があるものを CodeGear として判断する。また、C の一般関数でも `create` が関数名に含まれており、ポインタ型を返す関数は Interface のコンストラクタとして判断する。これらの情報をもとに、 CodeGear、DataGear の番号を作製し、enumCode.h と enumData.h として書き出す。

3.14　CbC xv6

CbC xv6 は GearsOS のシステムを利用して xv6 OS の置き換えを目指しているプロジェクトである。[19] xv6 は v6 OS[20] を x86 アーキテクチャ用に MIT によって実装し直されたものである。Raspberry Pi 上での動作を目指しているため、 ARM アーキテクチャ用に改良されたバージョンを利用している。[21]

書き換えにおいてはビルドシステムは CMake を利用し、 Perl クロスコンパイラを導入してたりと GearsOS のビルドシステムとほぼ同じシステムを利用している。GearsOS を

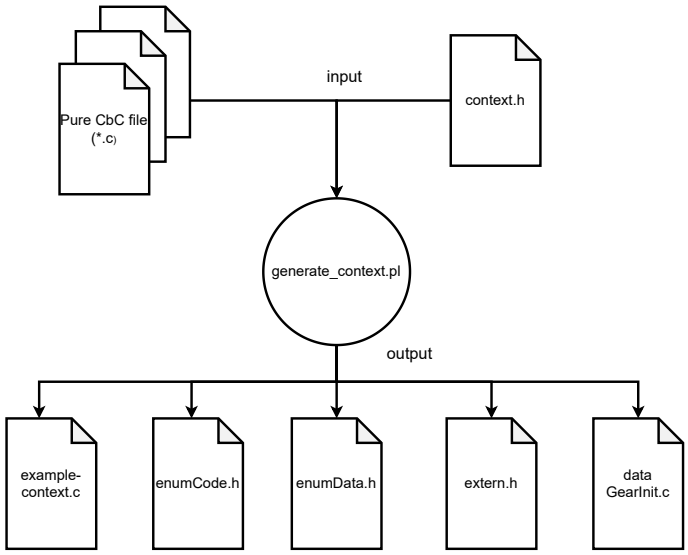

図 3.9: generate_context.pl を使ったファイル生成

使った比較的巨大な実用的なアプリケーションであるため、 xv6 の書き換えを進むに連れて様々な面で必要な機能や課題が生まれている。

従来の研究では xv6 は CMake を使わずに Make を用いてクロスコンパイルしていた。現在は xv6 に GearsOS の CMake を使ったビルドシステムを導入している。ARM 用のバイナリが出力できる CbC コンパイラがある場合、x86 マシンから CMake を利用してクロスコンパイル可能となっている。

CMake を導入したことで GearsOS の拡張構文が使えるようになった。xv6 は UNIX OS である為プロセス単位で処理を行っていたが、 ここに部分的に Context を導入した。xv6 では割り込みのフラグなどを大域変数として使っていた。GearsOS で実装する場合は DataGear 単位になるため、 これらのフラグも DataGear の形で実装し直した。この DataGear は各プロセスに対応する Context ではなく、中心的な Context がシングルトンで持っている必要がある。CbCxv6 の実装を通して Kernel の状況を記録しておく Context、つまり KernelContext が必要であることなどが判明した。

3.15　ARM用ビルドシステムの作製

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[22] 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可能に工夫をしなければならない。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。CMake 側に使用したいコンパイラの情報を渡せれば、以降は CMake 側が自動的に適切なビルドスクリプトを生成してくれる。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 3.10 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。

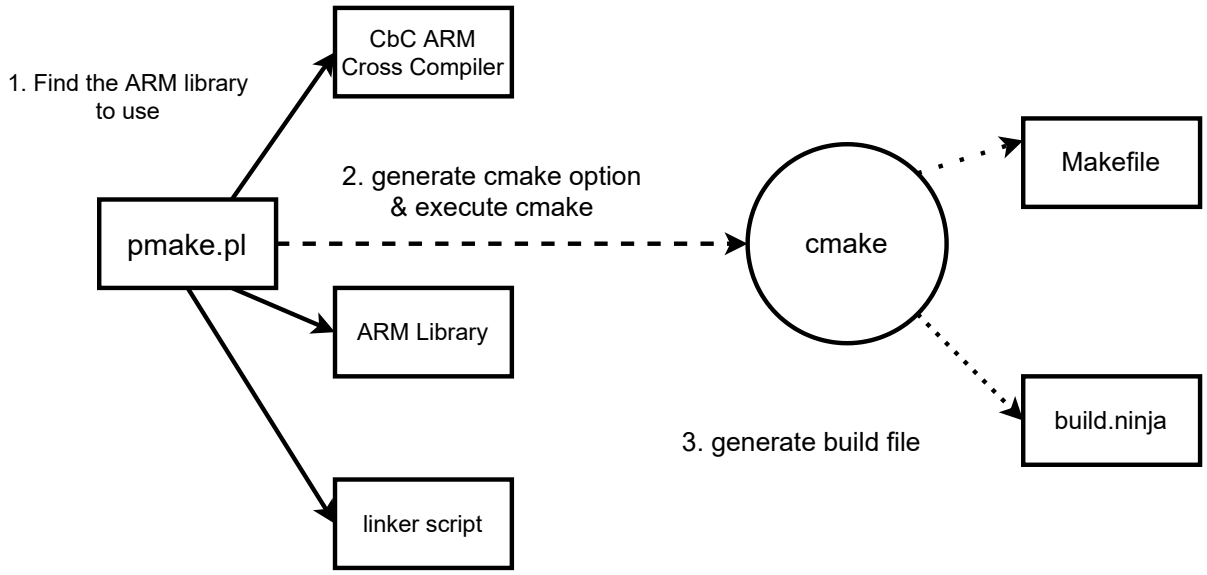

図 3.10: pmake.pl の処理フロー

第4章　新しくGearsOSに導入された機能の概要

本章では以降の章で解説するGearsOSに本研究を通して導入した機能について解説する。機能の詳細は以降の章の確認されたい。

4.1　ARMクロスコンパイル用のCMakeの定義

ARM用のアーキテクチャに向けてクロスコンパイルするCMakeを定義した。これによってGearsOSのビルドシステムに手を加えずにクロスコンパイルが可能となった。

4.2　Interface構文の簡素化

従来APIの定義とCodeGearの定義を別で記述する必要があった箇所を、より簡潔に記述できるように定義した。詳細は5.1章で述べる。

4.3　Interfaceの実装の型の導入

従来はInterfaceの実装の型は、プログラマが自分でメタ情報に変換し、Contextに書き込む必要があった。また、Interfaceの定義ファイルはあるものの、 Interfaceの実装は型定義ファイルが無かった。本研究では型定義ファイルを導入し、メタ情報の自動書き込み機能を実装した。詳細は5.2章などで述べる。

4.4　Interfaceで未定義のAPIの検知

従来のInterfaceは、実装していないAPI(CodeGear)があってもPerlスクリプトはCbCコードの変換をしてしまった。CbCコードの変換をする前に、 Perlスクリプトレベルで未定義のAPIを検知するようになった。詳細は5.14章で述べる。

4.5　Interfaceの引数の確認

Interface の API 呼び出しは、`goto meta` に変換されてしまうので、引数の数が揃っていない場合の確認が CbC レベルでは出来なかった。Perl スクリプトレベルで引数の数を確認するように実装し、GearsOS のプログラミングの安全性が向上した。詳細は 5.12 章で述べる。

4.6　InterfaceにないAPIの呼び出しの検知

Interface にない API の呼び出しも、従来は CbC に変換されコンパイルが走らないと解らなかった。Perl スクリプトレベルで API 呼び出しの度に Interface に定義があるかの確認を行うように改善した。詳細は 5.13 章で述べる。

4.7　別のInterfaceからの出力の取得

従来の GearsOS の Interface では、入出力は自分の Interface 内で完結している必要があった。このため、Stack Interface の出力を Other Interface で受け取る際に、自分で Stub を実装する必要があった。この別の Interface の入力を受け取る Stub の作製を自動化した。詳細は 6.6 章で述べる。

4.8　Interfaceの雛形ファイルの作製スクリプトの導入

満たすべき Interface と、満たしたい型が決定しても、従来は CodeGear の定義やコンストラクタをすべて手書きする必要があった。これらはバグの元であった為に、自動的に雛形ファイルを作製するスクリプトを導入した。詳細は 5.11 章で述べる。

4.9　実装のCodeGear名からメタ情報の切り離し

Interface の各 API である CodeGear は、従来は実装する際にプログラマが実装の型の名前を CodeGear の名前の末尾につける必要があった。この型名の情報はメタ情報であるため、 本研究では CodeGear の宣言時に型名を末尾につけず、コンパイル時にトランスパイラ側で変換するように実装した。詳細は 5.8 章で述べる。

4.10　自由な MetaCodeGear の作製、継続の入れ替え機能

従来は CodeGear の実行の後に継続する MetaCodeGear は__code meta で決め打ちだった。ユーザーが定義した MetaCodeGear に継続できる機能を追加した。詳細は 6.5 章で述べる。

4.11　Perl トランスパイラの変換ルーチンのデバッグ機能の追加

Perl トランスパイラでメタ計算を作製しているが、この Perl スクリプトは巨大な正規表現マッチのループで構成されている。変換したい CbC ファイルがどの行の正規表現パターンにマッチしたかを可視化できる機能を追加した。詳細は 6.9 章で述べる。

4.12　DataGear の型集合ファイルである context.h の自動生成

従来は Interface、Impl の型を定義し、使いたい DataGear を決めると、手動で context.h に DataGear の型を記述する必要があった。この型情報はコンパイル時に決定するので、自動化が可能である。その為 Perl トランスパイラを使い、自動的に生成する機能を実装した。詳細は 6.4 章で述べる。

4.13　GearsOS の初期化ルーチンの自動生成

GearsOS の例題を作製する際に必要な初期化ルーチンを自動生成するシンタックスを導入した。これによって GearsOS の例題を作製する際に、コピーアンドペーストで行っていた関数定義を省略することができた。詳細は 6.10 章で述べる。

4.14　ジェネリクスのサポート

GearsOS で型変数を使い、様々な型に対応するコードを生成できるジェネリクス機能を追加した。これによって型ごとに CodeGear の定義をする必要がなくなり、処理の共通化が図れる。詳細は 6.8 章で述べる。

第5章　GearsOSのInterfaceの改良

GearsOS のモジュール化の仕組みである Interface は、 GearsOS の中心的な機能である。Interface の取り扱いには様々なメタ計算が含まれ、このメタ計算は Perl スクリプトによって生成される。

Interface を GearsOS で使ったプログラミングをするにつれて、様々な不足している機能や、改善すべき点が見つかった。また Perl スクリプトが Interface を適切に取り扱う為の API も必要となることが分かった。本章では本研究で行った GearsOS の Interface の改良について述べる。

5.1　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。ソースコード 5.1 は Stack の Interface の例である。

ソースコード 5.1: 従来の Stack Interface

```
1  typedef struct Stack<Type, Impl>{
2          union Data* stack;
3          union Data* data;
4          union Data* data1;
5          /* Type* stack; */
6          /* Type* data; */
7          /* Type* data1; */
8          __code whenEmpty(...);
9          __code clear(Impl* stack,__code next(...));
10         __code push(Impl* stack,Type* data, __code next(...));
11         __code pop(Impl* stack, __code next(Type* data, ...));
12         __code pop2(Impl* stack, __code next(Type* data, Type* data1,
       ...));
13         __code isEmpty(Impl* stack, __code next(...), __code whenEmpty
       (...));
14         __code get(Impl* stack, __code next(Type* data, ...));
15         __code get2(Impl* stack, __code next(Type* data, Type* data1,
       ...));
16         __code next(...);
```

```
17 } Stack;
```

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 を使うプログラミングスタイルを取るケースも見られた。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 5.2: golang の interface 宣言

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

GearsOS の Interface は入力と出力の API を定義するものであるので、golang の Interface のように、関数の API を並べて記述するほうが簡潔であると考えた。改良した Interface の構文で Stack を定義したものをソースコード 5.3 に示す。

ソースコード 5.3: 変更後の Stack Interface

```
1  typedef struct Stack<>{
2          __code clear(Impl* stack,__code next(...));
3          __code push(Impl* stack,union Data* data, __code next(...));
4          __code pop(Impl* stack, __code next(union Data* data, ...));
5          __code pop2(Impl* stack, __code next(union Data* data, union Data
       * data1, ...));
6          __code isEmpty(Impl* stack, __code next(...), __code whenEmpty
       (...));
7          __code get(Impl* stack, __code next(union Data* data, ...));
8          __code get2(Impl* stack, __code next(union Data* data, union Data
       * data1, ...));
9          __code next(...);
10         __code whenEmpty(...);
11 } Stack;
```

従来の Interface では<Type, Impl>キーワードが含まれていた。これはジェネリクスの機能を意識して導入された構文である。Impl キーワードは実装自身の型を示す型変換とし

て使われていた。しかし基本Interfaceの定義を行う際にGearsOSのシステム上、CodeGearの第一引数は Impl 型のポインタが来る。これはオブジェクト指向言語で言う self に相当するものであり、 自分自身のインスタンスを示すポインタである。Impl キーワードは共通して使用されるために、 宣言部分からは取り外し、デフォルトの型キーワードとして定義した。Type キーワードは型変数としての利用を意識して導入されていたが、現在までの GearsOS の例題では導入されていなかった。ジェネリクスとしての型変数の利用の場合は T などの 1 文字変数がよく使われる。変更後の構文ではのちのジェネリクス導入のことを踏まえて、Type キーワードは削除した。

構文を変更するには、 GearsOS のビルドシステム上で Interface を利用している箇所を修正する必要がある。Interface は generate_stub.pl で読み込まれ、 CodeGear と入出力の DataGear の数え上げが行われる。この処理は Interface のパースに相当するものである。パース対象の Interface の構文は、変更前の構文にしか対応していなかった。後方互換性を維持したまま、新しい構文に対応させるために、generate_stub.pl が利用する Interface の解析ルーチンを両方の構文に対応させた。

5.2　Implement の型定義ファイルの導入

Interface を使う言語では、 Interface が決まるとこれを実装するクラスや型が生まれる。GearsOS も Interface に対応する実装が存在する。例えば Stack Interface の実装は SingleLinkedStack であり、 Queue の実装は SingleLinkedQueue や SynchronizedQueue が存在する。

この SynchronizedQueue は GearsOS では DataGear として扱われる。Interface の定義と同等な型定義ファイルが、 実装の型については存在しなかった。従来は context.h の DataGear の宣言部分に、構造体の形式で表現したものを手で記述していた。(ソースコード 5.4)

ソースコード 5.4: cotnext.h に直接書かれた型定義

```
1  union Data {
2     /* 略 */
3      // Queue Interface
4      struct Queue {
5          union Data* queue;
6          union Data* data;
7          enum Code whenEmpty;
8          enum Code clear;
9          enum Code put;
10         enum Code take;
11         enum Code isEmpty;
12         enum Code next;
13     } Queue;
14     struct SingleLinkedQueue {
```

```
15          struct Element* top;
16          struct Element* last;
17      } SingleLinkedQueue;
18      struct SynchronizedQueue {
19          struct Element* top;
20          struct Element* last;
21          struct Atomic* atomic;
22      } SynchronizedQueue;
23     /* 略 */
24 };
```

CbC ファイルからは context.h をインクルードすることで問題なく型の使用は可能である。Perl のトランスパイラである generate_stub.pl は Interface の型定義ファイルをパースしていた。しかし型定義ファイルの存在の有無が Interface と実装で異なっている為に、generate_stub.pl で Implement の型に関する操作ができない。Implement の型も同様に定義ファイルを作製すれば、generate_stub.pl で型定義を用いた様々な処理が可能となり、ビルドシステムが柔軟な挙動が可能となる。また型定義は一貫して*.h に記述すれば良くなるため、 プログラマの見通しも良くなる。本研究では新たに Implement の型定義ファイルを考案する。

