修士 (工学) 学位論文
Master's Thesis of Engineering

Gears Agda での形式手法を用いたプログラムの検証
Verification of programs using formal methods in Gears Agda

2023 年 3 月
March 2023

上地 悠斗
Yuto Uechi

琉球大学
大学院理工学研究科
工学専攻
知能情報プログラム

Computer Science and Intelligent Systems
Graduate School of Engineering and Science
University of the Ryukyus

修士 (工学) 学位論文
Master's Thesis of Engineering

Gears Agda での形式手法を用いたプログラムの検証
Verification of programs using formal methods in Gears Agda

2023 年 3 月
March 2023

上地 悠斗
Yuto Uechi

琉球大学
大学院理工学研究科
工学専攻
知能情報プログラム

Computer Science and Intelligent Systems
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: Gears Agda での形式手法を用いたプログラムの検証

氏　　名: 上地 悠斗

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)　　赤嶺 有平

印

(副　査)　　山田 孝治

印

(副　査)　　河野 真治

要旨

開発手法の一つとして、形式手法というものがある。形式手法とはプログラムの仕様を形式化、つまり数学的な記述を行い、書いたプログラムがそれを満たしているか検証を行う開発手法である。代表的な形式の検証手法として、定理証明やモデル検査が挙げられる。

形式手法で開発したプログラムは数学的に正しいことが証明されている。そのため高い安全性が必要とされる鉄道や電力などのインフラ分野に使用されている。一見完璧な手法であるように思えるが欠点として、形式化や検証が複雑なため膨大なコストを要する。シンプルな実装であれば仕様の形式化が容易に行えるが、そうであれば形式手法を使用する利点が薄くなる。

そのため、他のプログラミング言語と比べて検証に適している Gears Agda を使用する。Gears Agda とは当研究室で開発している Continuation based C (CbC) の概念を取り入れた記法で記述された定理証明支援器 Agda のことである。

定理証明によるプログラムの検証は一般的に難易度が高いが、Gears Agda では検証をプログラム単位に分割することができ、比較的容易に検証を行えるようになっている。

先行研究では implies を用いて Hoare Logic での定理証明を行っていた。しかし、それでは記述が長く複雑になっていた。そこで今回は Invariant というプログラムの不変条件を新たに取り入れた。これを検証しながら実行をすることで、Hoare Logic を用いた定理証明を比較的簡単に行えるようになった。

また、定理証明では並列処理の検証は困難である。出現する状態を一度全て出してからそれらの検証を行わないといけないためである。そのため、もう一つの代表的な検証手法であるモデル検査を Gears Agda にて行えるようにした。

これらのことから、本論文では Gears Agda の定理証明とモデル検査での検証手法について述べる。

Abstract

One of the development methods is called formal methods. Formal methods formalize the specification of a program, i.e., describe it mathematically, and verify that the written program satisfies the specification. The typical formal verification methods include theorem proving and model checking. Theorem proving and model checking are typical formal verification methods.

Programs developed by formal methods are proven to be mathematically correct. For this reason, they are used in infrastructure fields such as railroads and electric power, where high safety is required. However, the drawback of this seemingly perfect method is that it is extremely costly. Formalization of specifications is easy for simple implementations, but then it is not necessary to use formal methods.

For this reason, we use Gears Agda, which is more suitable for verification than other programming languages. Gears Agda is a programming language written in a notation that incorporates the concept of Continuation based C (CbC), which is being developed in our laboratory. Gears Agda is an Agda written in a notation that incorporates the concept of Continuation based C (CbC), which is being developed in our laboratory.

Verification of programs by theorem proving is generally difficult. Gears Agda can be divided into program units, making verification relatively easy.

In previous research, implies were used for theorem proving in Hoare Logic. However, this made the description long and complicated. Therefore, in this study, we introduced a new program invariant called Invariant. By verifying the invariant condition while executing the program, theorem proving using Hoare Logic became relatively easy.

In addition, verification of concurrency is difficult in theorem proving. Therefore, we made it possible to perform model checking, another typical verification method, in Gears Agda.

Based on the above, this paper describes the theorem proving and model checking verification methods of Gears Agda.

研究関連論文業績

- 上地 悠斗, 河野 真治. Gears Agda による Red Black Tree の検証. 第 63 回プログラミング・シンポジウム, Jan, 2022
- 上地 悠斗, 河野 真治. Gears Agda 上のモデル検査の形式化. 情報処理会 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, 2022

目 次

図 目 次

ソースコード目次

第1章　Gears Agda でのプログラムの検証

OS やアプリケーションの信頼性の向上は重要である。そのための手段として、プログラムが仕様を満たしているか検証する事が挙げられる。

しかし、仕様の形式化とその検証には膨大なコストを要する。そのため、他のプログラミング言語と比べて検証に適している Gears Agda を使用する。Gears Agda とは当研究室で開発している Continuation based C [1](以下 CbC) の概念を取り入れた記法で記述された Agda [2] のことである。

通常のプログラミング言語では関数を実行する際にメインルーチンからサブルーチンに遷移する。この際メインルーチンで使用していた変数などの環境はスタックされ、サブルーチンが終了し、メインルーチンに戻る際にスタックしていた環境に戻す流れとなる。CbC の場合はサブルーチンコールの際にアセンブラでいう jmp 命令 により関数遷移を行うことができる。そのため、環境をスタックに保持しない。したがって関数の実行では暗黙な環境が存在せず、関数は受け取った引数のみを使用する。この関数の単位を CodeGear と呼び、CodeGear の引数を DataGear と呼んでいる。

これにより検証を CodeGear 単位に分割することができ、比較的容易に検証を行えるようになっている。

また、先行研究 [3] では、Gears Agda での並列実行の検証について課題が残っていた。並列実行の検証を定理証明で行うには、考慮する状態の数が膨大になり困難である。そのため、Gears Agda でモデル検査 [4] を行うことでこの課題に対応する。

CbC の継続の概念はモデル検査とも相性がよい。[5] CbC が末尾関数呼び出しであるため CodeGear をそのままモデルの edge として定義することが可能である。

これらのことから、本論文では、Gears Agda での定理証明とモデル検査について述べる。

第2章　Continuation based C

Continuation based C[6] は関数呼び出しの際に jmp 命令で遷移をし、環境を持たずに遷移することができる C 言語である。すなわち C 言語の下位言語にあたり、よりアセンブラに近い記述を行う。

本章では CbC の概要について説明する。

2.1　CodeGear / DataGear

CbC では、プログラムの単位として DataGear と CodeGear という単位を用いる。

CodeGear はプログラムの処理そのものであり、一般的なプログラム言語における関数と同じ役割である。DataGear は CodeGear で扱うデータの単位であり、処理に必要なデータである。

jmp 命令で関数遷移するため関数遷移し実行が終了しても、もとの関数に戻ることはない。そのため次に遷移する CodeGear を指定する。したがって、CodeGear に Deta Gear を与え、それをもとに処理を行い、出力として DataGear を返し、また次の CodeGear に遷移していく流れとなる。これは、関数型プログラミングでは末尾再帰をしていることと同義である。