GearsOS ではすでに Interface の型定義ファイルを持っている。Implement の型定義ファイルも、 Interface の型定義ファイルと似たシンタックスにしたい。Implement の型定義ファイルで持たなければいけないのは、 どの Interface を実装しているかの情報である。この情報は他言語では Interface の実装を持つ型の宣言時に記述するケースと、型名の記述はせずに言語システムが実装しているかどうかを確認するケースが存在する。Java では implements キーワードを用いてどの Interface を実装しているかを記述する。[23] ソースコード 5.5 では、Pig クラスは Animal Interface を実装している。

ソースコード 5.5: Java の Implement キーワード

```
1  // interface
2  interface Animal {
3    public void animalSound(); // interface method (does not have a body)
4    public void sleep(); // interface method (does not have a body)
5  }
6
7  // Pig "implements" the Animal interface
8  class Pig implements Animal {
9    public void animalSound() {
10     // The body of animalSound() is provided here
11     System.out.println("The pig says: wee wee");
12   }
13   public void sleep() {
14     // The body of sleep() is provided here
15     System.out.println("Zzz");
16   }
17 }
```

golang では Interface の実装は特にキーワードを指定せずに、 その Interface で定義しているメソッドを、Implement に相当する構造体がすべて実装しているかどうかでチェックされる。これは golang はクラスを持たず、構造体を使って Interface の実装を行う為に、構造体の定義にどの Interface の実装であるかの情報をシンタックス上書けない為である。GearsOS では型定義ファイルを持つことができるために、 golang のような実行時チェックは行わず、 Java に近い形で表現したい。

導入した型定義で SynchronizedQueue を定義したものをソースコード 5.6 に示す。大まかな定義方法は Interface 定義のものと同様である。違いとして `impl` キーワードを導入した。これは Java の `implements` に相当する機能であり、 実装した Interface の名前を記述する。現状の GearsOS では Impl が持てる Interface は 1 つのみであるため、`impl` の後ろにはただ 1 つの型が書かれる。型定義の中では独自に定義した CodeGear を書いてもいい。これは Java のプライベートメソッドに相当するものである。特にプライベートメソッドがない場合は、 実装側で所持したい変数定義を記述する。SynchronizedQueue の例では `top` などが実装側で所持している変数である。

ソースコード 5.6: SynchronizedQueue の定義ファイル

```
1 typedef struct SynchronizedQueue <> impl Queue {
2   struct Element* top;
3   struct Element* last;
4   struct Atomic* atomic;
5 } SynchronizedQueue;
```

従来 context.h に直接記述していたすべての DataGear の定義は、 スクリプトで機械的に Interface および Implement の型定義ファイルに変換を行った。

context.h から Interface および Implement の型定義をファイルに分割することができた。しかし GearsOS の Context はすべての DataGear の型定義を持つ必要がある。この為、context.h には分割した型定義ファイルをもとに、CbC のメタレベルに変換された型情報を書き込む必要がある。この処理は generate_context.pl 内でビルド時に行うようにした。

5.3 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。

Implement の CodeGear から内部で goto する CodeGear の場合は引数として Impl 内部のデータ型は取り出すことができない。GearsOS では goto 文は、すべて goto meta に書き変わる。goto meta が発行されると Stub Code Gear に継続するが、現在のシステムでは Interface から値を Stub で取得する。Implement の内部の引数はこの時点では Stub から取得することはできず、 実装側の CodeGear の内部から取り出す必要がある。

Stub から Impl 内部のデータを取得しようと、 Impl を Interface と見立てて GearsOS のプログラミングをしたことがあった。ソースコード 5.7 では、fs Interface に対する実装の fs_impl を、 あたかも Interface のように見せるハックである。generate_stub.pl はコンストラクタがあり、#interface 構文で呼ばれていたら、対象の DataGear が Interface であると判断する。この場合はコンストラクタに対応する文字列をコメントとして書いている為、StubCodeGear は Impl を Interface だと思い込む。よって、 Stub で Context 内部の Impl の置き場所から値を取得するようになった。しかし、この誤魔化しはメタレベルとノーマルレベルの分離どころではなく、GearsOS の設計に反する記述であった。

ソースコード 5.7: Impl を Interface のようにふるまわせる為に、コンストラクタを偽装した例

```
1
2  /*
3  fs_impl* createfs_impl2();
4  */
5
6  __code allocinode(struct fs_impl* fs_impl, uint dev, struct superblock*
       sb, __code next(...)){
7
8      readsb(dev, sb);
9      Gearef(cbc_context, fs_impl)->inum = 1;
10     goto allocinode_loopcheck(fs_impl, inum, dev, sb, bp, dip, next(...))
       ;
11
12 }
```

これはそもそも、Impl の内部で持つ値は DataGear ではなく、DataGear に含まれる値であるということを意識出来なかった為である。GearsOS はデータの単位は DataGear で行われる。Interface の呼び出し時に使われる引数は DataGear として処理されるが、Impl はそもそも Impl 全体が 1 つの DataGear であるため、Stub 経由で値をとるのは間違っていたのであった。Impl の値を CodeGear の内部で使う場合は、第一引数で与えられる自分自身の DataGear の参照から取り出す必要がある。

5.4　Interfaceのパーサーの構築

従来の GearsOS のトランスパイラでは、 generate_stub.pl が Interface ファイルを開き、情報を解析していた。この情報解析は getDataGear 関数で行われていた。しかしこの関数は、CbC ファイルの CodeGear、DataGear の解析で使用するルーチンと同じものである。この為 Interface 特有のパースが出来ていなかった。

また、開いたヘッダファイルが Interface のファイルでも、そうでない C のヘッダファイルでも同様の解析をしてしまう。Interface の定義ファイルの構文はすでに統一されたものを使用している。Interface の定義の構文で実装されていない Interface ファイルを読み込んだ場合は、 エラーとして処理したい。また、Interface が満たすべき CodeGear の種類や InputDataGear の数の管理も行いたい。さらに Interface ではなく、Implement の定義ファイルも同様にパースし、情報を解析したい。

これらを実現するには、今まで generate_stub.pl で使っていた情報解析ルーチンをもとに、最初から Interface に特化したパーサーが必要となる。本研究では Gears::Interface モジュールとして Interface のパーサーを実装した。

5.4.1　Gears::Interfaceの構成

Gears::Interface は Perl のモジュールであるが、 実際はパーサー用の API を提供しているサブルーチンのまとまりである。その為オブジェクトを作らずに直接メソッドを呼び出して利用する。Gears::Interface は 2 種類の API を提供している。

5.4.2　パース API

1 つは `parse` メソッドである。これはパースしたいファイル名を与えると、Interface であった場合にヘッダファイルをパースして情報を返す API である。parseAPI で Stack Interface をパースした結果の値をソースコード 5.8 に示す。これは Perl の連想配列のリファレンスで表現されている。

ソースコード 5.8: parseAPI でパースした Stack Interface

```
1  \ {
2      content             [
3          [0]  "enum Code clear;",
4          [1]  "union Data* stack;",
5          [2]  "enum Code push;",
6          [3]  "union Data* data;",
7          [4]  "enum Code pop;",
8          [5]  "enum Code pop2;",
9          [6]  "union Data* data1;",
10         [7]  "enum Code isEmpty;",
```

```
        [8]  "enum Code get;",
        [9]  "enum Code get2;",
        [10] "enum Code next;",
        [11] "enum Code whenEmpty;"
    ],
    file_name           "Stack.h",
    inner_code_gears    {
        next        1,
        whenEmpty   1
    },
    name                "Stack"
}
```

content が持つ要素は配列であり、 これは Interface を CbC の構造体に変換した際の内容である。file_name にはパースしたファイルのパスが入る。inner_code_gears は、 Interface が継続として受け取る CodeGear の集合が入っている。Stack では next と whenEmpty は入力で受け取るため、 inner_code_gears に格納されている。name はファイルパスではなく、 Interface の名前が格納されている。

Impl ファイルである SingleLinkedStack.h をパースした結果をソースコード 5.9 に示す。

ソースコード 5.9: parseAPI でパースした SingleLinkedStack

```
\ {
    content             [
        [0] "struct Element* top;"
    ],
    file_name           "plautogen/impl/SingleLinkedStack.h",
    inner_code_gears    {},
    isa                 "Stack",
    name                "SingleLinkedStack"
}
```

ほとんど返す値は Interface の時のものと同様であるが、 Impl の場合は isa キーに、実装している Interface の名前が格納される。

この情報はパース対象が Interface、もしくは Implement でなければ返さない。パーサーは Interface であるかどうかを、構文の正規表現にマッチするかどうかで確認をする。(ソースコード 5.10)

ソースコード 5.10: Interface であるかどうかの確認

```
sub isThisFileInterface {
  my ($class, $filename) = @_;

  open my $fh, '<', $filename;
  my $line = <$fh>; #read top line  ex Typedef struct Stack<Type, Impl> {

  return 0 unless ($line =~ /typedef struct \w+\s?<.*>([\s\w{]+)/);

  my $annotation = $1;
  return 0 if ($annotation =~ /impl/);
```

```
11
12    return 1;
13  }
```

5.4.3　詳細なパース API

parse API はシンプルな結果を返していたが、 Interface に定義している CodeGear の引数など、詳細な情報を取得したいケースがある。Gears::Interface に、詳細なパース用の API である `detailed_parse` API を用意した。先ほどの Stack Interface をパースした結果をソースコード 5.11 に示す。新たな情報として codeName が連想配列の要素に追加されている。(ソースコード 2 行目) codeName は CodeGear の名前がキーになっており、value として引数の文字列情報が `args` に、、 Interface の呼び出し時に必要な引数の個数が `argc` に設定される。これらの情報は配列 codes からもアクセス可能となっている。(ソースコード 48 行目) Interface が持つ DataGear の一覧は、配列 data に格納される。(ソースコード 62 行目)

OutputDataGear がある CodeGear の一覧が、 hasOutputArgs に格納される。(ソースコード 68 行目) codeName と同様に、CodeGear の名前がキーとなっている。対応する値は、 出力する変数の名前と、その型の組のリストになっている。

この詳細なパースの結果は、以下に例を示す用途で使われる。

- implement の CodeGear の名前の保管
- Interface の CodeGear の定義と実装の対応の確認
- OutputDataGear がある API 呼び出しであるかの確認
- API 呼び出し時の引数のチェック

ソースコード 5.11: Stack Interface の詳細なパース

```
1  \ {
2      codeName            {
3          clear       {
4              argc     1,
5              args     "Impl* stack,__code next(...)",
6              name     "clear"
7          },
8          get         {
9              argc     1,
10             args     "Impl* stack, __code next(union Data* data, ...)",
11             name     "get"
12         },
13         get2        {
```

```
14              argc   1,
15              args   "Impl* stack, __code next(union Data* data, union Data
    * data1, ...)",
16              name   "get2"
17          },
18          isEmpty   {
19              argc   2,
20              args   "Impl* stack, __code next(...), __code whenEmpty(...)
    ",
21              name   "isEmpty"
22          },
23          pop       {
24              argc   1,
25              args   "Impl* stack, __code next(union Data* data, ...)",
26              name   "pop"
27          },
28          pop2      {
29              argc   1,
30              args   "Impl* stack, __code next(union Data* data, union Data
    * data1, ...)",
31              name   "pop2"
32          },
33          push      {
34              argc   2,
35              args   "Impl* stack,union Data* data, __code next(...)",
36              name   "push"
37          }
38      },
39      codes               [
40          [0] var{codeName}{clear},
41          [1] var{codeName}{push},
42          [2] var{codeName}{pop},
43          [3] var{codeName}{pop2},
44          [4] var{codeName}{isEmpty},
45          [5] var{codeName}{get},
46          [6] var{codeName}{get2}
47      ],
48      content             [
49          [0]  "enum Code clear;",
50          [1]  "union Data* stack;",
51          [2]  "enum Code push;",
52          [3]  "union Data* data;",
53          [4]  "enum Code pop;",
54          [5]  "enum Code pop2;",
55          [6]  "union Data* data1;",
56          [7]  "enum Code isEmpty;",
57          [8]  "enum Code get;",
58          [9]  "enum Code get2;",
59          [10] "enum Code next;",
60          [11] "enum Code whenEmpty;"
61      ],
62      data                [
63          [0] "union Data* stack;",
```

```
64          [1] "union Data* data;",
65          [2] "union Data* data1;"
66      ],
67      file_name           "Stack.h",
68      hasOutputArgs       {
69          get     {
70              data    "Data*"
71          },
72          get2    {
73              data     "Data*",
74              Data     "union",
75              data1    "*"
76          },
77          pop     {
78              data    "Data*"
79          },
80          pop2    {
81              data     "Data*",
82              Data     "union",
83              data1    "*"
84          }
85      },
86      inner_code_gears    {
87          next        1,
88          whenEmpty   1
89      },
90      name                "Stack"
91  }
```

5.4.4　Interface パーサーの呼び出し

定義したパーサーは都度呼ぶこともできるが、ヘッダファイルのパスを入力で与える必要がある。generate_stub.pl は実行時のコマンドライン引数としてヘッダファイルは与えられないので、スクリプト中で探索する必要がある。毎回パースしたい Interface 名の探索をするのは煩雑である。

基本的に generate_stub.pl では Interface の名前がすでに判明しており、その Interface のパースした結果を取得したいのがほとんどである。ここからスクリプト内部で、Interface の名前とパースした結果を対応させる連想配列を実装した。generate_stub.pl では、スクリプト起動時に連想配列を変換処理を行う前に作製する。ソースコード 5.12 のサブルーチンが連想配列を作り出す処理である。このサブルーチンでは、ヘッダファイルを起動時に全探査し、 すべてパースを行う。4 行目で Gears::Util の API 呼び出しをしているが、この API は GearsOS で使うヘッダファイルを、指定されたパスから再帰的に探索するものである。

なお同名のヘッダファイルが見つかった場合は、 変換をしている CbC ファイルと同じディレクトリにあるヘッダファイルが優先される。(ソースコード 13 行目)

ソースコード 5.12: ヘッダファイルの名前と Interface のパース結果の対応リストの作製

```
sub createHeaderNameToInfo {
    my ($fn, $search_root) = @_;
    my $dirname = dirname $fn;
    my $files = Gears::Util->find_headers_from_path($search_root);
    my $interface_name2headerpath = {};

    #This process assumes that there are no header files of the same name
    for my $file (@{$files}) {
      if ($file =~ m|/(\w+)\.\w+$|) {
        my $file_name = $1;
        my $isInterface = Gears::Interface->isThisFileInterface($file);
        if (defined $interface_name2headerpath->{$file_name}) {
          next if ($file !~ m|$dirname|);
        }
        $interface_name2headerpath->{$file_name} = { path => $file,
    isInterface => $isInterface };
      }
    }

    return $interface_name2headerpath;
}
```

5.5　Interface 定義ファイル内での include のサポート

Interface で定義した API の引数として、別のヘッダファイルに含まれる構造体を使いたい場合がある。DataGear に変換するのが望ましいが、実装が煩雑になることが予想される場合などで、一度構造体を引数として利用するケースがあった。従来は Interface の定義ファイルの中身は generate_stub.pl が API の解析しか使っておらず、ここで C 言語の`#include` マクロを使っても情報が落とされてしまっていた。この為 Interface で別のヘッダファイルの型を使いたい場合は、手動で context.h に`#include` 文を書く必要があった。

これらの include 処理を、Perl トランスパイラ側で自動で行うように実装を行った。これに伴い、Interface、Impl の定義ファイルで使いたいヘッダファイルを include 文で指定することが可能となった。(ソースコード 5.13)

ソースコード 5.13: Include 文を書けるようになった Implement の宣言

```
#include "state_db.h"
#include "memory.h"
#include "TaskIterator.h"

typedef struct MCWorker <> {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
...
```

```
9   StateDB parent;
10  StateDB root;
11  ...
12  } MCWorker;
```

5.6 Interfaceの実装のCbCファイルへの構文の導入

今までの GearsOS ではマクロに似た#interface 構文で使用する Interfce 名を指定した。しかし Interface を実装する場合も、 Interface の API を利用する際も同じシンタックスであった。この 2 つは意味が異なっている為、 シンタックスを分離したい。Implement の型定義ファイルを導入したので、Interface の実装をする場合に別のシンタックスを導入する。

導入された構文をソースコード 5.14 に示す。この例では Stack Interface の実装として SingleLinkedStack を定義する宣言である。

Implement の宣言の構文では、 まず#impl の後ろに実装したい Inteface の名前を入れる。続く as キーワードの後ろに、Implement の型名を記述する。宣言は generate_stub.pl が読み取り、 変換した後の CbC ファイルからは該当する行が削除される。

ソースコード 5.14: DataGear の使用の宣言

```
1  #impl "Stack.h" as "SingleLinkedStack.h"
```

5.7 内部データ構造に利用するDataGearの使用構文の導入

Interface および Implment で使う DataGear は、通常引数の形で CodeGear に入力される。これ以外で、CodeGear の内部でデータ構造の表現の為に利用される DataGear が存在する。例えば SingleLinkedStack の実装では、Stack にいれる値の表現として ElementDataGear を使う。Element は入力で受け取った union Data 型を使って、CodeGer 内部で作製している。

ソースコード 5.15: CodeGear 内部で作られる Element DataGear

```
1  __code pushSingleLinkedStack(struct SingleLinkedStack* stack, union Data*
       data, __code next(...)) {
2      Element* element = new Element();
3      element->next = stack->top;
4      element->data = data;
5      stack->top = element;
6      goto next(...);
7  }
```

意味的には Interface としてではなく、DataGear そのものを使いたいケースであるため#interface 構文以外の新しい DataGear の使用を宣言する構文を導入したい。

DataGear を明示的に使うことを宣言する構文として#data を導入した。これは使用したい DataGear の定義ファイルを#data の後ろに記述する。SingleLinkedStack の場合の例をソースコード 5.16 に示す。この例では Node と Element を使用している。

ソースコード 5.16: DataGear の使用の宣言

```
1 #impl "Stack.h" as "SingleLinkedStack.h"
2 #data "Node.h"
3 #data "Element.h"
```

この宣言は、使用している DataGear を調査し context.h を作製する generate_context.pl が発見できなければならない。しかし generate_context.pl は、メタ情報を含む形に generate_stub.pl はトランスパイルした後に実行される。GearsOS で拡張したマクロの用な Interface 関連の宣言はここで消されてしまう。DataGear の使用の宣言は、コメントの形で変換後のファイルに書き出すようにし、generate_context.pl でも参照可能にした。SingleLinkedStack の例題に対応する変換後のコードを、ソースコード 5.17 に示す。

ソースコード 5.17: DataGear の使用の宣言の Perl による変換後

```
1 // include "Node.h"
2 // include "Element.h"
```

5.8 Interface API に対応した CodeGear の名前の自動変換

Interface の API に対応した CodeGear を実装する際、今までは暗黙に Interface 名と Impl の型名をつなげた名前で定義していた。例えば Stack Interface の API である pop を SingleLinkedStack が実装した場合、CodeGear の名前は popSingleLinkedStack にしていた。

CodeGear の名前につけられる Impl の名前は、GearsOS のシステムが CodeGear の識別に使うメタな情報と言える。ユーザーレベルでは Interface の API と同じ名前の CodeGear を実装できると、GearsOS のメタな CodeGear の処理と分離可能である。この為 Perl スクリプトで、Interface の Implement の場合は CodeGear の名前を自動で変換する機能を実装した。ここで変換する CodeGear の名前は、 CodeGear の定義部分と、継続で渡す CodeGear の名前の部分である。

generate_stub.pl はソースコードの情報を読み取るフェーズと、変換した情報を書き込むフェーズに別けられる。まずは読み取りの際の処理をソースコード 5.18 に示す。generate_stub.pl は、CodeGear の宣言時に、自分が今変換している Interface の Impl の CbC

ファイルかどうかをまず確認する。(ソースコード 6 行目) Impl の CbC ファイルであった場合、変数$implInterfaceInfo に具体的な値が入っているため、if 文に進む。if 文の中では Interface のパースの結果と、今定義している CodeGear の名前を比較する。ここで Interface の API である CodeGear の名前と、今 CbC ファイルで定義している CodeGear の名前が等しい場合、後ろに型名をつけた CodeGear の名前に変換し、スクリプト内で変換したことを記憶する。

ソースコード 5.18: CodeGear の名前が等しいかどうかの確認

```
1  } elsif (/^\_\_code (\w+)\((.*)\)(.*)/) {
2      debug_print("getDataGear",__LINE__, $_) if $opt_debug;
3      my $codeGearName = $1;
4      my $args = $2;
5
6      if ($implInterfaceInfo->{parsedInterfaceInfo}) {
7        if (exists_codegear_in_interface({codeGearName => $codeGearName,
       parsedInfo => $implInterfaceInfo->{parsedInterfaceInfo}})) {
8          my $replaceCodeGear = "${codeGearName}$implInterfaceInfo->{
       implementation}"; #${pop}SingleLinkedStack
9          $replaceCodeGearNames->{$codeGearName} = $replaceCodeGear;
10         $codeGearName = $replaceCodeGear;
11       }
12     }
```

CodeGear の宣言は、 Context を引数に含めるようにすでに書き換えるルーチンで処理されていた。この部分で CodeGear の名前を変更したい。実際に書き出している処理の部分をソースコード 5.19 に示す。この際に書き込む CodeGear の名前は、定義の CodeGear 名を正規表現でキャプチャし、変数$currentCodeGearName に代入している。(5 行目) 読み込み時に作製した、名前の変更があることを保存する連想配列$replaceCodeGearNames に、今書き込もうとしている CodeGear の名前を問い合わせる。連想配列側に CodeGear の名前に対応する値があった場合は書き換え対象なので、 $currentCodeGearName を、ソースコード 5.18 で作製した CodeGear の名前に変換する。

ソースコード 5.19: CodeGear の名前の変更

```
1  } elsif (/^\_\_code (\w+)\((.*)\)(.*)/) {
2      debug_print("generateDataGear",__LINE__, $_) if $opt_debug;
3      $inCode = 1;
4      %localVarType = ();
5      $currentCodeGearName = $1;
6      my $args = $2;
7      my $tail = $3;
8
9      #replace Code Gear Name to Implemenatation
10     if (exists $replaceCodeGearNames->{$currentCodeGearName}) {
11         $currentCodeGearName = $replaceCodeGearNames->{
       $currentCodeGearName};
12     }
```

実際に変換される様子を見る。ソースコード 5.20 は、 Phils Interface の実装のソースコードの一部である。Phils Interface にある pickup_lfork と、 eating の CodeGear の定義をしている。3 行目では checkAndSet に継続として、 pickup_rfork と eating を渡している。これらはそれぞれ Phils Interfce に定義がある CodeGear の名前であり、 このファイル中で実装している。

ソースコード 5.20: PhilsInterface の実装

```
1  __code pickup_lfork(struct PhilsImpl* phils, __code next(...)) {
2      struct AtomicT_int* left_fork = phils->Leftfork;
3      goto left_fork->checkAndSet(-1, phils->self, pickup_rfork, eating);
4
5  }
6
7  __code eating(struct PhilsImpl* phils, __code next(...)) {
8      printf("%d: eating\n", phils->self);
9      goto putdown_rfork();
10 }
```

変換された結果をソースコード 5.21 に示す。まず CodeGear の宣言時に名前の末尾に実装の型名である PhilsImpl がついている。Gearef マクロに変換されている為見づらいが、6 行目、7 行目で enum の定義に変換されて context に書き込まれているのが解る。

ソースコード 5.21: 変換された PhilsInterface の実装

```
1  __code pickup_lforkPhilsImpl(struct Context *context,struct PhilsImpl*
       phils, enum Code next) {
2      struct AtomicT_int* left_fork = phils->Leftfork;
3      Gearef(context, AtomicT_int)->atomicT_int = (union Data*) left_fork;
4      Gearef(context, AtomicT_int)->oldData = -1;
5      Gearef(context, AtomicT_int)->newData = phils->self;
6      Gearef(context, AtomicT_int)->next = C_pickup_rforkPhilsImpl;
7      Gearef(context, AtomicT_int)->fail = C_eatingPhilsImpl;
8      goto meta(context, left_fork->checkAndSet);
9
10 }
11
12 __code pickup_lforkPhilsImpl_stub(struct Context* context) {
13   PhilsImpl* phils = (PhilsImpl*)GearImpl(context, Phils, phils);
14   enum Code next = Gearef(context, Phils)->next;
15   goto pickup_lforkPhilsImpl(context, phils, next);
16 }
17
18 __code eatingPhilsImpl(struct Context *context,struct PhilsImpl* phils,
       enum Code next) {
19     printf("%d: eating\n", phils->self);
20     goto meta(context, C_putdown_rforkPhilsImpl);
21 }
22
23 __code eatingPhilsImpl_stub(struct Context* context) {
24   PhilsImpl* phils = (PhilsImpl*)GearImpl(context, Phils, phils);
```

```
25    enum Code next = Gearef(context, Phils)->next;
26    goto eatingPhilsImpl(context, phils, next);
27  }
```

5.9　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、 コンパイル時に下記の問題点が多発した。

- 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る。このため、トランス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。

5.10　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[24] 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。[25] 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`[26] 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`[27] 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の呼び出しや、 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。

5.11　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 5.1 に示す。

impl2cbc.pl では、 実行時引数に Implement の型定義ファイルを与える。impl2cbc.pl の内部で Implement のパース結果から Interface を特定し、 雛形を生成する。Interface の定義ファイルが複数見つかった場合、Implement の型定義ファイルがあるディレクトリと同じファイルが優先される。コマンドラインからの呼び出しと、生成した結果をソースコードに示す。

ソースコード 5.22: impl2cbc の実行方法

```
1 perl tools/impl2cbc.pl examples/DPP2/PhilsImpl.h
```

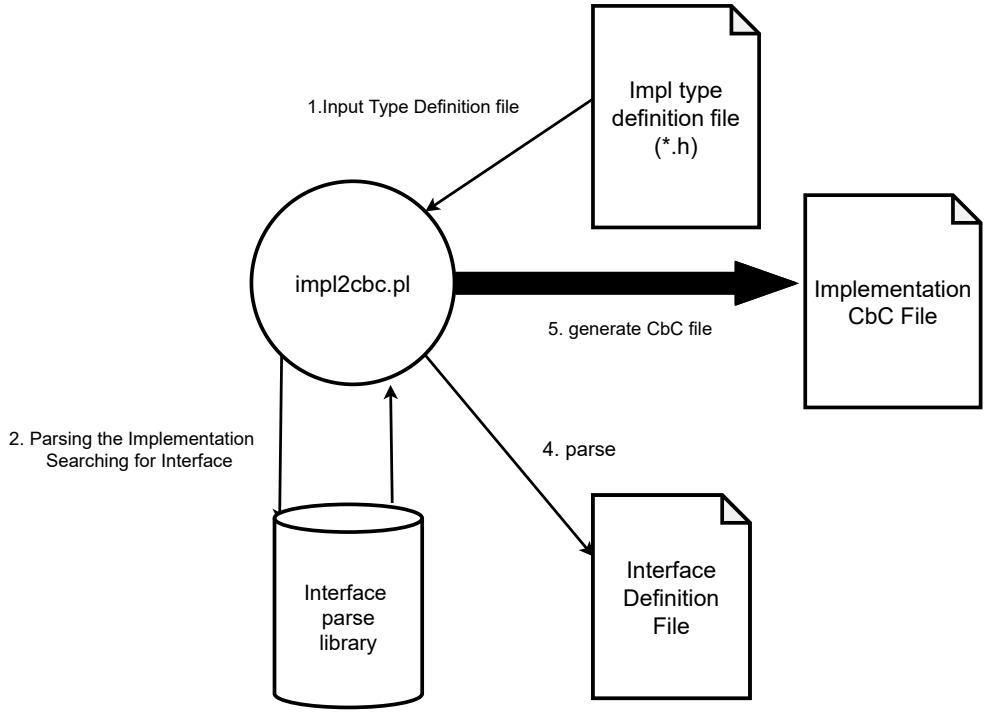

図 5.1: impl2cbc の処理の流れ

ソースコード 5.23: 生成された雛形ファイル

```
#include "../../../context.h"
#interface "Phils.h"

// ----
// typedef struct PhilsImpl <> impl Phils {
//   int self;
//   struct AtomicT_int* Leftfork;
//   struct AtomicT_int* Rightfork;
//   __code next(...);
// } PhilsImpl;
// ----

Phils* createPhilsImpl(struct Context* context) {
    struct Phils* phils  = new Phils();
    struct PhilsImpl* phils_impl = new PhilsImpl();
    phils->phils = (union Data*)phils_impl;
    phils_impl->self = 0;
    phils_impl->Leftfork = NULL;
    phils_impl->Rightfork = NULL;
    phils->putdown_lfork = C_putdown_lforkPhilsImpl;
    phils->putdown_rfork = C_putdown_rforkPhilsImpl;
    phils->thinking = C_thinkingPhilsImpl;
    phils->pickup_rfork = C_pickup_rforkPhilsImpl;
    phils->pickup_lfork = C_pickup_lforkPhilsImpl;
    phils->eating = C_eatingPhilsImpl;
    return phils;
}
__code putdown_lfork(struct PhilsImpl* phils, __code next(...)) {

    goto next(...);
}

__code putdown_rfork(struct PhilsImpl* phils, __code next(...)) {

    goto next(...);
}

__code thinking(struct PhilsImpl* phils, __code next(...)) {

    goto next(...);
}

__code pickup_rfork(struct PhilsImpl* phils, __code next(...)) {

    goto next(...);
}

__code pickup_lfork(struct PhilsImpl* phils, __code next(...)) {

    goto next(...);
}

```

```
53 __code eating(struct PhilsImpl* phils, __code next(...)) {
54
55     goto next(...);
56 }
```

5.11.1　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。

5.11.2　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。generate_stub.pl 内で作製する場合は、 すでにメタ情報を含むコードに書き換えたものを作製する。その為厳密には同じコードを生成する訳ではない。

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。

5.12　Interface の引数の数の確認

GearsOS のノーマルレベルでは、Interface の API の呼び出しは `interface->method(arg)` の呼び出し方であった。`arg` は引数であり、これは Interface で定義した API の引数の一致している必要がある。Interface の定義の引数は、Impl の実装自身が第一引数でくる制約があった。この制約の為に、厳密には Interface の定義ファイルに書かれている CodeGear の引数と、Interface の呼び出しの引数は数が揃ってはいない。generate_stub.pl は第一引数が実装自身の型であるので、union Data 型にキャストし、Context の引数保存場所に書き込むようになっている。問題が第 1 引数以外の引数が揃っていない場合である。generate_stub.pl を通すと、次の継続は goto meta に変換されてしまい、引数情報が抜けてしまう。その為引数はすべて適切に context に書き込まれている必要があるが、 一部引数が足りず書き込みが出来なかったケースでも、 CbC コンパイラレベルでは引数関係のエラーが発生しない。また上手く Interface の入力の数を取得できなかった場合も、generate_stub.pl は止まらずにマクロを生成してしまう。Gearef を通して context に書き込む右辺値が抜けているコードなどがよく発生した。この場合は原因を.c ファイルと.cbc ファイル、Interface ファイル、context ファイルのすべてを確認しなければならず、デバッグが非常に困難だった。Interface の API 呼び出し時の引数検知は、 Interface の型定義ファイルから CodeGear の入力の数の取得が不十分であるのが主な原因であった。