2.2　CbC と C 言語の違い

同じ仕様で CbC と C 言語で実装した際の違いを、実際のコードを元に比較する。比較するにあたり、再起処理が含まれるコードの例として、フィボナッチ数列の n 番目を求めるコードを挙げる。C 言語で記述したものがソースコード 2.1 になり、CbC で記述したものがソースコード 2.2 になる。CbC は実行を継続するため、return を行えない。そのため C 言語での実装も return を書かず関数呼び出しを行い、最後に exit をして実行終了するように記述している。

```
1 void fin(unsigned long long n){
2   printf("%lld", n);
3     exit(0);
4 }
```

```
5
6  void fib(unsigned long long n, unsigned long long a, unsigned long long b){
7    if (n==0) fin(a);
8    if (n==1) fin(b);
9    fib(n-2, a+b, a+b+b);
10 }
11
12 int main(int argc, char *argv[]){
13     unsigned long long n=atoll(argv[1]);
14   fib(n,0,1);
15 }
```

ソースコード 2.1: C 言語で記述した フィボナッチ数列の n 番目を求めるコード

```
1  __code fin(unsigned long long n){
2    printf("%lld", n);
3  }
4
5  __code fib(unsigned long long n, unsigned long long a, unsigned long long b){
6    if (n==0) goto fin(a);
7    if (n==1) goto fin(b);
8    goto fib(n-2, a+b, a+b+b);
9  }
10
11 int main(int argc, char *argv[]){
12     unsigned long long n = atoll(argv[1]);
13   goto fib(n,0,1);
14 }
```

ソースコード 2.2: CbC で記述した フィボナッチ数列の n 番目を求めるコード

軽量実装になっているのか、上記のコードをアセンブラ変換した結果を見て確認する。全てを記載すると長くなるので、アセンブラ変換した際の fib 関数のみを記載する。C 言語で記述したコードをアセンブラ変換した結果がソースコード 2.3。CbC で記述したコードをアセンブラ変換した結果がソースコード 2.4 になる。

比較すると、fib 関数の内部で再度 fib 関数を呼び出す際、C 言語で実装したソースコード 2.3 の 30 行目では callq で fib 関数を呼び出している。対して CbC で実装したソースコード 2.4 の 32 行目では、jmp により fib 関数に移動している。

```
1  0000000100000e52 _fib:
2  100000e52: 55 pushq %rbp
3  100000e53: 48 89 e5 movq %rsp, %rbp
4  100000e56: 48 83 ec 20 subq $32, %rsp
5  100000e5a: 48 89 7d f8 movq %rdi, -8(%rbp)
6  100000e5e: 48 89 75 f0 movq %rsi, -16(%rbp)
7  100000e62: 48 89 55 e8 movq %rdx, -24(%rbp)
8  100000e66: 48 83 7d f8 00 cmpq $0, -8(%rbp)
9  100000e6b: 75 0c jne 12 <_fib+0x27>
10 100000e6d: 48 8b 45 f0 movq -16(%rbp), %rax
11 100000e71: 48 89 c7 movq %rax, %rdi
12 100000e74: e8 ab ff ff ff callq -85 <_fin>
13 100000e79: 48 83 7d f8 01 cmpq $1, -8(%rbp)
14 100000e7e: 75 0c jne 12 <_fib+0x3a>
15 100000e80: 48 8b 45 e8 movq -24(%rbp), %rax
16 100000e84: 48 89 c7 movq %rax, %rdi
```

```
100000e87: e8 98 ff ff ff callq -104 <_fin>
100000e8c: 48 8b 55 f0 movq -16(%rbp), %rdx
100000e90: 48 8b 45 e8 movq -24(%rbp), %rax
100000e94: 48 01 c2 addq %rax, %rdx
100000e97: 48 8b 45 e8 movq -24(%rbp), %rax
100000e9b: 48 01 c2 addq %rax, %rdx
100000e9e: 48 8b 4d f0 movq -16(%rbp), %rcx
100000ea2: 48 8b 45 e8 movq -24(%rbp), %rax
100000ea6: 48 01 c1 addq %rax, %rcx
100000ea9: 48 8b 45 f8 movq -8(%rbp), %rax
100000ead: 48 83 e8 02 subq $2, %rax
100000eb1: 48 89 ce movq %rcx, %rsi
100000eb4: 48 89 c7 movq %rax, %rdi
100000eb7: e8 96 ff ff ff callq -106 <_fib>
100000ebc: 90 nop
100000ebd: c9 leave
100000ebe: c3 retq
```

ソースコード 2.3: c で記述した際の fib 関数のアセンブラ

```
0000000100000e3a _fib:
100000e3a: 55 pushq %rbp
100000e3b: 48 89 e5 movq %rsp, %rbp
100000e3e: 48 89 7d f8 movq %rdi, -8(%rbp)
100000e42: 48 89 75 f0 movq %rsi, -16(%rbp)
100000e46: 48 89 55 e8 movq %rdx, -24(%rbp)
100000e4a: 48 83 7d f8 00 cmpq $0, -8(%rbp)
100000e4f: 75 0d jne 13 <_fib+0x24>
100000e51: 48 8b 45 f0 movq -16(%rbp), %rax
100000e55: 48 89 c7 movq %rax, %rdi
100000e58: 5d popq %rbp
100000e59: e9 b5 ff ff ff jmp -75 <_fin>
100000e5e: 48 83 7d f8 01 cmpq $1, -8(%rbp)
100000e63: 75 0d jne 13 <_fib+0x38>
100000e65: 48 8b 45 e8 movq -24(%rbp), %rax
100000e69: 48 89 c7 movq %rax, %rdi
100000e6c: 5d popq %rbp
100000e6d: e9 a1 ff ff ff jmp -95 <_fin>
100000e72: 48 8b 55 f0 movq -16(%rbp), %rdx
100000e76: 48 8b 45 e8 movq -24(%rbp), %rax
100000e7a: 48 01 c2 addq %rax, %rdx
100000e7d: 48 8b 45 e8 movq -24(%rbp), %rax
100000e81: 48 01 c2 addq %rax, %rdx
100000e84: 48 8b 4d f0 movq -16(%rbp), %rcx
100000e88: 48 8b 45 e8 movq -24(%rbp), %rax
100000e8c: 48 01 c1 addq %rax, %rcx
100000e8f: 48 8b 45 f8 movq -8(%rbp), %rax
100000e93: 48 83 e8 02 subq $2, %rax
100000e97: 48 89 ce movq %rcx, %rsi
100000e9a: 48 89 c7 movq %rax, %rdi
100000e9d: 5d popq %rbp
100000e9e: eb 9a jmp -102 <_fib>
```