この問題は Perl スクリプトレベルで引数のチェックを十分に行う必要がある。すでに Interface のパーサーは実装している為、 パーサー経由で呼び出している API を持つ Interface の情報を取得する。パースした結果の情報に、 各 CodeGear の引数情報と引数の数を取得できれば、それらと API 呼び出し時に与えられている引数を比較すればチェックが可能である。現状は引数の数が揃っているかどうかで確認をしている。

Inteface の引数を確認し、Gearef マクロを生成している generate_stub.pl の箇所に、引数の確認処理を実装した。(ソースコード 5.24) ここで API 呼び出し時の引数は、`$tmpArgs`

に代入されている。CbC の関数呼び出しの引数はカンマで区切るので、2 行目でカンマで文字列を分割し、引数を配列@args に変換している。

generate_stub.pl はローカル変数のすべての型を記録しているので、6 行目で API 呼び出しをしているインスタンスの名前から Interface を特定する。特定後、ヘッダファイルの場所を取得し、8 行目で Interface のパーサーを呼び出している。パーサーから取得した情報から、メソッドの引数の数を 14 行目で取得し、 引数が格納されている配列@args の要素数と比較している。

ソースコード 5.24: Perl レベルでの引数チェック

```
1  #$tmpArgs =~ s/\(.*\)/\(\)/;
2  my @args = split(/,/,$tmpArgs);
3
4  #....
5
6  my $nextType           = $currentCodeGearInfo->{localVar}->{$next}  //
       $currentCodeGearInfo->{arg}->{$next};
7  my $nextTypePath       = $headerNameToInfo->{$nextType}->{path};
8  my $parsedNextTypePath = Gears::Interface->detailed_parse($nextTypePath);
9
10 unless (exists $parsedNextTypePath->{codeName}->{$method}) {
11   die "[ERROR] not found $next definition at $_ in $filename\n";
12 }
13 my $nextMethodInfo = $parsedNextTypePath->{codeName}->{$method};
14 my $nextMethodWantArgc = $nextMethodInfo->{argc};
15
16 if ($nextMethodWantArgc != scalar(@args)) {
17   die "[ERROR] invalid arg $line  you shoud impl $nextMethodInfo->{args}\
       n";
18 }
```

Perl スクリプトでエラーを検知すると、 エラーで終了する。ソースコード 5.25 の Interface の insertTest1 を呼び出す例題でエラーを発生さる。

ソースコード 5.25: StackTestInterface の定義

```
1  typedef struct StackTest <> {
2    __code insertTest1(Impl* stackTest, struct Stack* stack, __code next
       (...));
3    __code next(...);
4  } StackTest;
```

ソースコード 5.26 で API を呼び出しているが、 この呼び出し方法では stack が引数にない。

ソースコード 5.26: StackTestInterface の API 呼び出し (引数不足)

```
1  Stack* stack = createSingleLinkedStack(context);
2  StackTest* stackTest = createStackTestImpl3(context);
3  goto stackTest->insertTest1(shutdown);
```

GearsOS のビルドを行うと、ソースコード 5.27 のエラーが発生し、以降のビルドが停止する。Cmake はエラーを検知するとビルドを止めるように Makefile を作製するため、GearsOS の拡張構文のレベルで停止ができる。

ソースコード 5.27: Interface の API 呼び出し時の引数エラー

```
1 [ 12%] Generating c/examples/pop_and_push/main.c
2 [ERROR] invalid arg     goto stackTest->insertTest1(shutdown);
3   you shoud impl Impl* stackTest, struct Stack* stack, __code next(...)
4 make[3]: *** [CMakeFiles/pop_and_push.dir/build.make:81: c/examples/
      pop_and_push/main.c] Error 25
5 make[3]: *** Deleting file 'c/examples/pop_and_push/main.c'
```

generate_stub.pl 側で、出てきたローカル変数と型の組はすべて保存している。Interface 側の CodeGear の定義にも当然引数の型と名前は書かれている。このローカル変数の型と、CodeGear の定義の引数の型が、完全に一致しているかどうかのチェックを行うと、さらに強固な引数チェックが可能となる。ただし引数で渡す際に、例えば int 型の値の加算処理などを行っていると、その処理の結果が int 型になっているかどうかを Perl レベルでチェックする必要が出てしまう。

5.13　Interface の API にないものを呼び出した場合の検知

InterfaceAPI 呼び出し時に、そもそも Interface ファイルに定義していない API を呼び出してしまうことがある。これも CbC ファイルの変換前に処理を行いたい。

API 呼び出し時の処理は、ソースコード 5.24 の処理そのものであるため、この処理の中に未実装の API を検知する様にした。呼び出し元の Interface の情報パースした結果、ヘッダファイルに API の定義がなかった場合は 11 行目の `unless` に処理が落ち、 エラー終了する。

ソースコード 5.28 の例では、 Phils Interface に存在しない sleeping を呼び出している。この状態でビルドを実行すると、ソースコード 5.29 のエラーが Make 時に発生し、ビルドが停止する。

ソースコード 5.28: 存在しない sleeping の呼び出し

```
1 par goto phils0->sleeping(exit_code); // Not define
2 par goto phils1->thinking(exit_code);
```

ソースコード 5.29: 存在しない API の呼び出し時のエラー

```
1 [ 11%] Generating c/examples/DPP2/main.c
2 [ERROR] not found phils0 definition at     par goto phils0->sleeping(
      exit_code);
3  in examples/DPP2/main.cbc
```

```
4 | make[3]: *** [CMakeFiles/DPP2.dir/build.make:105: c/examples/DPP2/main.c
  |   ] Error 25
5 | make[3]: *** Deleting file 'c/examples/DPP2/main.c'
6 | make[2]: *** [CMakeFiles/Makefile2:442: CMakeFiles/DPP2.dir/all] Error 2
7 | make[1]: *** [CMakeFiles/Makefile2:450: CMakeFiles/DPP2.dir/rule] Error
  |   2
8 | make: *** [Makefile:293: DPP2] Error 2
```

5.14　Interface の API を完全に実装していない場合の検知

Interface の API で定義した CodeGear は、Impl 側はすべて実装している必要がある。しかし、 CodeGear の実装を忘れてしまうケースがある。これを Perl レベルで検知したい。

generate_stub.pl は 2 度 CbC ファイルを読み込む。書き出しに移る前に、変換しようとしている CbC ファイルの CodeGear の情報はすべて取得できている為に、ここで検知可能である。初回の CbC ファイルの読み込み終了時に、 検出できた CodeGear の名前と、CbC ファイルが実装しようとしている Interface の定義を見比べる。CodeGear をすべて満たしていなかった場合はエラーを出したい。

ソースコード 5.30 は、Interface が要求している API を実装したかを確認する部分である。変換しようとしている CbC ファイルが何かの Interface を実装しようとしている場合、 Interfaec の定義ファイルのパース結果から、満たすべき CodeGear の一覧を取得する。(ソースコード 1、2 行目) 実装していた場合は 6 行目でマークをつけ、 マークがなかった CodeGear が検知された時点でエラーを発生させる。(12 行目)

ソースコード 5.30: Interface の API 呼び出し時の引数エラー

```
1 |if ($implInterfaceInfo->{isImpl} && $filename =~ /\.cbc\z/) {
2 |  for my $shouldImplCode (map { $_->{name} } grep { $_->{args} =~ /Impl/
  |    } @{$implInterfaceInfo->{parsedInterfaceInfo}->{codes}}) {
3 |    my $isDefine = $shouldImplCode;
4 |    for my $implCode (keys %{$codeGearInfo}) {
5 |      if ($implCode =~  /$shouldImplCode/) {
6 |        $isDefine = 1;
7 |        next;
8 |      }
9 |    }
10|
11|    if ($isDefine ne 1) {
12|      die "[ERROR] Not define $isDefine at $filename\n";
13|    }
14|  }
15|}
```

ソースコード 5.31 の例では、Phils Interface の実装時に eating CodeGear の実装を忘れた際のエラーである。CMake がエラーを検知し、ビルドが停止するために、 GearsOS の

拡張構文レベルでのエラー検知が実現できている。

ソースコード 5.31: 未実装の Interface の API があることを知らせるエラー

```
[ 33%] Generating c/examples/DPP2/PhilsImpl.c
[ERROR] Not define eating at examples/DPP2/PhilsImpl.cbc
make[3]: *** [CMakeFiles/DPP2.dir/build.make:101: c/examples/DPP2/
    PhilsImpl.c] Error 25
make[2]: *** [CMakeFiles/Makefile2:442: CMakeFiles/DPP2.dir/all] Error 2
make[1]: *** [CMakeFiles/Makefile2:450: CMakeFiles/DPP2.dir/rule] Error 2
make: *** [Makefile:293: DPP2] Error 2
```

5.15　par goto の Interface 経由の呼び出しの対応

従来の par goto では Interface 経由の呼び出しは想定していなかった。par goto で継続したい CodeGear は Interface の API としてではなく、 Interface を入力として受け取る CodeGear として実装する必要があった。しかし食事する哲学者の問題 (Dining Philosophers Problem、 DPP) の検証などでは、特定の Interface が並列で動いている必要がある。DPP の例題の場合は、哲学者 (Philosopher) の Interface は並列で処理される必要がある。GearsOS で実装した DPP の例題で、par goto を実行している箇所をソースコード 5.32 に示す。

ソースコード 5.32: 5 つの PhilosopherInterface からの par goto

```
Phils* phils0 = createPhilsImpl(context,0,fork0,fork1);
Phils* phils1 = createPhilsImpl(context,1,fork1,fork2);
Phils* phils2 = createPhilsImpl(context,2,fork2,fork3);
Phils* phils3 = createPhilsImpl(context,3,fork3,fork4);
Phils* phils4 = createPhilsImpl(context,4,fork4,fork0);

par goto phils0->thinking(exit_code);
par goto phils1->thinking(exit_code);
par goto phils2->thinking(exit_code);
par goto phils3->thinking(exit_code);
par goto phils4->thinking(exit_code);

goto code2();
```

この記述は Perl トランスパイラによって、ソースコード 5.33 のメタ記述に変換される。

ソースコード 5.33: 5 つの PhilosopherInterface からの par goto の Perl 変換後

```
Phils* phils0 = createPhilsImpl(context,0,fork0,fork1);
Phils* phils1 = createPhilsImpl(context,1,fork1,fork2);
Phils* phils2 = createPhilsImpl(context,2,fork2,fork3);
Phils* phils3 = createPhilsImpl(context,3,fork3,fork4);
Phils* phils4 = createPhilsImpl(context,4,fork4,fork0);

```

```
7  struct Element* element;
8                  context->task = NEW(struct Context);
9                  initContext(context->task);
10                 context->task->next = phils0->thinking;
11                 context->task->idgCount = 0;
12                 context->task->idg = context->task->dataNum;
13                 context->task->maxIdg = context->task->idg + 0;
14                 context->task->odg = context->task->maxIdg;
15                 context->task->maxOdg = context->task->odg + 0;
16 GET_META(phils0)->wait = createSynchronizedQueue(context);
17                 element = &ALLOCATE(context, Element)->Element;
18                 element->data = (union Data*)context->task;
19                 element->next = context->taskList;
20                 context->taskList = element;
21                 context->task = NEW(struct Context);
22                 initContext(context->task);
23                 context->task->next = phils1->thinking;
24
25 ...
```

PhilsInterface は大本の context で作製しているため、par goto で作製した Context に Interface の情報が保存されていなかった。処理を実行すると Interface の値を StubCodeGear で取り出す際に、初期化をしていない値をとってしまい、 セグメンテーション違反が発生する。この問題は、Gearef マクロを利用して作製した par goto 用の Context の引数用の保存場所に、それぞれ実装のポインタを書き込むことで解決する。

ソースコード 5.34 では、par goto 作製した Context である context>task に、 Gearef マクロを用いて引数として Phils のインスタンスを代入している。(ソースコード 3、 15 行目) また thinking の引数は CodeGear が必要であったので、これも next に設定している。(ソースコード 4、 16 行目) この処理は、generate_stub.pl で、 par goto 時に Interface 呼び出しをしている際のパターンを新たに実装し、Interface のパースの結果から得られた引数に書き込む様なルーチンで実現している。

ソースコード 5.34: 改善された Interface 経由での par goto

```
1                  context->task->maxOdg = context->task->odg + 0;
2  GET_META(phils0)->wait = createSynchronizedQueue(context);
3  Gearef(context->task, Phils)->phils = (union Data*) phils0;
4  Gearef(context->task, Phils)->next = C_exit_code;
5                  element = &ALLOCATE(context, Element)->Element;
6                  element->data = (union Data*)context->task;
7                  element->next = context->taskList;
8                  context->taskList = element;
9                  context->task = NEW(struct Context);
10                 initContext(context->task);
11                 context->task->next = phils1->thinking;
12 ...
13                 context->task->maxOdg = context->task->odg + 0;
14 GET_META(phils1)->wait = createSynchronizedQueue(context);
15 Gearef(context->task, Phils)->phils = (union Data*) phils1;
```

```
16 Gearef(context->task, Phils)->next = C_exit_code;
17                      element = &ALLOCATE(context, Element)->Element;
```

第6章　トランスパイラによるメタ計算

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。すべてを純粋な CbC で記述すると記述量が膨大になる。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でる。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。

6.1　トランスパイラ

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、 トランスパイラ、トランスコンパイラ、トランスラなどと呼ぶ。本論文では以下トランスパイラと呼ぶ。トランスパイラとしては JavaScript を古い規格の JavaScript に変換する Babel[28] がある。

またトランスパイラは､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 にノーマルレベル、メタレベルの書き別けの機能などを追加した拡張言語であると言える。コンパイル時に CMake によって呼び出される 2 種類の Perl スクリプトで等価な純粋な CbC に変換される。これらの Perl スクリプトは GearsOS の CbC から純粋な CbC へと変換している為に一種のトランスパイラと言える。

6.2　トランスパイラによるメタレベルのコード生成

トランスパイラはノーマルレベルで記述された GearsOS を、メタレベルを含む CbC へと変換する役割である。変換時に様々なメタ情報を CbC のファイルに書き出すことが可能である。従来は Stub の生成や、 引数の変更などを行っていたが、 さらにメタレベルのコードをトランスパイラで作製したい。トランスパイラ上でメタレベルのコードを作製することによって、 GearsOS 上でのアプリケーションの記述が容易になり、かつメタレベルのコードを柔軟に扱うことができる。本研究では様々なメタレベルのコードを、トランスパイラで生成することを検討した。

6.3　トランスパイラ用の Perl ライブラリ作製

従来の Perl トランスパイラは generate_stub.pl と generate_context.pl の 2 種類のスクリプトで構築されていた。これらのスクリプトはそれぞれ独立した処理を行っていた。

しかし本研究を進めるにつれて、Interface のパーサーやメタ計算部分の操作を行う API など、Perl スクリプトで共通した実装が見られた。さらに generate_stub.pl ら Perl スクリプトの行数や処理の複雑度が上がり、適切に処理をモジュール化する必要が生じた。この為新しく実装した Perl トランスパイラが利用する API は、 Perl のモジュール機能を利用しモジュールの形で実装した。以下に実装したモジュールファイルと、その概要を示す。

- `Gears::Context`
  - context.h の自動生成時に呼び出されるモジュール
  - 変換後の CbC のコードを解析し、使用されている DataGear の数え上げを行う
- `Gears::Interface`
  - Interface および Implement のパーサー
- `Gears::Template` Gears::Template 以下は Perl スクリプトが生成する際に、テンプレートとして呼び出すファイルの定義などがある
  - `Gears::Template::Context`
    - context.h のテンプレート
  - `Gears::Template::Context::XV6`
    - CbC Xv6 専用の context.h のテンプレート
  - `Gears::Template::Gmain`

　　∗ GearsOS Main 関数のテンプレート

- `Gears::Stub`
  - Stub Code Gear 生成時に呼び出されるモジュール

これらは generate_stub.pl および generate_context.pl および、本研究で作製した Perl のツールセットからも呼び出される。

6.4　context.h の自動生成

GearsOS の Context の定義は context.h にある。Context は GearsOS の計算で使用されるすべての CodeGear、DataGear の情報を持っている。context.h では DataGear に対応する `union Data` 型の定義も行っている。Data 型は C の共用体であり、 Data を構成する要素として各 DataGear がある。各 DataGear は構造体の形で表現されている。各 DataGear 自体の定義も context.h の union Data の定義の中で行われている。

DataGear の定義は Interface ファイルで行っていた。Interface ファイルは GearsOS 用に拡張されたシンタックスのヘッダファイルを使っており、 直接 CbC からロードすることができない。その為従来はプログラマが静的に Interface ファイルを CbC の文脈に変換し、 context.h に構造体に変換したものを書いていた。この手法では手書きでの構築のために自由度は高かったが、 GearsOS の例題によっては使わない DataGear も、 context.h から削除しない限り context に含んでしまう問題があった。さらに Interface ファイルで定義した型を context.h に転記し、それをもとに Impl の型を考えて CbC ファイルを作製する必要があった。これらをすべてユーザーが行うと、ファイルごとに微妙な差異が発生したりとかなり煩雑な実装を要求されてしまう。DataGear の定義は Interface ファイルを作製した段階で決まり、 使用している DataGear、CodeGear はコンパイル時に確定する。使用している各 Gear がコンパイル時に確定するならば、 コンパイルの直前に実行される Perl トランスパイラでも Gear の確定ができるはずである。ここから context.h をコンパイルタイミングで Perl スクリプト経由で生成する手法を考案した。

6.4.1　ビルド時の context.h の生成タイミング

context.h をビルドの途中で生成するには、CMake が context.h を作製するようにプログラミングする必要がある。しかし CMake の文法はきわめて複雑であるので、現状の GearsOS の CMakeLists.txt の定義を変更したくない。この為には現在ビルド時に動作する generate_stub.pl か generate_context.pl のいずれかで生成を行いたい。

GearsCbC からメタ計算を含む CbC ファイルに変換する generate_stub.pl は各 CbC ファイルを 1 つ 1 つ呼び出していた。context.h を生成しようとする場合、 プロジェクトで利用する全 CbC ファイルを扱う必要がある。従って generate_stub.pl では context.h の作製はできない。

Context の初期化ルーチンを作製する generate_context.pl は、その特性上すべての CbC ファイルをロードしていた。したがって context.h を作製する場合はこのスクリプトで行うと現状の CMake に手をつけずに変更ができる。context.h を作製するため、generate_context.pl の処理は図 6.1 の処理を実行することになる。

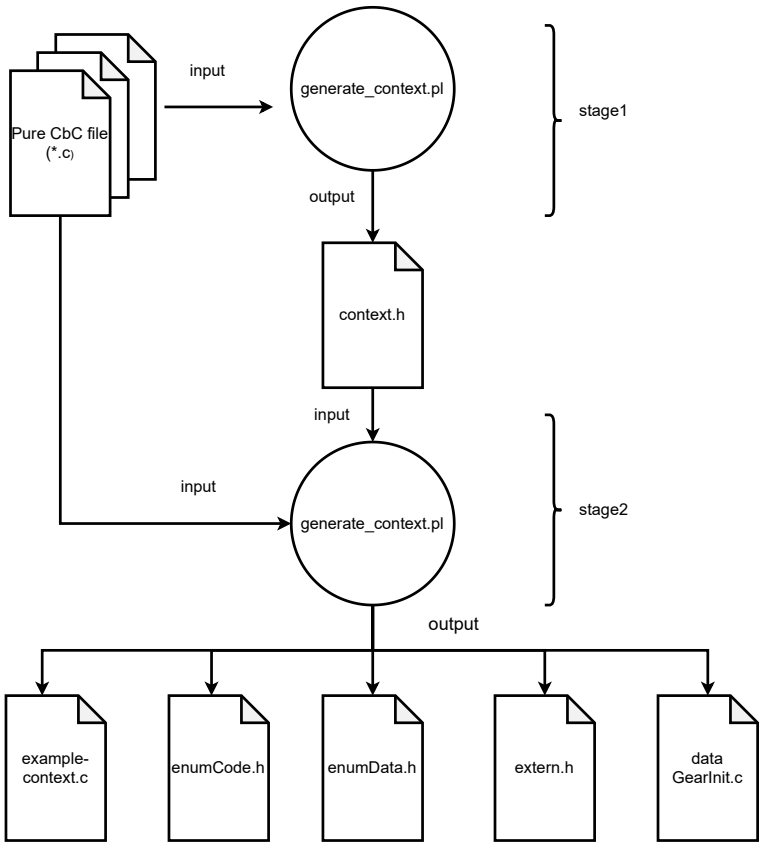

図 6.1: generate_context.pl を使った context.h とファイル生成

6.4.2　context.h の生成処理

generate_context.pl で context.h を作製したい。この為には、プロジェクトで使用している DataGear を特定する必要がある。generate_context.pl の時点で CbC ファイルはメタ

情報を含む表現に変換されている。Interface の使用と、実装を示す`#interface` や`#impl` 構文はこの時点では落とされている為、別の情報から使用している DataGrar を取得する必要がある。取得から context.h の作製までの流れを図 6.2 に示す。

DataGear は GearsOS のメタレベルでは構造体で表現されていた。また、DataGear は各 CodeGear の引数の形で受け渡されている。ここから DataGear を取得するには、すべての CodeGear の引数をチェックし、構造体の名前を取得すればいい。すべての CbC ファイルを調査後、 取得した構造体の名前に対応するヘッダファイルがあるかどうかを調査する。この調査は Gears::Context モジュールで行う。

ヘッダファイルがあった場合、 Gears::Interface の API を利用して、 Interface もしくは Impl のファイルとして利用可能であるかを確認する。Interface、Impl ファイルでなかった場合は、ただの構造体であるので context.h に含む情報からは除外する。収集された DataGear はソースコード 6.1 に示す Perl の連想配列に変換される。連想配列は、最初のキーが Interface の名前であり、Implement がある場合は Interface に対応する値の value に、impl という連想配列が入れ子で入っている。この中に Interface を実装している Impl の型情報が記録されている。

ソースコード 6.1: context.h に出力する DataGear の集合

```
1   },
2   Phils         {
3       elem   {
4           content            [
5               [0] "enum Code putdown_lfork;",
6               [1] "union Data* phils;",
7               [2] "enum Code putdown_rfork;",
8               [3] "enum Code thinking;",
9               [4] "enum Code pickup_rfork;",
10              [5] "enum Code pickup_lfork;",
11              [6] "enum Code eating;",
12              [7] "enum Code next;"
13          ],
14          file_name          "/home/anatofuz/src/firefly/hg/Gears/Gears/
    src/parallel_execution/../parallel_execution/examples/DPP2/Phils.h",
15          inner_code_gears   {
16              next   1
17          },
18          name               "Phils"
19      },
20      impl   {
21          PhilsImpl   {
22              content            [
23                  [0] "int self;",
24                  [1] "struct AtomicT_int* Leftfork;",
25                  [2] "struct AtomicT_int* Rightfork;",
26                  [3] "enum Code next;"
27              ],
28              file_name          "/home/anatofuz/src/firefly/hg/Gears/
```

```
    Gears/src/parallel_execution/../parallel_execution/examples/DPP2/
    PhilsImpl.h",
29                inner_code_gears   {},
30                isa                "Phils",
31                name               "PhilsImpl"
32            }
33        }
34    },
35    Queue         {
36        elem   {
37            content            [
38                [0] "enum Code whenEmpty;",
```

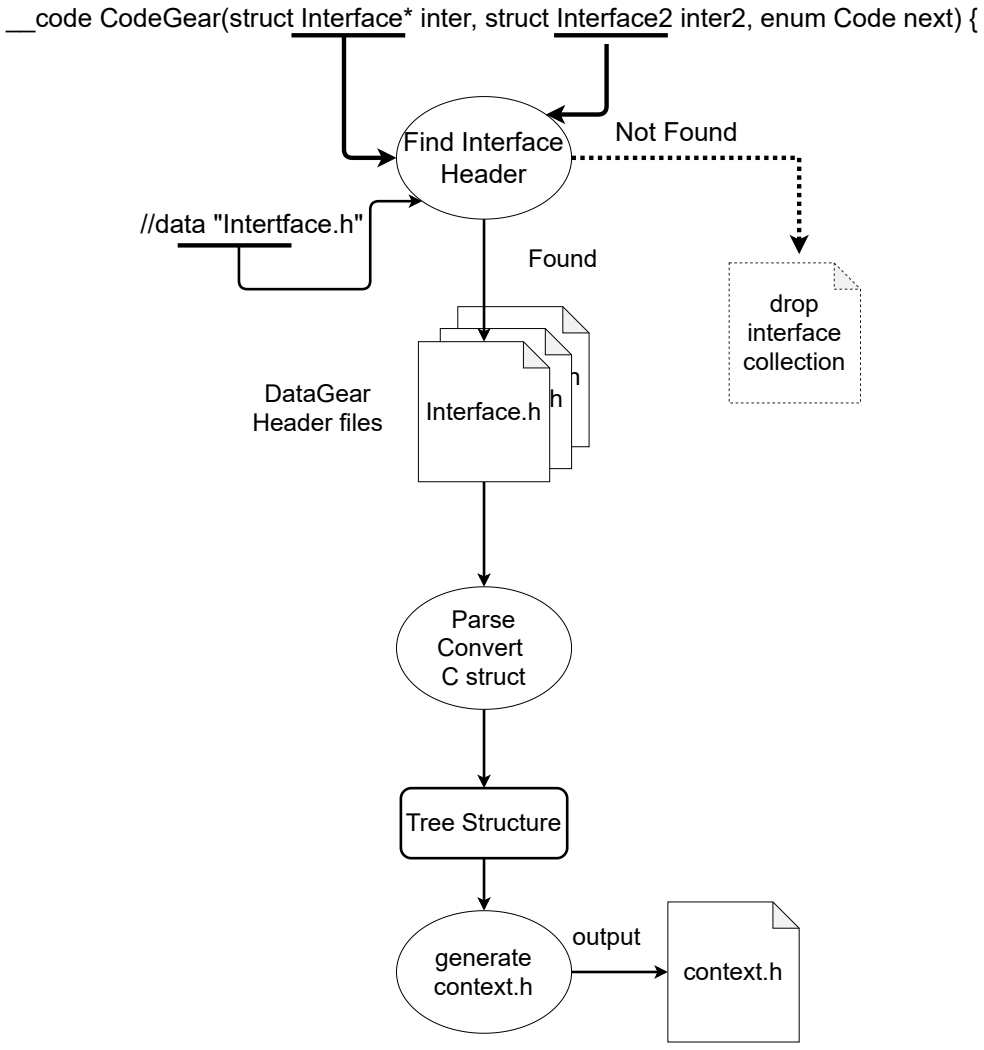

図 6.2: DataGear の収集と context.h 生成の処理イメージ

作製された context.h の union Data の定義をソースコード 6.2 に示す。context.h の union Data は、アルファベット順でソートされ、 Interface、Impl の順で記述される。ここにはデバッグ用に変換した定義ファイルのパスがコメントで埋め込まれる。また、多重 include 防止用のマクロも生成される。

ソースコード 6.2: 生成された context.h の union Data 定義

```
union Data {
    ///home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
   /../parallel_execution/Atomic.h
#ifndef ATOMIC_STRUCT
    struct Atomic {
        union Data* atomic;
...
#else
    struct Node;
#endif
    ///home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
   /../parallel_execution/examples/DPPMC/Phils.h
#ifndef PHILS_STRUCT
    struct Phils {
        union Data* phils;
        enum Code putdown_lfork;
        enum Code putdown_rfork;
        enum Code thinking;
        enum Code pickup_rfork;
        enum Code pickup_lfork;
        enum Code eating;
        enum Code next;
    } Phils;
#define PHILS_STRUCT
#else
    struct Phils;
#endif
    ///home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
   /../parallel_execution/examples/DPPMC/PhilsImpl.h
#ifndef PHILSIMPL_STRUCT
    struct PhilsImpl {
        int self;
        struct AtomicT_int* Leftfork;
        struct AtomicT_int* Rightfork;
        enum Code next;
    } PhilsImpl;
#define PHILSIMPL_STRUCT
#else
    struct PhilsImpl;
#endif
...
}
```

6.4.3　Interface 定義の include ファイルの解決

DataGear の定義を自動で context.h に生成することが可能となった。ここで DataGear が別のヘッダファイルに定義している構造体などをフィールドで持つケースを扱う。別のヘッダファイルで定義している構造体を使う場合、context.h の中で DataGear に対応する構造体の定義をする前までに、include する必要が発生する。

ヘッダファイルとして state_db.h を include する宣言が書かれた Worker Intertface の実装の mcWorker(ソースコード 6.3) がある。generate_context.pl では、DataGear の定義ファイルを Gears::Interface のパース API でパースし、情報を取得していた。

ソースコード 6.3: mcWorker Impl の定義

```
1  #include "state_db.h"
2  #include "memory.h"
3  #include "TaskIterator.h"
4
5  typedef struct MCWorker <> {
6    pthread_mutex_t mutex;
7    pthread_cond_t cond;
8  ...
9    StateDB parent;
10   StateDB root;
11 ...
12 } MCWorker;
```

パースした結果の情報含まれる include する必要があるヘッダファイルの一覧を取得し、context.h を生成するタイミングで、 struct Context の定義の前に include するコードを挿入する。(ソースコード 6.4) include の結果では、 3、5、7 行目にそれぞれ include するマクロが生成されている。各行の上の行には、 include の記述があった DataGear のファイルパスが記載される。

ソースコード 6.4: context.h 内での include

```
1  #include "c/enumData.h"
2  // use /home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
       /../parallel_execution/ModelChecking/MCWorker.h
3  #include "/home/anatofuz/src/firefly/hg/Gears/Gears/src/
       parallel_execution/../parallel_execution/ModelChecking/memory.h"
4  // use /home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
       /../parallel_execution/plautogen/impl/SingleLinkedQueue.h
5  #include "/home/anatofuz/src/firefly/hg/Gears/Gears/src/
       parallel_execution/../parallel_execution/plautogen/impl/../../
       ModelChecking/TaskIterator.h"
6  // use /home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
       /../parallel_execution/ModelChecking/MCWorker.h
7  #include "/home/anatofuz/src/firefly/hg/Gears/Gears/src/
       parallel_execution/../parallel_execution/ModelChecking/TaskIterator.h"
8  // use /home/anatofuz/src/firefly/hg/Gears/Gears/src/parallel_execution
       /../parallel_execution/ModelChecking/MCWorker.h
```

```
9  #include "/home/anatofuz/src/firefly/hg/Gears/Gears/src/
       parallel_execution/../parallel_execution/ModelChecking/state_db.h"
10 struct Context {
11     enum Code next;
12     struct Worker* worker;
13 ....
```

6.4.4　context.h のテンプレートファイル

Perl のモジュールとして `Gears::Template::Context` を作製した。xv6 プロジェクトの場合は一部ヘッダファイルに含める情報が異なるため、 xv6 のビルド用にサブモジュールとして `Gears::Template::Context::XV6` も実装した。これらのテンプレートモジュールは generate_context.pl の実行時のオプションで選択可能である。