ソースコード 2.4: cbc で記述した際の fib 関数のアセンブラ

2.3　Meta CodeGear / Meta DataGear

プログラムを記述する際に、メモリ管理、スレッド管理、資源管理等のプログラムの本筋とは別に実装が必要な場合がある。これらの計算をメタ計算と呼ぶ。

CbC ではメタ計算を分離するために Meta CodeGear、Meta DataGear を定義している。Meta CodeGear ではこのメタ計算を行い、Meta DataGear は CbC 上のメタ計算で扱われる DataGear である。例えば stub CodeGear では Context と呼ばれる接続可能な CodeGear、DataGear のリストや、DataGear のメモリ空間等を持った Meta DataGear を扱っている。

図 2.1 のように CodeGear を実行する前後や DataGear を内包する Meta Gear が存在している。

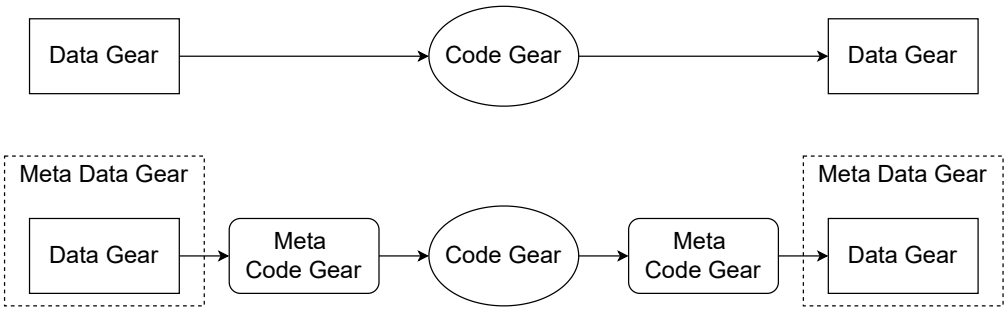

図 2.1: メタ計算を可視化した CodeGear と DataGear

第3章　定理証明支援系言語 Agda

Agda [7] [8] は純粋関数型言語である。Agda は依存型という型システムを持ち、型を第一級オブジェクトとして扱う。

もともとの Agda には自然数などの普遍的な定義すら存在しない。その実装ができることも本章で取り扱っているが、一般的には agda-standard-libralies [9] を使用する。

本章では Agda の基本と Agda による定理証明の方法について述べる。

3.1　Agdaの基本

本節では Agda の基本事項についてソースコードを例に出しながら説明を行う。

3.1.1　関数の実装

Agda での関数の 定義方法について ソースコード 3.1 を例として解説する。基本事項として、ℕ というのは自然数 (Natulal Number) のことである。また - (ハイフン) が 2 つ連続して並んでいる部分はコメントアウトであり、ここでは関数を実行した際の例を記述している。したがって、これは 2 つの自然数を受け取って足す関数であることが推測できる。

```
1 plus : (x y : ℕ) → ℕ
2 plus x zero = x
3 plus x (suc y) = plus (suc x) y
4
5 -- plus 10 20
6 -- 30
```

ソースコード 3.1: plus の実装

この関数の定義部分の説明をする。コードの 1 行目に : (セミコロン) がある。この : の前が関数名になり、その後ろがその関数の定義となる。: 以降の (x y : ℕ) は関数は x, y の自然数 2 つを受けとるという意味になる。→ 以降は関数が返す型を記述している。まとめると、この関数 plus は、型が自然数である 2 つの変数 x, y を受け取り、自然数を返すという定義になる。

Agda では関数の定義をしたコードの直下で実装を行うのが常である。関数名を記述した後に引数を記述して受け取り、= (イコール) 以降で引数に対応した実装をする。

今回の場合 plus x zero であれば +0 である為、そのまま x を返す。実装 2 行目の方で受け取った y の値を減らし、x の値を増やして再び plus の関数に遷移している。受け取った y を +1 されていたとして y の値を減らしている。

関数の実装全体をまとめると、x と y の値を足す為に y から x に数値を 1 つずつ渡す。y が 0 になった際に計算が終了となっている。指折りでの足し算を実装していると捉えても良い。

3.1.2　二項演算子の実装

_ (アンダースコア) を用いることで入力を受け取る事ができる。これを用いることで、二項演算子を実装することができる。以下に、二項演算子を使用した ソースコード 3.1 と同義の関数の例を以下 ソースコード 3.2 挙げる。

```
1 _+_ : (x y : ℕ) → ℕ
2 x + zero = x
3 x + suc y = (suc x) + y
4
5 -- 10 + 20
6 -- 30
```

ソースコード 3.2: 二項演算子を用いた plus の実装

利点としては、直感的な記号論理の記述ができる。以下、記号論理は基本的に二項演算子を使用して記述する。

3.1.3　Data 型の実装

Deta 型とは分岐のことである。そのため、それぞれの動作について実装する必要がある。例として既出の Data 型である ℕ の実装を ソースコード 3.3 に示す。

```
1 data ℕ : Set where
2   zero : ℕ
3   suc : (n : ℕ) → ℕ
```

ソースコード 3.3: Natural の定義

実装から、ℕ という型は zero と suc の 2 つのコンストラクタを持っていることが分かる。それぞれの仕様を見てみると、zero は ℕ のみであるが、suc は (n : ℕ) → ℕ である。つまり、suc 自体の型は ℕ であるが、そこから ℕ に遷移するということである。そのため、suc からは suc か zero に遷移する必要があり、また zero に遷移することで停止する。したがって、数値は zero に遷移するまでの suc が遷移した数によって決定される。

Data 型にはそれぞれの動作について実装する必要があると述べたが、言い換えればパターンマッチをする必要があると言える。これは puls 関数で suc 同士の場合と、zero が含まれる場合の両方を実装していることの説明となる。

3.1.4　Record 型の実装

Record 型とはオブジェクトあるいは構造体ののようなものである。ソースコード 3.4 は AND の関数となる。p1 で前方部分が取得でき、p2 で後方部分が取得できる。

```
record _∧_ (A B : Set) : Set where
  field
    p1 : A
    p2 : B
```

ソースコード 3.4: And の記述

定義は「A ならば B」かつ「B ならば C」なら「A ならば C」となる。ソースコード 3.5 を以下に示す。

```
syllogism : {A B C : Set} → ((A → B) ∧ (B → C)) → (A → C)
syllogism x a = _∧_.p2 x (_∧_.p1 x a)
```

ソースコード 3.5: syllogism の記述

コードの解説をすると、引数として x と a が関数に与えられている。引数 x の中身は ((A → B) ∧ (B → C)) 、引数 a の中身は A である。したがって、(_∧_.p1 x a) で (A → B) に A を与えて B を取得し、_∧_.p2 x で (B → C) であるため、これに B を与えると C が取得できる。よって A を与えて C を取得することができたため、三段論法を定義できた。

3.2　本論で使用する Agda の記法

本論では、ソースコードを出し、それの実施内容について述べるが、特殊な記法を用いている場合があるので、前もって解説をする。

3.2.1　Agda の省略記法

Recode が入力された場合のことを考える。この際、入力時に record を展開してしまうと、コードが長くなってしまい、煩雑になってしまう。これを防ぐために、with を使用し、必要な変数のみ取り出してパターンマッチを行う。例を ソースコード 3.6 に示す。

```
record env : Set where
  field
    a : ℕ
    b : ℕ
    c : ℕ
open env

patternmatch-default : env → ℕ
patternmatch-default record { a = a ; b = b ; c = c } = c

patternmatch-extraction : env → ℕ
patternmatch-extraction env with c env
patternmatch-extraction env | c = c

patternmatch-extraction′ : env → ℕ
patternmatch-extraction′ env with c env
... | c = c
```

ソースコード 3.6: 入力を省略する Agda コードの例

patternmatch-default は入力されている record をそのまま展開することで、値を取得している。

patternmatch-extraction では、with を使用して入力されている record の中から対象の値だけ取得している。このように、入力時に record を展開せずに中の値を取得することもできる。

patternmatch-extraction' では、入力が同じ場合に ... で省略ができることを使用し、さらに省略を行っている。

今後のソースコードでは、必要な変数のみ取り出すことでコードを見やすくする。

3.3　Agda による定理証明

本説では、実際に Agda による定理証明の方法を簡単な例とその応用で可能な加算の交換法則の証明を用いて説明する。

3.3.1　自然数に 0 を足しても値が等しいことを証明する Agda

ソースコード 3.7 は型定義にも書いてあるとおり、自然数に 0 を足しても値が等しいことを証明するコードとなっている。このように、Agda では証明したい論理式を型定義に書き、実装部分に当たる 2 行目以降に証明を書くことができる。

```
+zero : { y : ℕ } → y + zero ≡ y
+zero {zero} = refl
+zero {suc y} = cong suc ( +zero {y} )
```

ソースコード 3.7: 自然数に 0 を足しても値が等しいことを証明する Agda

実際にコードの中で行われている証明について説明する。{zero} のパターンは y が 0 であったときの場合のことである。0 に 0 を足しても 0 になるのは自明である。そのため refl (reflexivity) が使える。これは Agda で ≡ の両辺が等しい場合に使われる記法である。つまり、$zero \equiv zero$ ということであり、それは証明として正しいことが得られた。

{suc y} は y が 0 以上、つまり 1 以上のパターンである。前述した zero の場合は簡単に証明できた。ここで、足し算の実装を行った際のように、y が 0 であった場合に持っていくことができれば証明が行えることがわかる。そのために、ソースコード 3.8 の cong を使用する。

```
cong : ∀(f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

ソースコード 3.8: cong の定義

これは、$x \equiv y$ の時 $fx \equiv fy$ が成り立つという関数になる。ここで注目すべきは $x \equiv y \rightarrow fx \equiv fy$ ということだ。つまり、f を suc に見立てることで、$zero \equiv zero \rightarrow suc\, zero \equiv suc\, zero$ が見えてくる。したがって、{suc y} の場合は $cong\, suc\, (+zero\, \{y\})$ となる。これにより y を減少させ続ける。0 まで到達すると refl を得られる。あとは cong を使っていることにより元の値まで suc される流れとなる。以上のことから、自然数に 0 を足しても値が等しいことを証明できた。

3.3.2　加算の交換法則を証明する Agda

次に、加算の交換法則を証明する。ソースコード 3.9 になる。

再び zero であった場合から考える。今回は受け取る引数は 2 つあるが x を基準としてパターンマッチさせている。x が 0 とは、y に 0 を足すということである。それは前項にてすでに行っている。そのため、rewrite による変形を使用する。これは = の前に rewrite 構文を使用することで等式を変形規則で記述することができる。

次に x が zero 以外の場合を考える。今回は ≡-Reasoning により式を展開している。

```
+-comm : (x y : ℕ) → x + y ≡ y + x
+-comm zero y rewrite (+zero {y}) = refl
+-comm (suc x) y = let open ≡-Reasoning in
  begin
  (suc x) + y ≡⟨⟩
  suc (x + y) ≡⟨ cong suc (+-comm x y) ⟩
  suc (y + x) ≡⟨ sym (+-suc {y} {x}) ⟩
  y + suc x ■

-- +-suc : {x y : ℕ} → x + suc y ≡ suc (x + y)
-- +-suc {zero} {y} = refl
-- +-suc {suc x} {y} = cong suc (+-suc {x} {y})
```

ソースコード 3.9: 加算の交換法則を証明する Agda

これにより $x + y \equiv y + x$ つまり $(suc\, x) + y \equiv y + (suc\, x)$ を求めている構文を説明すると、begin が始まりで ■ で終わりとなる。■ は emacs-agda では \qed で入力することができ、意味はそのまま Quod Erat Demonstrandum (証明終了) となる。証明の内部では、証明として正しいことが得られた ≡ の左項が始まる式であり、次の行の左項との一致を証明するのが右項となっている。5 行目は自明であるため証明を省略できる。6 行目は $suc(x + y) \equiv suc(y + x)$ の証明をしている。これは cong suc を使ってこの関数自体を呼び出せば良い。7 行目では $suc(y + x) \equiv y + suc\, x$ の証明を行っている。これについては既存の方法ではできないので、10 行目の+-suc に sym をかけている。sym は $x \equiv y \rightarrow y \equiv x$ であり、等式なら左右を入れ替えてもよいとなる。+-suc では再帰的に x を減少させることで refl を得るようになっている。

これらのことから、$x + y \equiv y + x$ を求めることができたため、Agda にて加算の交換法則を証明することができた。Agda ではこのような形で等式を変形しながら証明を行う事ができる。

第4章　GearsAgda 形式で書く Agda

CbC の継続の概念を取り入れた Agda の記法を説明する。Agda では関数の再帰呼び出しが可能であるが、CbC では値が帰って来ない。そのため Agda で実装を行う際には再帰呼び出しを行わないようにする。

以下で Gears Agda の記述方法を足し算を行うプログラムを用いて説明する。

```
1 record Env : Set where
2   field
3     varx : ℕ
4     vary : ℕ
5 open Env
```

ソースコード 4.1: Agda での DataGear の定義