呼び出しには Perl の動的モジュールロード機能を利用している。各モジュールに共通の API を記述しており、 プロジェクトごと使うテンプレートに限らず共通して呼び出すことが可能である。

6.5　meta.pm による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、基本は Context からの DataGear の取り出しを行う。これ以外のことを行う場合は、 DataGear の取り出しを含めてメタ計算を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。これを避けるには、Stub 以外の Meta Code Gear をユーザーが自由に定義でき、任意にその MetaCodeGear に継続できる必要がある。従来の GearsOS では、継続先の MetaCodeGear は `__code meta` に決め打ちとなっているために、 自在に変更する API が必要となる。

6.5.1　__ncode による自由な MetaCodeGear の定義

ユーザーが自由に MetaCodeGear を定義できる API として__ ncode を定義した。これは CbC のマクロで__code になるように制御されている。

generate_stub.pl、generate_context.pl の両方の Perl トランスパイラは、 __code で定義されている CodeGer はノーマルレベルの CodeGear だと解釈する。ノーマルレベルの CodeGear は Stub の生成や、メタレベルの情報を含んだものに変換される。対して__ncode は、MetaCodeGear であると判断されるので、これらの変換が行われない。ユーザーはメタレベルの計算をすべて実装する必要はあるものの、__ncode を使うと自由に MetaCodeGear を定義できる。ソースコード 6.5 は__ncode によって定義した、 モデル検査用の MetaCodeGear である。

ソースコード 6.5: __ncode によって定義された MetaCodeGear

```
1  __ncode mcMeta(struct Context* context, enum Code next) {
2      context->next = next; // remember next Code Gear
3      struct MCWorker* mcWorker =  (struct MCWorker*) context->worker->
       worker;
4      StateNode st ;
5      StateDB out = &st;
6      struct Element* list = NULL;
7      struct MCTaskManagerImpl* mcti = (struct MCTaskManagerImpl *)mcWorker
       ->taskManager->taskManager;
8      out->memory = mcti->mem;
9      out->hash = get_memory_hash(mcti->mem,0);
10 ...
11     goto meta(context, context->next);
12 }
```

6.5.2　meta.pm

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。これはいわゆるリフレクション処理に該当する。

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 6.6 に、 この使用例での CodeGear の継続の様子を図 6.3 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 6.6 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 6.6: 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;
```

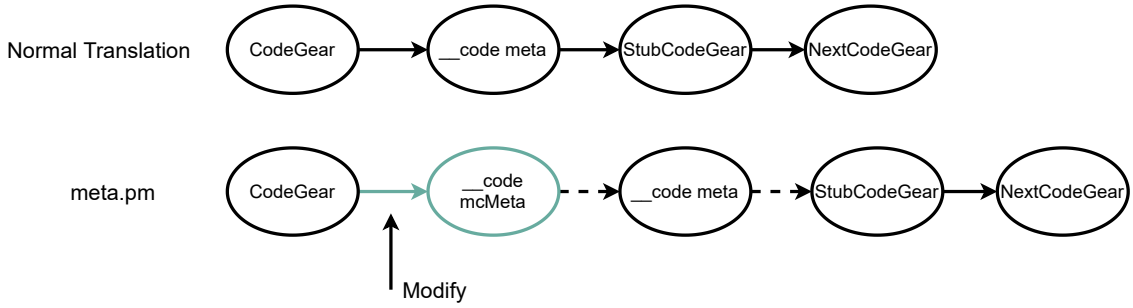

図 6.3: meta.pm を使った継続先の MetaCodeGear の切り替え

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。これらの処理は Perl のクロージャの形で表現しており、 トランスパイラ側では共通の API で呼び出すことが可能である。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。ソースコード 6.7 に、meta.pm の設定を書かなかった場合の変換結果を示す。ソースコード 6.8 では、meta.pm の設定を置いた場合の変換結果である。継続先が meta からモデル検査用の MetaCodeGear である mcMeta に切り替わっていることが解る。

ソースコード 6.7: 通常の thinkingPhilsImpl のメタレベルのコード

```
1 __code thinkingPhilsImpl(struct Context *context,struct PhilsImpl* phils,
      struct Fork* fork, enum Code next) {
2     printf("%d: thinking\n", phils->self);
3     goto meta(context, C_pickup_lforkPhilsImpl);
4 }
```

ソースコード 6.8: meta.pm によって mcMeta へと継続が切り替わった thinkingPhilsImpl

```
1 __code thinkingPhilsImpl(struct Context *context,struct PhilsImpl* phils,
      struct Fork* fork, enum Code next) {
2     printf("%d: thinking\n", phils->self);
3     goto mcMeta(context, C_pickup_lforkPhilsImpl);
4 }
```

6.6　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 6.9 に示す。

ソースコード 6.9: 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" as "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 6.10 に示す。

ソースコード 6.10: 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 6.11 に示す。

ソースコード 6.11: 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 で作製している。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。pop2 の API を呼び出すと、 Stack Interface 中の data に Stack に保存されていたデータのアドレスが書き込まれる。

当初 Perl スクリプトが生成した pop2Test1 の stub CodeGear はソースコード 6.12 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 6.4 に示す。

ソースコード 6.12: 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。pop2 で取り出したデータは、 上記で確認した通り 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 6.12 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。これは Interface の Impl の CodeGear は、Interface から値を取得するという GearsOS のルールの為である。現状では pop2 でせっかく取り出した値を

StubCodeGear で取得できない。

ここで必要となってくるのは、 実装している Interface 以外の呼び出し元の Interface からの値の取得である。今回の例では StackTest Interface ではなく Stack Interface から data、 data1 を取得したい。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 のトランスパイラで Stub Code を生成したい。

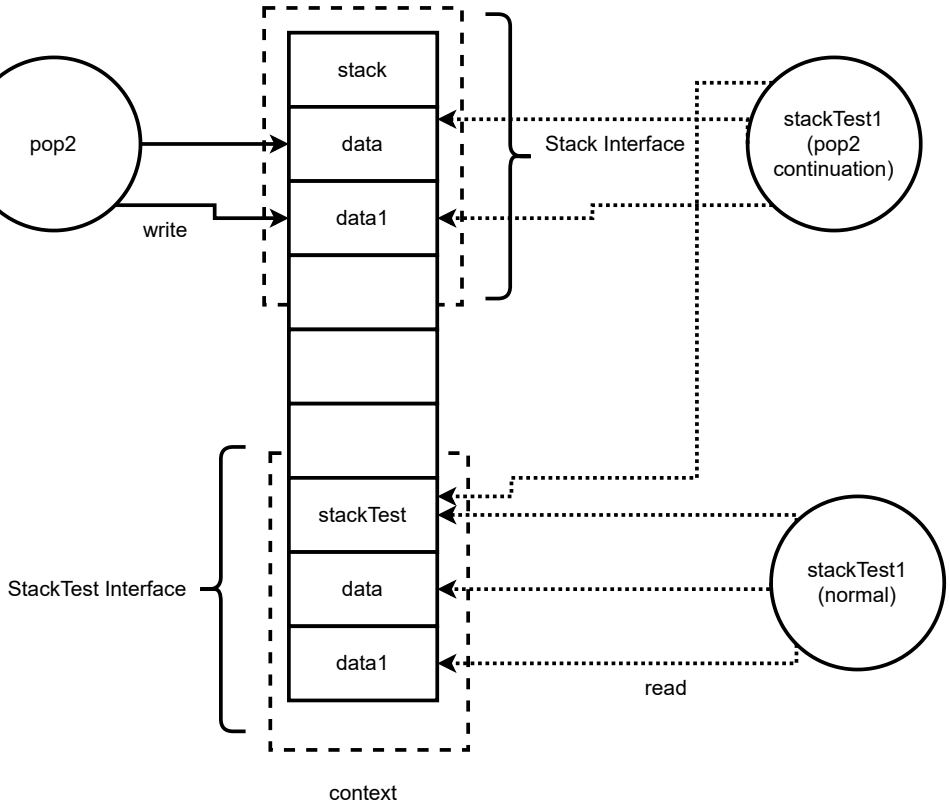

図 6.4: stackTest1 の stub の概要

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 トランスパイラが生成している Stub を書き換える方法も取れる。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、 Stub を別に別ける必要がある。

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 のトランスパイラである `generate_stub.pl` に導入した。

6.7　別 Interface からの書き出しを取得する Stub の生成

別 Interface からの書き出しを取得する場合、 generate_stub.pl では次の点をサポートする機能をいれれば実現可能である。

- goto 先の CodeGear が出力を持つ Interface でかつ継続で渡している CodeGear が別 Interface の場合の検知
  - この場合は goto している箇所で渡している継続の enum を、新たに作製した stub の enum に差し替える
- 継続で実行された場合に別に Interface から値をとってこないといけない CodeGear 自身
  - Stub を別の Interface から値をとる実装のものを別に作製する

`generate_stub.pl` 内では変換対象の CbC のソースコードを 2 度読み込む。最初の読み込み時に継続の状況を確認し、 2 度目の読み込み時に状況を踏まえてコードを生成すれば良い。初回の読み込み時に Interface 経由の `goto` 文があった場合に、別 Interface からの出力があるかなどの情報を確認したい。

6.7.1　初回 CbC ファイル読み込み時の処理

Interface 経由での goto 文は `goto interface->method()` の形式で呼び出される。ソースコード 6.13 はこの形式で来ていた行を読み込んだタイミングで実行される処理である。

ソースコード 6.13: goto 時に使用する interface の解析

```
1  } elsif (/^(.*)goto (\w+)\-\>(\w+)\((.*)\);/) {
2    debug_print("getDataGear",__LINE__, $_) if $opt_debug;
3    # handling goto statement
4    # determine the interface you are using, and in the case of a goto
     CodeGear with output, create a special stub flag
5    my $prev = $1;
6    my $instance = $2;
7    my $method = $3;
8    my $tmpArgs = $4;
9    my $typeName = $codeGearInfo->{$currentCodeGear}->{arg}->{$instance};
10   my $nextOutPutArgs = findExistsOutputDataGear($typeName, $method);
11   my $outputStubElem = { modifyEnumCode => $currentCodeGear,
     createStubName => $tmpArgs };
12
13   if ($nextOutPutArgs) {
14     my $tmpArgHash = {};
15     for my $vname (@$nextOutPutArgs) {
16       $tmpArgHash->{$vname} = $typeName;
17     }
18
19     $outputStubElem->{args} = $tmpArgHash;
20
21     #We're assuming that $tmpArgs only contains the name of the next
     CodeGear.
22     #Eventually we need to parse the contents of the argument. (eg.
     @parsedArgs)
23     my @parsedArgs = split /,/ , $tmpArgs; #
24
25     $generateHaveOutputStub->{counter}->{$tmpArgs}++;
26     $outputStubElem->{counter} = $generateHaveOutputStub->{counter}->{
     $tmpArgs};
27     $generateHaveOutputStub->{list}->{$currentCodeGear} =
     $outputStubElem;
28   }
```

1 行目の正規表現は Interface 経由での goto 文の正規表現パターンである。変数$instance は Interface のインスタンスである。正規表現パターンでは interface->method の->の前に来ている変数名に紐づけられる。変数$method は goto 先の Interface の API である。正規表現パターンでは interface->method の->の後に来ている API 名である。ソースコード 6.9 の pop2Test では、 stack->pop2 の呼び出しをしているため、 stack がインスタンスであり、 pop2 が API である。現在解析している goto 文が含まれている CodeGear の名前は、変数$currentCodeGear で別途保存している。連想配列である$codeGearInfo の中には、 各 CodeGear で使われている変数と変数の型などの情報が格納されている。ソースコード 6.13 の 9 行目では、 $codeGearInfo 経由で Interface のインスタンスから、具体的にどの型が呼ばれているかを取得する。pop2Test では、 インスタンス stack に対応する型名は Stack と解析される。

ソースコード 6.13 の 10 行目で実行されている findExistsOutputDataGear は generate_stub.pl 内の関数である。これは Interface の名前とメソッド名を与えると、 Interface の定義ファイルのパース結果から出力の有無を確認する動きをする。出力がある場合は出力している変数名の一覧を返す。ソースコード 6.9 の例では pop2 は data と data1 を出力している為、 これらがリストとして関数から返される。出力がない場合は偽値を返すために 13 行目からの if 文から先は動かない。出力があった場合は generate_stub.pl の内部変数に出力する変数名と、 Interface の名前の登録を行う。生成する Stub は命名規則は、Stub の本来の CodeGear の名前の末尾に_に続けて数値をいれる。__code CodeGearStub の場合は、 __code CodeGearStub_1 となる。変換する CodeGear は、別の CodeGear の API 呼び出しによって別に変換される可能性がある。この変換を切り分けたいため、一意の ID として数値を入れている。

27 行目で$generateHaveOutputStub の list 要素に現在の CodeGear の名前と、 出力に関する情報を代入している。現在の CodeGear の名前を保存しているのは、この後のコード生成部分で enum の番号を切り替える必要があるためである。ソースコード 6.9 の例では pop2Test が使う enum を書き換える必要がある為、 ここの$currentCodeGear は pop2Test となる。ここで作製した$outputStubElem は、返還後の CbC コードを生成しているフェーズで呼びされる。

6.7.2　enum の差し替え処理

ソースコード 6.14 の箇所は遷移先の enum を Perl スクリプトで生成し、 GearsOS が実行中に enum を context に書き込むコードを生成するフェーズである。

ソースコード 6.14: Gearef のコード生成部分

```
1 if ($outputStubElem && !$stub{$outputStubElem->{createStubName}."_stub
    "}->{static}) {
2   my $pick_next = "$outputStubElem->{createStubName}_$outputStubElem->{
    counter}";
3   $return_line .= "${indent}Gearef(${context_name}, $ntype)->$pName =
    C_$pick_next;\n";
4   $i++;
5   next;
6 }
```

if 文で条件判定をしているが、前者は出力があるケースかどうかのチェックである。続く条件式は GearsOS のビルドルールとして静的に書いた stub の場合は変更を加えない為に、静的に書いているかどうかの確認をしている。変数$pick_next で継続先の CodeGear の名前を作製している。CodeGear の名前は一度目の解析で確認した継続先に_の後ろに ID をつけたものを結合している。ここで作製した CodeGear の名前を、3 行目で context に書き込む CbC コードとして生成している。

実際に生成された例題をソースコード 6.15 に示す。

ソースコード 6.15: enum の番号が差し替えられた CodeGear

```
1 __code pop2TestStackTestImpl3(struct Context *context,struct
      StackTestImpl3* stackTest, struct Stack* stack, enum Code next) {
2     Gearef(context, Stack)->stack = (union Data*) stack;
3     Gearef(context, Stack)->next = C_pop2Test1StackTestImpl3_1;
4     goto meta(context, stack->pop2);
5 }
```

6.7.3　対応する Stub Code Gear の作製

enum の番号に対応する Stub CodeGear を次は作製する必要がある。StubCodeGear はすでに作製されているオリジナルの StubCodeGear の中身のうち、 OutputDataGear の取得をしている箇所を、別 Interface から取得するように変更する必要がある。Perl スクリプトの文字列置換を使ってこの方法を実装する。

generate_stub.pl は、スクリプトで StubCodeGear の中身を文字列として作製し、CbC ファイルに書き出していた。まず、 StubCodeGear を複製する必要があるため、 中身の文字列を保存するように修正した。CbC ファイルのすべての行を読み込み、ファイルの変換が終了したタイミングで、 enum の差し替えが行われていた場合に差し替えようの StubCodeGear の作製を行う。実際に行っている箇所を、ソースコード 6.16 に示す。

この処理では、新たに作成するべき StubCodeGear の名前を変数`$createStubName` に 5 行目で代入している。StubCodeGear の名前は、1 度目の読み込み時に作製されたものである。変換対象の変数と、取得するべき Interface の組は、連想配列`$replaceArgs` に設定されている。StubCodeGear の中身は`$replaceStubContent` に 7 行目で代入し、これを`$replaceArgs` の変数と Interface の組の分だけ置換する。

ソースコード 6.16: StubCodeGear の生成箇所