```
1 plus-c : {l : Level} {t : Set l} → Env → (exit : Env → t) → t
2 plus-c env exit = plus-p (vary env) env exit where
3   plus-p : {l : Level} {t : Set l} → ℕ → Env → (exit : Env → t) → t
4   plus-p zero env exit = exit env
5   plus-p (suc reducer) env exit = plus-p reducer record env{varx = (suc (varx env)) ;
        vary = reducer} exit
```

ソースコード 4.2: Agda での CodeGear の定義

ソースコード 4.1 が DataGear の定義をしている。今回は足し算を実装するので、varx に vary を足すことを考える。そのためそれらが 2 つの自然数を持つようにしている。

ソースコード 4.2 では CodeGear の定義になる。最初に DataGear となる env を受け取ったあと、そのまま次の関数に遷移させている。

Agda の記述は Curry-Howard 対応になっていて、最初に関数名のあとに:（コロン）の後ろに命題を記述し、そのあとに関数名のあとに引数を書き、=（イコール）の後ろに定義を記述している。

Gears Agda での CodeGear の命題は必ず （Env → t） → t で終了するようになっている。この （Env → t） は引数で受け取る型で Env を受け取って t を返すという意味になる。これが CodeGear を実行したあとの末尾関数呼び出しを行う次の CodeGear となる。最後に t を返すのは、これ自体が CodeGear であることを示している。

引数を受け取ったあとに別の関数に再度渡している。これは、Agda の繰り返し処理を行う際に停止性を見失うために減少列を引数に取っている。内部の処理は reducer を減らしながら varx を増やし、vary の値を varx に与えていくことで足し算を定義している。

基本的に繰り返し実行するコードを実装する場合には、実行時に減少しその関数がいずれ停止することを示す reducer を含めるようにしている。

対照的に reducer を含めなかった際の CodeGear を ソースコード 4.3 に示す。

```
1 {-# TERMINATING #-}
2 plus-c-term : {l : Level} {t : Set l} → Env → (exit : Env → t) → t
3 plus-c-term env exit with vary env
4 ... | zero = exit (record { varx = varx env ; vary = vary env })
5 ... | suc y = plus-c-term (record { varx = suc (varx env) ; vary = y }) exit
```

ソースコード 4.3: Agda での 停止性が示せない CodeGear の例

Agda ではパターンマッチを行うことで場合分けを考えることができるが、受け取った CodeGear である env を with を使用してパターンマッチを試みている。パターンマッチ自体は可能だが、この方法だと Agda が関数が停止することを認識できない。そのため、{-# TERMINATING #-} を関数定義の前にアノテーションし、この関数が停止することを記述してコンパイルが通るようにしている。

ソースコード 4.4 は受け取った引数で DataGear を初期化してそれを CodeGear に与えることで実行を行っている。

```
1 plus : ℕ → ℕ → Env
2 plus x y = plus-c (record { varx = x ; vary = y }) (λ env → env)
```

ソースコード 4.4: Agda での CodeGear の初期化

今回の例では 引数から DataGear を作成するのは複雑ではない。そのため、一度で DataGear を作成してそのまま CodeGear に渡している。引数から DataGear を作成するのが複雑な場合は、一度入力から DataGear を作成する CodeGear を用いる。加えて、実行なので命題の部分にある出力の型は Env になっている。

4.1 Gears Agda での Meta Gears

通常の Meta Gears はノーマルレベルの CodeGear、 DataGear では扱えないメタレベルの計算を扱う単位である。今回は定理証明やモデル検査を行う際に使用する [10]。

- Meta DataGear
  Agda 上で Meta DataGear を持つことでデータ構造自体が関係を持つデータを作ることができる。通常の DataGear を wraping している。今回はこれを用いることで、定理証明では不変状態の条件を保存し、モデル検査ではその時点での状態を保存する。

- Meta CodeGear
  Meta CodeGear は 通常の CodeGear では扱えないメタレベルの計算を扱う CodeGear

である。Agda での Meta CodeGear は Meta DataGear を引数に取りそれらの関係を返す CodeGear である。今回はここで定理証明やモデル検査を行う。

第5章　Gears Agda による定理証明

先行研究 [3] では、Gears Agda ではないプログラムの Hoare Logic による検証 [11] を参考にそれを Gears Agda に適応して while loop の検証を行っていた [12]。本研究では、Invariant というプログラムの不変条件を定義し、それを検証することで、比較的容易に検証を行うことができるようになった (本章 2 節) 本章では Gears Agda による定理証明の方法について説明する。

5.1　Hoare Logic

Hoare Logic[13] とは C.A.R Hoare、R.W Floyd が考案したプログラムの検証の手法である。これは、「プログラムの事前条件 (P) が成立しているとき、コマンド (C) 実行して停止すると事後条件 (Q) が成り立つ」というものである。これは CbC の実行を継続するという性質に非常に相性が良い。Hoare Logic を表記すると以下のようになる。

$$\{P\}\ C\ \{Q\}$$

この 3 つ組は Hoare Triple と呼ばれる。

Hoare Triple の事後条件を受け取り、異なる条件を返す別の Hoare Triple を繋げることでプログラムを記述していく。

Hoare Logic の検証では、「条件がすべて正しく接続されている」かつ「コマンドが停止する」ことが必要である。これらを満たし、事前条件から事後条件を導けることを検証することで Hoare Logic の健全性を示すことができる。

つまり、Hoare Triple の事後条件が次のコマンドの事前条件になり、それを実行終了まで繋げていくことで Hoare Logic によるプログラムの検証とする。

5.2　Invariant を用いた Hoare Logic による検証の方法

図 2.1 を用いて Agda にて Hoare Logic による CodeGear を検証する流れを説明する。input DataGear が Hoare Logic 上の Pre Condition(事前条件) となり、output DataGear が Post Condition となる。各 DataGear が Pre / Post Condition を満たしているかの検証

は、各 Condition を Meta DataGear で定義し、条件を満たしているのかを Meta CodeGear で検証する。

この際、検証する DataGear はプログラムの不変条件となり、これを Invariant と呼ぶ。

5.3　Gears Agda にて Hoare Logic を用いた検証の例

ここでは Gears Agda にて Hoare Logic を用いた検証の例として、While Loop プログラムを実装・検証する。

5.3.1　While Loop の実装

まずは前述した Gears Agda の記述形式に基づいて While Loop を実装する。実装はまず、 ソースコード 5.1 のように CodeGear に遷移させる DataGear の定義から行う。

```
record Env : Set where
  field
    varn : ℕ
    vari : ℕ
open Env
```

ソースコード 5.1: DataGear の定義

そのため最初の CodeGear は引数を受け取り、Env を作成する CodeGear となる ソースコード 5.2。

```
whileTest : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : Env → t) → t
whileTest c10 next = next (record {varn = c10 ; vari = 0} )
```

ソースコード 5.2: DataGear の定義を行う CodeGear

次に、目的である While Loop の実装を行う。ソースコードは coderefwhile-loop のようになる。

```
{-# TERMINATING #-}
whileLoop : {l : Level} {t : Set l} → Env → (Code : Env → t) → t
whileLoop env next with lt 0 (varn env)
whileLoop env next | false = next env
whileLoop env next | true = whileLoop (record {varn = (varn env) - 1 ; vari = (vari
    env) + 1}) next
```

ソースコード 5.3: Loop の部分を担う CodeGears

また Agda では停止性の検出機能が存在し、プログラム中 に停止しない記述が存在するとコンパイル時にエラーが出る。その場合は関数定義の直前に {-# TERMINATING #-} のタグを付けると停止しないプログラムをコンパイルすることができる

これらの CodeGear を繋げることで、 While Loop を行う ソースコード 5.4 を実装することができる。

```
whileLoopC : ℕ → Env
whileLoopC n = whileTest n (λ env → whileLoop env (λ env1 → env1 ))
```

ソースコード 5.4: While Loop を行う CodeGear

5.3.2　While Loop の検証

While Loop の実装コードを元に、前述した Pre / Post Condition を記述していくことで Hoare Logic を用いた検証を行う。

検証を行う CodeGear も Gears Agda による単純な実装と同じく DataGear から行う。ソースコード 5.5 がそれに当たる。

```
whileTest′ : {l : Level} {t : Set l} → {c10 : ℕ } → (Code : (env : Env) → ((vari
    env) ≡ 0) /\ ((varn env) ≡ c10) → t) → t
whileTest′ {_} {_} {c10} next = next env proof2
  where
    env : Env
    env = record {vari = 0 ; varn = c10}
    proof2 : ((vari env) ≡ 0) /\ ((varn env) ≡ c10) -- PostCondition
    proof2 = record {pi1 = refl ; pi2 = refl}
```

ソースコード 5.5: While Loop を行う CodeGear

今回は検証を行いたいため 5.1 で述べたように、実装に加えて Pre / Post Condition を持つ必要がある。init 時の Pre Condition のみ特殊で Agda での関数定義の際に記述し、「DataGear に正しく初期値が設定される」という Invariant を使用する。これが ((vari env) ≡ 0) ∧ ((varn env) ≡ c10) に当たる。そして init 時以外の、Pre Condition と Post Condition には実行開始から実行終了までの間の Invariant を記述する。今回は While Loop の Invariant として、今回 $loop$ させたい回数 $(c10)$ = 残りの $loop$ する回数 $(vern)$ + 今回 $loop$ した回数 $(vari)$ を設定した。これが init 時の Post Condition となる。

また、init 時の Pre Condition と同じ値で Post Condition を設定しなければならない。init 時の Pre Condition を Post Condition に変換する ソースコード 5.6 を記載する。

```
conversion1 : {l : Level} {t : Set l } → (env : Env) → {c10 : ℕ } → ((vari env)
    ≡ 0) /\ ((varn env) ≡ c10)
              → (Code : (env1 : Env) → (varn env1 + vari env1 ≡ c10) → t) → t
conversion1 env {c10} p1 next = next env proof4 where
     proof4 : varn env + vari env ≡ c10
     proof4 = let open ≡-Reasoning in begin
          varn env + vari env ≡⟨ cong ( λ n → n + vari env ) (pi2 p1 ) ⟩
          c10 + vari env ≡⟨ cong ( λ n → c10 + n ) (pi1 p1 ) ⟩
          c10 + 0 ≡⟨ +-sym {c10} {0} ⟩
          c10
        ■
```

ソースコード 5.6: init 時の Pre Condition を Post Condition に変換する CodeGear

ここで変換されて作成された Post Condition はプログラム実行中の Invariant となるため、停止するまでこの Pre / Post Condition を用いる。

以下の ソースコード 5.7 は停止性の検証を行っていないが、 Wile Loop の Loop 部分の検証を行う CodeGear となる。

```
{-# TERMINATING #-}
whileLoop′ : {l : Level} {t : Set l} → (env : Env) → {c10 : ℕ } → ((varn env) + (
    vari env) ≡ c10)
   → (Code : (e1 : Env )→ vari e1 ≡ c10 → t) → t
whileLoop′ env proof next with ( suc zero ≤? (varn env) )
whileLoop′ env {c10} proof next | no p = next env ( begin
      vari env ≡⟨ refl ⟩
      0 + vari env ≡⟨ cong (λ k → k + vari env) (sym (lemma1 p )) ⟩
      varn env + vari env ≡⟨ proof ⟩
      c10 ■ ) where open ≡-Reasoning
whileLoop′ env {c10} proof next | yes p = whileLoop′ env1 (proof3 p ) next where
     env1 = record {varn = (varn env) - 1 ; vari = (vari env) + 1}
     1<0 : 1 ≤ zero → ⊥
     1<0 ()
     proof3 : (suc zero ≤ (varn env)) → varn env1 + vari env1 ≡ c10
     proof3 (s≤s lt) with varn env
     proof3 (s≤s z≤n) | zero = ⊥-elim (1<0 p)
     proof3 (s≤s (z≤n {n′}) ) | suc n = let open ≡-Reasoning in begin
           n′ + (vari env + 1) ≡⟨ cong ( λ z → n′ + z ) ( +-sym {vari env} {1} ) ⟩
           n′ + (1 + vari env ) ≡⟨ sym ( +-assoc (n′) 1 (vari env) ) ⟩
           (n′ + 1) + vari env ≡⟨ cong ( λ z → z + vari env ) +1≡suc ⟩
           (suc n′ ) + vari env ≡⟨⟩
           varn env + vari env ≡⟨ proof ⟩
           c10
        ■
```

ソースコード 5.7: 停止性以外の Loop の検証も行う CodeGear

Loop が停止することを示していないため、関数定義の直前に {-# TERMINATING #-} が記述されている。こちらも Loop の実装以外に、Pre / Post Condition を満たしているか検証を行い、次の CodeGear に渡している。

ここまでで定義した Pre / Post Consition が付いている CodeGear を繋げることで、停止性以外の While Loop の検証を行う CodeGear を実装できる。ソースコード 5.8 のようになる。

```
whileTestSpec1 : (c10 : ℕ) → (e1 : Env ) → vari e1 ≡ c10 → ⊤
whileTestSpec1 _ _ x = tt

proofGears : {c10 : ℕ } → ⊤
proofGears {c10} = whileTest′ {_} {_} {c10} (λ n p1 → conversion1 n p1 (λ n1 p2 →
    whileLoop′ n1 p2 (λ n2 p3 → whileTestSpec1 c10 n2 p3 )))
```

ソースコード 5.8: 停止性以外の While Loop の検証を行う CodeGear

ソースコード 5.9 停止性の検証も行う While Loop の検証を行う CodeGear を実装する