```
1  #Create a stub when the output is a different interface
2  for my $modifyEnumCodeCodeGear (keys %{$generateHaveOutputStub->{list}})
       {
3    my $outputStubElem      = $generateHaveOutputStub->{list}->{
       $modifyEnumCodeCodeGear};
4    my $targetStubName      = $outputStubElem->{createStubName};
5    my $createStubName      = "$outputStubElem->{createStubName}
       _$outputStubElem->{counter}";
6    my $replaceArgs         = $outputStubElem->{args};
7    my $replaceStubContents = $dataGearName{$targetStubName};
8
9    #If the stub was handwritten, skip
10   if ($stub{"${targetStubName}_stub"}->{static}) {
11     next;
12   }
```

```
13
14    for my $arg (keys %$replaceArgs) {
15      my $interface = $replaceArgs->{$arg};
16      $replaceStubContents =~ s/,(.*)\)->$arg/, $interface)->$arg/;
17    }
18
19    generateStub($fd,$createStubName,$replaceStubContents);
20  }
```

ソースコード 6.17 で pop2Test1StackTestImpl3 の本体と、StubCodeGear を確認する。pop2...Impl3_stub は StackTest Interface からすべての値を取得している。対して pop2...Impl3_1_stub は、Stack Interface の継続で渡される為に、値を Stack から取り出す必要がある。見ると data、dta1 の値を、Gearef マクロを通して Stack Interface から取得するように変更されている。どちらの Stub も継続先は pop2Test1StackTestImpl3 であり、引数で渡す値の型と個数は揃っているために、実装側の変更をする必要がない。これによって柔軟なメタ計算の生成が可能となった。

ソースコード 6.17: 生成された StubCodeGear と、もとの CodeGear

```
1  __code pop2Test1StackTestImpl3(struct Context *context,struct
       StackTestImpl3* stackTest, union Data* data, union Data* data1, struct
        Stack* stack, enum Code next) {
2      String* str = (String*)data;
3      String* str2 = (String*)data1;
4
5      printf("%d\n", str->size);
6      printf("%d\n", str2->size);
7      goto meta(context, next);
8  }
9
10
11 __code pop2Test1StackTestImpl3_stub(struct Context* context) {
12   StackTestImpl3* stackTest = (StackTestImpl3*)GearImpl(context,
       StackTest, stackTest);
13   Data* data = Gearef(context, StackTest)->data;
14   Data* data1 = Gearef(context, StackTest)->data1;
15   Stack* stack = Gearef(context, StackTest)->stack;
16   enum Code next = Gearef(context, StackTest)->next;
17   goto pop2Test1StackTestImpl3(context, stackTest, data, data1, stack,
       next);
18 }
19
20
21 __code pop2Test1StackTestImpl3_1_stub(struct Context* context) {
22   StackTestImpl3* stackTest = (StackTestImpl3*)GearImpl(context,
       StackTest, stackTest);
23   Data* data = Gearef(context, Stack)->data;
24   Data* data1 = Gearef(context, Stack)->data1;
25   Stack* stack = Gearef(context, StackTest)->stack;
26   enum Code next = Gearef(context, StackTest)->next;
```

```
27   goto pop2Test1StackTestImpl3(context, stackTest, data, data1, stack,
      next);
28 }
```

6.8　ジェネリクスのサポート

型の安全性を保ったまま、柔軟な関数の定義を可能とする機能にジェネリクスがある。ジェネリクスは、型自体を変数 (型変数) と設定し、あらゆる型でも同様のふるまいを行うという機能である。

GearsOS では Interface 宣言に使われる Type、Impl などの型キーワードは、そもそもジェネリクスを意識して作られていた。このジェネリクスを GearsOS にサポートしたい。

ジェネリクスは型変数を利用して関数、クラスを宣言する部分と、型変数に具体的な値を代入して操作する部分に意味が分離できる。GearsOS では関数、クラス宣言の部分は、Interface の宣言および Implmenet の実装に該当する。型変数に具体的な値をいれる部分は、 Interface を利用する場所、もしくは DataGear として使う場所に相当する。

6.8.1　ジェネリクスを使った Interface の定義

ジェネリクスを使って Interface を定義した例を、ソースコード 6.18 に示す。GearsOS でジェネリクスを扱う場合は、型名の宣言に続く<>の部分に、型変数を記述する。この例では、型変数として T を使用している。型変数 T がどのような値を使うかは、ジェネリクスを呼び出している箇所で確定する。

ソースコード 6.18: ジェネリクスを使った AtomicT の定義

```
1 typedef struct AtomicT <T>{
2     __code checkAndSet(Impl* atomicT,T oldData, T newData, __code next
     (...), __code fail(...));
3     __code set(Impl* atomicT,T newData, __code next(...));
4     __code next(...);
5     __code fail(...);
6 } AtomicT;
```

6.8.2　ジェネリクスを使った Impl の定義

Implement の定義でも、ジェネリクスは同様に使うことが可能である。AtomicT の実装である AtomicTImpl の定義を、ソースコード 6.19 に示す。Interface でジェネリクスを使った場合、Impl 側でもジェネリクスが伝搬される。この場合は型変数が、Interface と同様の T であるので、Interface で決定された T の型と同様の型に決まる。

ソースコード 6.19: ジェネリクスを使った AtomicT の実装の定義

```
typedef struct AtomicTImpl <T> impl AtomicT {
  T atomic;
  T init;
  __code next(...);
} AtomicTImpl;
```

6.8.3　ジェネリクスを使った CodeGear の記述

Impl を定義したので対応する CbC ファイルを確認する。ソースコード 6.20 は、ジェネリクスを使った CodeGear の実装の一部である。型変数を使った CodeGear の記述では、型変数は通常の型のように記述できる。今回は T が型変数であるため、型 T を通常の構造体の型のように見て実装することが可能である。ただし、 型変数を含む Interface および Impl は、名前の後ろに<T>のように型変数を付け加える必要がある。

ソースコード 6.20: ジェネリクスを使った AtomicT の実装

```
#include "../context.h"
#impl "AtomicT.h" as "AtomicTImpl.h"
#include <stdio.h>

AtomicT<T> *createAtomicTImpl(struct Context* context, T init){
    struct AtomicT<T>* atomicT = new AtomicT();
    struct AtomicTImpl<T>* atomicT_impl = new AtomicTImpl();
    atomicT->atomicT = (union Data *)atomic_t_impl;
...
    return atomicT;
}

__code checkAndSet_AtomicTImpl(struct AtomicTImpl* atomicT_impl, T
    oldData, T newData, __code next(...), __code fail(...)){
    if (__sync_bool_compare_and_swap(&atomicT->atomic, init, newData)){
        goto next(...);
    }
    goto fail(...);
}
```

6.8.4　DataGear 定義内でのジェネリクスの型決定

ジェネリクスに実際の型を入れる際に、DataGear の定義時に決定するケースがある。ソースコード 6.21 は、Phils Interface の Impl である PhilsImpl の型定義ファイルである。Impl が持つ変数として、AtomicT に int を具体的な型として与えた型を定義している。

ソースコード 6.21: Impl ファイル内でのジェネリクスの型の決定

```
typedef struct PhilsImpl <> impl Phils {
  int self;
  AtomicT<int> Leftfork;
  AtomicT<int> Rightfork;
  __code next(...);
} PhilsImpl;
```

6.8.5　CodeGear 定義内でのジェネリクスの型決定

型定義をヘッダファイルでする以外の方法として、 CodeGear で定義するケースもある。ソースコード 6.22 に示す例では、 Interface である AtomicT は int 型が型決定される。実装のコンストラクタである createAtomicTImpl も、int 型で型決定されている。

ソースコード 6.22: Generics の CodeGear 内での型決定

```
__code createTask1(struct LoopCounter* loopCounter, struct TaskManager*
    taskManager) {
     AtomicT<int>* fork0 = createAtomicTImpl<int>(contex,-1); // model
    checking : fork0
     AtomicT<int>* fork1 = createAtomicTImpl<int>(contex,-1); // model
    checking : fork1
     AtomicT<int>* fork2 = createAtomicTImpl<int>(contex,-1); // model
    checking : fork2
     AtomicT<int>* fork3 = createAtomicTImpl<int>(contex,-1); // model
    checking : fork3
     AtomicT<int>* fork4 = createAtomicTImpl<int>(contex,-1); // model
    checking : fork4

     Phils* phils0 = createPhilsImpl(context,0,fork0,fork1); // model
    checking : phils0
     Phils* phils1 = createPhilsImpl(context,1,fork1,fork2); // model
    checking : phils1
...
```

6.8.6　ジェネリクスの型決定手法

確認したように、ジェネリクスは型変数の定義と型に具体的な値をいれる 2 種類の使い方がある。ジェネリクスを導入する場合、これらのどちらの文脈で使われているかを Perl トランスパイラ側で判定する必要がある。

これには Interface のパーサーで取得できる、型変数のリスト情報を使用する。ジェネリクスは<>記号の中に型変数、もしくは具体的な型をいれる。解析しているファイルに登場する<>の中の文字列が、型変数として登録されていた場合は、そのソースコードは具体的な型をいれていない型定義ファイルであると捉える。逆に型変数情報になかった場合は

具体的な型が決定された状況である。この場合は、どの DataGear であるかと、どの型に決定されたかを記憶する。

6.8.7　ジェネリクスの型生成

ジェネリクスに代入された具体的な型が決定した後は、ジェネリクスの型をそれぞれの具体的な型に合わせて変形させる必要がある。これは GearsOS は CbC 上に実装されているが、CbC は型変数や、ジェネリクスの<>の様な構文をサポートしていない為である。つまり、ジェネリクスで拡張された構文を、等価な CbC のソースコードに書き換える必要がある。

GearsOS では次のアルゴリズムで型を変形する。

- Interface の名前の末尾に_具体的な型を与える
  - AtomicT Interface に int を与えた場合、 AtomicT_int
- Impl の名前の末尾に_具体的な型を与える
  - AtomicTImpl の場合 int を与えた場合、 AtomicT_intImpl_int

この操作はすべての CbC ファイルについて操作する必要があるため、generate_context.pl 内で行われる。すべての CbC ファイル、ヘッダファイルについて調査を行い、型変数と具体的な型の組を作製する。

CodeGear と DataGear によって、型決定後のオペレーションが異なる。DataGear の場合は context.h に書き込む union Data 型で計算で使うすべての型が決定する。そのため、union Data 型を作製する DataGear の集合から、型変数を持つ型を削除し、代わりに具体的な型をあてはめた型をいれる。これによって、ジェネリクスで確定した型が context.h に書き込まれるようになる。

CodeGear の場合は、変換した.c ファイルを再度開き、使用されているジェネリクスの記述を置換する。置換自体は上記のアルゴリズムに沿って行われ、型変数の宣言に使われる<>記法は削除される。

型変数を持つ Interface/Impl のコンストラクタが含まれるファイルは、対応する具体的な型の分コードを生成する。関数名も置換されるため衝突は発生しない。実際に変換したコードをソースコード 6.23 に示す。この例では、AtomicT に int を具体的な型として与えていた為、AtomicT_int 型が生成されている。

ソースコード 6.23: AtomicT の型をジェネリクスによって AtomicT_int に置換した例

```
1 __code putdown_rforkPhilsImpl(struct Context *context,struct PhilsImpl*
     phils, enum Code next) {
2     struct AtomicT_int* right_fork = phils->Rightfork;
```

```
3     Gearef(context, AtomicT_int)->atomicT_int = (union Data*) right_fork;
4     Gearef(context, AtomicT_int)->newData = -1;
5     Gearef(context, AtomicT_int)->next = C_putdown_lforkPhilsImpl;
6     goto meta(context, right_fork->set);
7 }
```

6.9　generate_stub.pl のデバッグ機能の追加

変換された GearsOS のコードが意図しない結果になっていた場合、generate_stub.pl のデバッグをする必要がある。Perl スクリプトであるので Perl のデバッガを使えばデバッグは可能である。しかし、変換する GearsOS の行数もあり、さらに generate_stub.pl 自体の複雑度から、バグを生じている場所の検討をつけるのが難しい。

generate_stub.pl は巨大な正規表現パターンマッチで構成されたスクリプトであるので、CbC のコードのどの行を呼んでいる時に、 Perl スクリプトのどの行にマッチしたかが重要となる。本来マッチしてほしい正規表現パターンにマッチしていないケースは、どの行にマッチしたかのログが解れば一発で確認することができる。また、怪しい正規表現パターンの行に Perl デバッガで break point を張ってデバッグすることも可能である。

この機能を generate_stub.pl の起動時オプションの形で実装した。`--debug` オプションをつけると、デバッグ表示が行われる。(ソースコード 6.24、 6.25)

ソースコード 6.24: generate_stub.pl のデバッグモードでの起動

```
1 $ perl generate_stub.pl --debug examples/DPP2/PhilsImpl.cbc
```

ソースコード 6.25: マッチした Perl スクリプトの行番号と、CbC コードの対応表示

```
1  [getDataGear] match 199 : #impl "Phils.h" as "PhilsImpl.h"
2  [getDataGear] match 142 : typedef struct Phils <> {
3  [getDataGear] match 353 :   __code putdown_lfork(Impl* phils, __code next
       (...));
4  [getDataGear] match 353 :   __code putdown_rfork(Impl* phils, __code next
       (...));
5  [getDataGear] match 353 :   __code thinking(Impl* phils, __code next(...)
       );
6  [getDataGear] match 353 :   __code pickup_rfork(Impl* phils, __code next
       (...));
7  [getCodeGear] match 409 : typedef struct Worker<>{
8  [getCodeGear] match 414 :     __code taskReceive(Impl* worker, struct
       Queue* tasks);
9  [getCodeGear] match 414 :     __code shutdown(Impl* worker);
10 [getCodeGear] match 414 :     __code next(...);
11
12 ...
13
14 [getDataGear] match 330 : } TaskManager;
```

```
15 [getCodeGear] match 409 : typedef struct TaskManager<>{
16 [getCodeGear] match 414 :   __code spawn(Impl* taskManager, struct
       Context* task, __code next(...));
17 [getCodeGear] match 414 :   __code spawnTasks(Impl* taskManagerImpl,
       struct Element* taskList, __code next1(...));
18 [getCodeGear] match 414 :   __code setWaitTask(Impl* taskManagerImpl,
       struct Context* task, __code next(...));
19 [getCodeGear] match 414 :   __code shutdown(Impl* taskManagerImpl, __code
        next(...));
20 [getCodeGear] match 414 :   __code incrementTaskCount(Impl*
       taskManagerImpl, __code next(...));
21 [getCodeGear] match 414 :   __code decrementTaskCount(Impl*
       taskManagerImpl, __code next(...));
22 [getCodeGear] match 414 :   __code next(...);
23 [getCodeGear] match 414 :   __code next1(...);
24 [getDataGear] match 159 : Phils* createPhilsImpl(struct Context* context,
        int id, AtomicT_int* right, AtomicT_int* left) {
25 ...
```

Perl スクリプトでどの行にマッチしたかの情報は、 Perl の特殊変数__LINE__を利用した。この特殊変数は、特殊変数を呼び出した Perl の行番号が取得できるメタ API である。

ソースコード 6.26: debug_print

```
1 sub debug_print {
2   my ($functionName, $lineNumber, $line) = @_;
3   print "[$functionName] match $lineNumber : $line";
4 }
```

ソースコード 6.27: debug_print の呼び出し