```
1  TerminatingLoopS : {l : Level} {t : Set l} (Index : Set ) → {Invraiant : Index →
       Set } → ( reduce : Index → ℕ)
2    → (loop : (r : Index) → Invraiant r → (next : (r1 : Index) → Invraiant r1 →
         reduce r1 < reduce r → t ) → t)
3    → (r : Index) → (p : Invraiant r) → t
4  TerminatingLoopS {_} {t} Index {Invraiant} reduce loop r p with <-cmp 0 (reduce r)
5  ... | ≈ tri ¬a b ¬c = loop r p (λ r1 p1 lt → ⊥-elim (lemma3 b lt) )
6  ... | tri< a ¬b ¬c = loop r p (λ r1 p1 lt1 → TerminatingLoop1 (reduce r) r r1 (≤-
       step lt1) p1 lt1 ) where
7      TerminatingLoop1 : (j : ℕ) → (r r1 : Index) → reduce r1 < suc j → Invraiant r1
           → reduce r1 < reduce r → t
8      TerminatingLoop1 zero r r1 n≤j p1 lt = loop r1 p1 (λ r2 p1 lt1 → ⊥-elim (lemma5
           n≤j lt1))
9      TerminatingLoop1 (suc j) r r1 n≤j p1 lt with <-cmp (reduce r1) (suc j)
10     ... | tri< a ¬b ¬c = TerminatingLoop1 j r r1 a p1 lt
11     ... | ≈ tri ¬a b ¬c = loop r1 p1 (λ r2 p2 lt1 → TerminatingLoop1 j r1 r2 (subst
           (λ k → reduce r2 < k ) b lt1 ) p2 lt1 )
12     ... | tri> ¬a ¬b c = ⊥-elim ( nat-≤> c n≤j )
```

ソースコード 5.9: 停止性の検証も行う Loop 部分の CodeGear

停止することを Agda が理解できるように記述する。そのため引数に減少する変数 reduce を追加し、loop するとデクリメントするように実装する。

loop には先ほど実装した loop の部分を担う CodeGear が次の関数遷移先を引数に受け取れるようにした whileLoopSeg を使用する。

そしてこれらを繋げて While Loop の検証を行えるソースコード 5.10 を実装できた。

```
1  proofGearsTermS : {c10 : ℕ } → ⊤
2  proofGearsTermS {c10} = whileTest′ {_} {_} {c10} (λ n p → conversion1 n p (λ n1 p1
       →
3      TerminatingLoopS Env (λ env → varn env)
4          (λ n2 p2 loop → whileLoopSeg {_} {_} {c10} n2 p2 loop (λ ne pe →
               whileTestSpec1 c10 ne pe)) n1 p1 ))
```

ソースコード 5.10: 停止性の検証も行う While Loop の CodeGear

5.4　Gears Agda における Binary Tree の検証

ここでは Gears Agda にて再帰的なデータ構造を検証する例として、Binary Tree [14] を実装・検証する。

5.4.1　Gears Agda における木構造の設計

本研究では、Gears Agda にて Binary Tree の検証を行う際に、Agda が変数に対して再代入を許していないことが問題になってくる。

そのため下図 5.1 のように、木構造の root から leaf に 辿る際に見ている node から下の tree をそのまま stack に持つようにする。

そして insert や delete を行った後に stack から tree を取り出し、元の木構造を再構築していきながら root へ戻る。

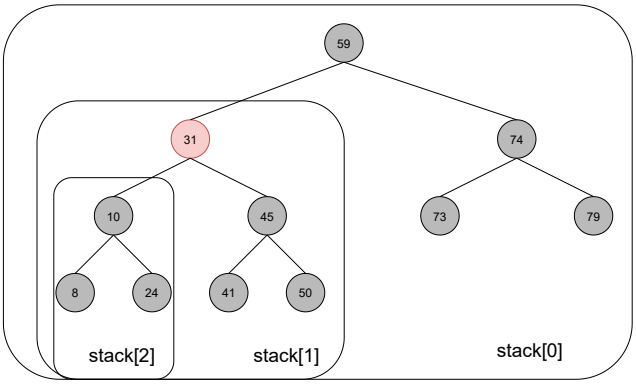

図 5.1: tree を stack して目的の node まで辿った例

このようにして Gears Agda にて Binary Tree を実装していく。

5.4.2　Gears Agda における Binary Tree の実装

Binary Tree と 遷移させる DataGear となる Env の定義は ソースコード 5.11 のようになる。

```
data bt {n : Level} (A : Set n) : Set n where
  leaf : bt A
  node : (key : ℕ) → (value : A) →
    (left : bt A ) → (right : bt A ) → bt A

record Env {n : Level} (A : Set n) : Set n where
  field
    varn : ℕ
    varv : A
    vart : bt A
    varl : List (bt A)
open Env
```

ソースコード 5.11: Binary Tree の DataGear

bt は、木での順序としての意味を持つ key とその中身 value はどのような型でも入れられるように「A : Set n」となっている。そして left, right には bt A を持つようにし、木構造を構築している。

Env では、 find, insert, delete の対象となる値を保存し、 CodeGear に与えられるようにするために varn, varv を持っている。加えて変更を加える bt を持つ vart と、前述した木構造を持っておくための List である varl を Env に設定している。

7 章で述べた Gears Agda での木構造を保ったまま root から目的の node まで辿る CodeGear がソースコード 5.12 になる。

```
find : {n m : Level} {A : Set n} {t : Set m} → (env : Env A )
          → (next : (env : Env A ) → t )
          → (exit : (env : Env A ) → t ) → t
find env next exit = find-c (vart env) env next exit where
    find-c : {n m : Level} {A : Set n} {t : Set m} → (tree : bt A) → (env : Env A )
          → (next : (env : Env A ) → t )
          → (exit : (env : Env A ) → t ) → t
    find-c leaf env next exit = exit env
    find-c n@(node key-t value ltree rtree) env next exit with <-cmp (varn env) key-t
    ... | tri< a ¬b ¬c = find-c ltree record env {vart = ltree ; varl = (n :: (varl
        env))} next exit
    ... | ≈ tri ¬a b ¬c = exit record env {vart = n}
    ... | tri> ¬a ¬b c = find-c rtree record env {vart = rtree ; varl = (n :: (varl
        env))} next exit
```

ソースコード 5.12: root から目的の node まで辿る CodeGear

まず、関数の実装が始まってすぐに Env の vart を指定したものと引数をそのまま find-c の関数に遷移している。ここで展開しているのは Env の vart で、そのまま Env から展開した vart をパターンマッチすると Agda が追えなくなってしまい、{-# TERMINATING #-} を使用することになってしまう。

そのため関数を新たに定義し、展開したものを受け取り、パターンマッチすることで {-# TERMINATING #-} を使用せずに loop を定義できるようになる。

木を stack に入れるのは単純で、操作の対象の key となる varn と node の key を比較を行う。その後、本来の木構造と同じで、操作の対象の key が小さいなら left の tree を次の node として遷移する。大きいなら right の tree を次の node として遷移していく。

操作の対象となる node に辿り着き、操作を行った後、Stack に持っている tree から再構築を行う。

そのコードが ソースコード 5.13 となる。

```
replace : {n m : Level} {A : Set n} {t : Set m} → (env : Env A )
    → (next : Env A → t )
    → (exit : Env A → t) → t
replace env next exit = replace-c (varl env) env next exit where
    replace-c : {n m : Level} {A : Set n} {t : Set m} → List (bt A) → (env : Env A
        )
        → (next : Env A → t )
        → (exit : Env A → t) → t
    replace-c st env next exit with st
    ... | [] = exit env
    ... | leaf :: st1 = replace-c st1 env next exit
    ... | n@(node key-t value ltree rtree) :: st1 with <-cmp (varn env) (key-t)
```