```
1 if (/^typedef struct (\w+)\s*<(.*)>/) {
2     debug_print("getDataGear",__LINE__, $_) if $opt_debug;
3     $inTypedef = 1;
```

6.10　GearsOS 初期化コードの自動生成

GearsOS では、TaskManager や Worker、Context の初期化を起動時に行わなければならない。GearsOS の例題では、これらの初期化関数や、初期化に関連する CodeGear はほぼ共通であった。

新しい例題を実装する際に、これらの初期化ルーチンを毎回コピー&ペーストし、実際に記述したい箇所を変更するという手法がとられていた。この初期化ルーチンも一種のメタ的な操作であるために、ユーザーから分離させたい。そこで自動で初期化ルーチンを作製する、特殊な CodeGear の定義である gmain を定義した。

gmain で定義した main ファイルをソースコード 6.28 に示す。gmain を使用する CbC ファイルは、 他のファイルと変わらずに、 Interface を使う場合は#interface 構文で呼

び出す。通常と異なるのは、 GearsOS を停止させる CodeGear として shutdown が API として定義されている。goto で継続する CodeGear の引数として、この shutdown を渡すと、最後に GearsOS の終了ルーチンに継続するようになる。これを main.cbc として定義する。

ソースコード 6.28: gmain を使った MainCodeGear 定義

```
#interface "Stack.h"
#interface "StackTest.h"

__code gmain(){
    Stack* stack = createSingleLinkedStack(context);
    StackTest* stackTest = createStackTestImpl3(context);
    goto stackTest->insertTest1(stack, shutdown);
}
```

generate_stub.pl で、main.cbc を変換した後をソースコード 6.29 に示す。

ソースコード 6.29: gmain 定義の変換後の CbC Code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <unistd.h>
#include "../../../context.h"

int cpu_num = 1;
int length = 102400;
int split = 8;
int* array_ptr;
int gpu_num = 0;
int CPU_ANY = -1;
int CPU_CUDA = -1;

__code initDataGears(struct Context *context,struct LoopCounter*
   loopCounter, struct TaskManager* taskManager) {
    loopCounter->i = 0;
    taskManager->taskManager = (union Data*)createTaskManagerImpl(context
   , cpu_num, gpu_num, 0);
    goto meta(context, C_prevTask);
}

__code initDataGears_stub(struct Context* context) {
    LoopCounter* loopCounter = Gearef(context, LoopCounter);
    TaskManager* taskManager = Gearef(context, TaskManager);
    goto initDataGears(context, loopCounter, taskManager);
}

__code prevTask(struct Context *context,struct LoopCounter* loopCounter)
   {
    goto meta(context, C_createTask);
}
```

```
31
32
33 __code prevTask_stub(struct Context* context) {
34     LoopCounter* loopCounter = Gearef(context, LoopCounter);
35     goto prevTask(context, loopCounter);
36 }
37
38 __code createTask(struct Context *context,struct LoopCounter* loopCounter
       , struct TaskManager* taskManager) {
39     Stack* stack = createSingleLinkedStack(context);
40     StackTest* stackTest = createStackTestImpl3(context);
41     Gearef(context, StackTest)->stackTest = (union Data*) stackTest;
42     Gearef(context, StackTest)->stack = stack;
43     Gearef(context, StackTest)->next = C_shutdown;
44     goto meta(context, stackTest->insertTest1);
45 }
46
47 __code createTask_stub(struct Context* context) {
48     LoopCounter* loopCounter = Gearef(context, LoopCounter);
49     TaskManager* taskManager = Gearef(context, TaskManager);
50     goto createTask(context, loopCounter, taskManager);
51 }
52
53 __code shutdown(struct Context *context,struct TaskManager* taskManager)
       {
54     Gearef(context, TaskManager)->taskManager = (union Data*) taskManager
       ;
55     Gearef(context, TaskManager)->next = C_exit_code;
56     goto meta(context, taskManager->shutdown);
57 }
58
59 __code shutdown_stub(struct Context* context) {
60     goto shutdown(context, &Gearef(context, TaskManager)->taskManager->
       TaskManager);
61 }
62
63
64
65 void init(int argc, char** argv) {
66     for (int i = 1; argv[i]; ++i) {
67         if (strcmp(argv[i], "-cpu") == 0)
68             cpu_num = (int)atoi(argv[i+1]);
69         else if (strcmp(argv[i], "-l") == 0)
70             length = (int)atoi(argv[i+1]);
71         else if (strcmp(argv[i], "-s") == 0)
72             split = (int)atoi(argv[i+1]);
73         else if (strcmp(argv[i], "-cuda") == 0) {
74             gpu_num = 1;
75             CPU_CUDA = 0;
76         }
77     }
78 }
79
```

```
80 int main(int argc, char** argv) {
81     init(argc, argv);
82     struct Context* main_context = NEW(struct Context);
83     initContext(main_context);
84     main_context->next = C_initDataGears;
85     goto start_code(main_context);
86 }
```

変換した結果では、コマンドライン引数で GearsOS の Worker の数などを指定できる関数 init や、 TaskManager の初期化を行う initDataGears、GearsOS を終了する CodeGear である shutdown などが生成される。gmain のスコープの中に記述したものは、createTask の中に移動される。createTask では、GearsOS の起動に必要な TaskManager などの引数が受け渡されるようになっている。これによって GearsOS の例題を実装する際に、考慮しなければいけない煩雑な設定を緩和することが可能となった。

第7章　評価

7.1　GearsOSの構文作製

GearsOSで使われるInterface、およびそのImplementの型定義ファイルを導入した。GearsOSでプログラミングする際に通常のC言語やJavaなどの言語の様に、まず型を作成してからプログラミングすることが可能になった。言語機能としてはC言語や純粋なCbCより進化しており、現代的と言われるRustやgolangと比較しても十分に実用的な言語になったと言える。

ただし現状のGearsOSでは1ファイルに1つの型定義しかできない。アプリケーションとしてGearsOSを動かす現在の例題ではそこまで問題になっていない。しかし、CbCでxv6などの実用的なアプリケーションを実装する場合は、ファイルの数が莫大になる可能性がある。1ファイル内で様々な型が定義可能になれば、より見通しの良いプログラミングが可能であると考えられる。

7.2　GearsOSのトランスパイラ

GearsOSのトランスパイラは、従来はエラーを出さずCbCファイルを変換するだけであった。本研究によってPerlトランスパイラレベルでのエラーの生成を可能にし、GearsOSのメタレベルのコードを読まずともバグ検知が可能となった。またInterfaceのパーサーなどのAPIを定義したことによって、トランスパイラ側での様々な処理の拡張性を高めることが可能となった。

今まではGearsOSのモジュール化の仕組みとして使っていたInterfaceであるが、より一般的なInterfaceの使いかたに近づいたと言える。特に今まではInterfaceで定義したAPIを満たしていなくても、GearsOSは容赦なくビルドを進めてしまう問題があった。これによりメタコードが含まれた状態でしかコンパイルエラーを発見できなかった。またコンパイルエラーも出ず、動作させてみないと解らないエラーも存在していた。

Interface機能が充実したことにより、これらのエラーがビルド時に明確に解るようになった。従来行っていたデバッグもコンパイルエラーが発生するため不必要になり、よりプログラミングすることがやりやすい言語になった。また、目的としていたメタレベル

とノーマルレベルの分離を、コンパイルエラーという観点でもできるようになったと考える。

しかし導入したジェネリクス機能については議論の余地がある。Perl トランスパイラ側での実装を行ったが、非常に実装するのが困難であった。これは C 言語から派生した GearsOS のシンタックスと、それを正規表現を主に使ってコード変換を行う Perl トランスパイラが、ジェネリクスとの相性が合わなかった為である。ジェネリクスを使う場合、GearsOS のコードを字句解析や構文解析を詳細にする必要があることが実装を通して判明した。しかしこれを Perl トランスパイラで行うと、Perl 側で CbC コンパイラを実装することになる。これはもはや CbC コンパイラ側にジェネリクスを導入したほうが信頼性が高い。さらに Perl トランスパイラは基本置換で処理を行う。ジェネリクスの場合、型名を置換する必要があり、不必要な場所まで置換してしまう恐れがある。その為 Perl トランスパイラレベルではジェネリクスのサポートを続けるのは困難であると考える。

7.3　GearsOS のメタ計算

従来の GearsOS では context.h や StubCodeGear などのメタレベルのコードは、比較的手動で実装する必要があった。自由度は高いものの、様々な場面で登場するメタレベルのコードをすべて書くのは煩雑である。本研究によってメタ計算の自動生成機能がより強化され、さらにメタレベルの計算を活用することが可能となった。特に meta.pm による MetaCodeGear の切り替えは強力であり、任意の MetaCodeGear に継続することで、モデル検査や GearsOS のデバッグ機能などを組み込むことが可能となった。

第8章　結論

本研究ではGearsOSの構文の拡張、およびトランスパイラの機能の強化を行った。Interfceの定義ファイルの簡潔化や、Implementの型定義ファイルの導入などでより形式的にプログラミングすることが可能となった。また、型定義ファイルを入れたことにより、Interfaceなどを扱う様々な機能の自動化が可能となった。特に今までは手動で実装していたcontext.hの型定義部分を、ビルド時に自動生成することが可能となった。メタレベルの計算を柔軟に扱いつつ、従来手作業で実装していた様々なメタ計算を自動生成できた。

Interfaceの型定義ファイルを取り扱う様々なAPIを用意した。従来はgenerate_stub.plで呼び出されていた簡易的なパーサーを、Interface用に再実装し、様々な情報を定義ファイルから取得できるようになった。これによりImplementのCbCのファイルの雛形生成や、 Perlインタプリタレベルでの警告の生成などが可能となった。

導入したmeta.pmによってメタ計算を自在に操作することが可能となった。また、Interface経由でのpar gotoの呼び出しが可能となった。これらの組み合わせで、並列にオブジェクトを動作させ、モデル検査させる仕組みがGearsOS上に整備された。実際にモデル検査器を現在のGearsOS上にすでに動作させたことがあり、 GearsOSを使ってOSの信頼性を向上させる目標に対して前進できたと言える。

8.1　今後の課題

8.1.1　context.h定義時の依存関係の解決

context.hの自動生成機能を実装したが、 union Data型の定義時にならべるDataGearに対応する構造体の順序が問題になる。ソースコード8.1では、構造体Aはメンバとして構造体Bを参照している。

ソースコード 8.1: エラーが出るunionの定義

```
1 union Data {
2   struct A {
3     struct B b;
4   } A;
5   struct B {
6     int i;
```

```
7   } B;
8 };
```

ポインタではなく値そのものを使う場合、 C 言語では使用する型の定義は、使う前に書かなければならない。そのため本来はソースコード 8.2 の様に定義する必要がある。

ソースコード 8.2: 構造体の定義順を考慮した union の定義

```
1 union Data {
2  struct B {
3     int i;
4   } B;
5   struct A {
6     struct B b;
7   } A;
8 };
```

GearsOS では DataGear の構造体の相互参照は基本はポインタで行われている。その為コンパイル時に致命的なエラーは存在していないが、 実装の手法によっては DataGear そのものを内包したいケースがある。この場合は context.h で定義する構造体の依存関係を調査し、適切に出力する必要がある。

現在は context.h がすでに存在していた場合 context.h の再生成は行わないようになっているため、 context.h を手で修正すればエラーは回避可能である。しかし結局手動で行ってしまっているために、 generate_context.pl で依存関係の解決を自動的に行う必要がある。

8.1.2　xv6 上での完全な動作

GearsOS の機能拡張の初期は、実装した Perl ツール群を xv6 にも移植、検証を行っていた。しかし xv6 上の開発を近頃は止めてしまっており、本研究で作ったツールが xv6 で動作するかは不明である。また、xv6 上に移植する GearsOS プロジェクトは、ファイルシステムやプロセス管理などの OS の根幹部分がまだ移植出来ていない。この為には GearsOS と同じデータ単位を使う Christie フレームワークの知見などをもとに、ファイルシステムなどを実装する必要がある。これには既存の xv6 のもつ API と GearsOS の Interface や継続を中心とする API の齟齬が問題となる。GearsOS で UNIX を表現する際に必要な API については、今後考察しなければならない。

8.1.3　Perl トランスパイラが提供する機能の GearsOS 組み込み

今の GearsOS はコンパイル時に Perl スクリプトによってメタ計算が変換されている。使用する DataGear の種類を探し、context.h の作製や CodeGear に一意にふる番号の作製

などは、本来は OS 自体が持つべき機能である。例えば使用するべき DataGear の種類を探し、context.h を作製するのは、 現状の OS のリンカとローダーに近い役割をしている。これらの機能自体を CbC、GearsOS で記述し、GearsOS のコア機能に組み込みたい。

GearsOS 自身でプログラム可能となると、GearsOS のモデル検査機能が使えるようになる。これは、OS の動作の信頼性の保証につながる。さらに GearsOS の機能として組み込むと、GearsOS の中で新たに GearsOS のプログラムを作ることも可能となる。これは GearsOS の拡張性の強化につながる。

GearsOS に Perl スクリプトの機能を組み込む場合は、CodeGear、DataGear をさらに GearsOS の中から扱う API が必要である。Perl スクリプトが提供している機能は、それ自身が巨大なメタ計算であるので、MetaCodeGear のさらなる充実が必要となる。例えばダイナミックに MetaCodeGear を生成する API などがあれば、Perl インタプリタを GearsOS のビルドシステムから落とすことが可能ではないかと考える。

8.1.4　Perl トランスパイラの処理の複雑さ

本研究で様々な機能を Perl トランスパイラに実装したが、これにより Perl トランスパイラの複雑度が増してしまった。特に generate_stub.pl は性質上グローバル変数を多用するので、 処理の全貌をつかむまでが非常に困難である。現在は Perl 標準の警告プラグマをつけており、未定義値を使ってしまった場合に警告を出すようにしているが、 これも不十分である。

モジュール化をはじめとする Perl トランスパイラの全体的なリファクタリング、もしくは再実装などをするべき段階に来ていると考える。また、Perl ではなくトランスパイラ自体も CbC で記述し、モデル検査などにつなげることも可能と言える。

8.1.5　CbC の構文上に構築していることの問題

現在の GearsOS は CbC の構文上に構築しているため、Perl トランスパイラ側で複雑な処理をしようとすると、CbC コンパイラを Perl で実装することに近い状態が発生する。CbC はアセンブラ的に使うことが可能であるので、 GearsOS を C のシンタックスとは別のシンタックスを与えた言語として制定し、 CbC を出力するように実装をし直したい。こうすることで C の構造上の問題を回避できると考える。

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。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。また、山場となったシステム更新を共に乗り越えた城後明慈さん、宮平賢さんをはじめとするシステム管理チームの皆様に感謝します。最後に、 有意義な時間を共に過ごした理工学研究科情報工学専攻の学友、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。

2021 年 3 月
清水隆博

参考文献

[1] 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.

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 並列信頼研究室. Cbc_gcc. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/. Accessed: 2021-02-02.

[13] 並列信頼研究室. Cbc_llvm. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_llvm/. Accessed: 2021-02-02.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 並列信頼研究室. Cbc_xv6. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_xv6/. Accessed: 2021-02-02.

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

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

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

[23] Java implements keyword. https://www.w3schools.com/java/ref_keyword_implements.asp.

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

[25] yaohaizh. Add unimplemented methods code action.

[26] josharian/impl. https://github.com/josharian/impl.

[27] golang. golang/vscode-go.

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

付 録A　研究会業績

A-1　研究会発表資料

- CbC を用いた Perl6 処理系 清水 隆博, 河野真治 第 60 回プログラミング・シンポジウム, Jan, 2019
- 継続を基本とした OS Gears OS 清水 隆博, 河野真治 第 61 回プログラミング・シンポジウム, Jan, 2020
- xv6 の構成要素の継続の分析 清水 隆博，河野 真治 (琉球大学), 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, 2020