```
12    ... | tri< a ¬b ¬c = replace-c st1 record env{vart = (node key-t value (vart env
          ) rtree); varl = st1} next exit
13    ... | ≈ tri ¬a b ¬c = replace-c st1 record env{vart = (node key-t (varv env)
          ltree rtree); varl = st1} next exit
14    ... | tri> ¬a ¬b c = replace-c st1 record env{vart = (node key-t value ltree (
          vart env)); varl = st1} next exit
```

ソースコード 5.13: Stack から tree を再構築する CodeGear

これも ソースコード 5.12 と同じように構成されており、varn と node の key を比較し、vart を List から持ってきた node の どこに加えるかを決めるようになっている。

以上の流れを繋げることで、Binary Tree の insert と find を実装できた。delete は insert の値を消すようにすると実装ができる。

5.4.3 Gears Agda における Binary Tree の検証

検証も前述した While Loop の 検証と同じようにしていく。しかし、 Binary Tree の Invariant は 2 つ以上あるため、これを関数定義の際に全て書くと煩雑になってしまうため、事前に記述して関数化しておく。それがソースコード 5.14 になる。

```
1  treeInvariant : {n : Level} {A : Set n} → (tree : bt A) → Set
2  treeInvariant leaf = ⊤
3  treeInvariant (node key value leaf leaf) = ⊤
4  treeInvariant (node key value leaf n@(node key1 value1 t1 t2)) = (key < key1) ∧
       treeInvariant n
5  treeInvariant (node key value n@(node key1 value1 t t1) leaf) = treeInvariant n ∧ (
       key < key1)
6  treeInvariant (node key value n@(node key1 value1 t t1) m@(node key2 value2 t2 t3))
       = treeInvariant n ∧ (key < key1) ∧ (key1 < key2) ∧ treeInvariant m
7
8  stackInvariant : {n : Level} {A : Set n} → (tree : bt A) → (stack : List (bt A))
       → Set n
9  stackInvariant {_} {A} _ [] = Lift _ ⊤
10 stackInvariant {_} {A} tree (tree1 :: [] ) = tree1 ≡ tree
11 stackInvariant {_} {A} tree (x :: tail @ (node key value leaf right :: _) ) = (right
       ≡ x) ∧ stackInvariant tree tail
12 stackInvariant {_} {A} tree (x :: tail @ (node key value left leaf :: _) ) = (left
       ≡ x) ∧ stackInvariant tree tail
13 stackInvariant {_} {A} tree (x :: tail @ (node key value left right :: _ )) = ( (
       left ≡ x) ∧ stackInvariant tree tail) ∨
       ( (right ≡ x) ∧ stackInvariant tree tail)
14 stackInvariant {_} {A} tree s = Lift _ ⊥
```

ソースコード 5.14: Binary Tree の Invariant

この Invariant は、treeInvariant が tree の 左にある node の key が小さく、右にある node の方が大きいことを条件としている。

stackInvariant は Stack にある tree が、次に取り出す Tree の一部であることを条件としている。

これを先ほど実装した CodeGear に対して加えることで検証していく。先ほど実装した ソースコード 5.12 に対して加えると ソースコード 5.15 のようになる。

```
1  findP : {n m : Level} {A : Set n} {t : Set m} → (env : Env A)
2            → treeInvariant env ∧ stackInvariant env
3            → (exit : (env : Env A) → treeInvariant env ∧ stackInvariant env → t )
                 → t
4  findP env Cond exit = findP-c (vart env) env Cond exit where
5     findP-c : {n m : Level} {A : Set n} {t : Set m} → (tree : bt A) → (env : Env A)
6           → treeInvariant env ∧ stackInvariant env
7           → (exit : (env : Env A) → treeInvariant env ∧ stackInvariant env → t )
                 → t
8     findP-c leaf env Cond exit = exit env Cond
9     findP-c n@(node key-t value ltree rtree) env Cond exit with <-cmp key-t (varn env
          )
10    ... | tri< a ¬b ¬c = findP-c ltree record env {vart = ltree ; varl = (n :: (varl
           env))} {} exit
11    ... | ≈ tri ¬a b ¬c = exit record env {vart = n} {}
12    ... | tri> ¬a ¬b c = findP-c rtree record env {vart = rtree ; varl = (n :: (varl
           env))} {} exit
```

ソースコード 5.15: Binary Tree の検証

現時点では条件を満たしていることの証明まで行っていないがコード中の !! に記述を行い、前述した While Loop と同じように中身を記述することで検証を行える。

第6章　Gears Agda によるモデル検査

定理証明では数学的な証明をするため状態爆発を起こさず検証を行うことができる。しかし、専門的な知識が多く難易度が高いという欠点がある。加えて、CbC では並列処理も実行できる [15] が、並列処理を定理証明するには検証する状態が膨大になり困難である。そのため、並列処理はモデル検査にて検証する方が良い。

6.1　モデル検査とは

モデル検査 (Model Cheking) とは、検証手法のひとつである。これはプログラムが入力に対して仕様を満たした動作を行うことを網羅的に検証することを指す。しかし、モデル検査と定理証明を比較した際に、モデル検査は入力が無限になる状態爆発が起こり得るという問題がある。実際にモデル検査で行うことは、検証したい内容の時相理論式 φ を作り、対象のシステムの初期状態 s のモデル M があるとき、M, s が φ を満たすかを調べる。

6.2　Dining Philosophers Problem

今回はモデル検査を行う対象として Dining Philosophers Problem (以下 DPP) を用いることとした。DPP とは資源共有問題であり、モデル検査をする際に挙げられる代表的な問題である。

DPP のストーリーを図 6.1 に示している。

したがって、哲学者は以下のようなフローを独立して並列に繰り返し実行することとなる。

1. しばらくの間思考を行う
2. 食事をするために右のフォークを取る
3. 右のフォークを取ったら、次は左のフォークを取る
4. 両方のフォークを取ったら、しばらく食事をする

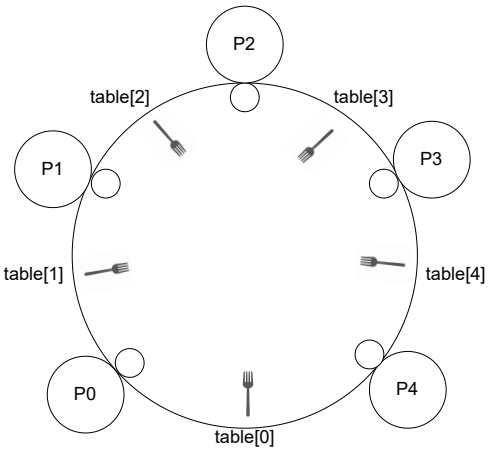

図 6.1: Dining Philosophers Program のイメージ図

5. 思考に戻るために左手に持っているフォークをテーブルに置く

6. 左手のフォークを置いたあとは右のフォークをテーブルに置く

この際、すべての哲学者が同時に右のフォークを取った場合のことを考える。すべての哲学者はフォークを持っている。次に哲学者は左のフォークを取ろうとする。しかしフォークは哲学者の人数と同じ数だけ存在しているため、テーブルの上にフォークはすでにひとつも存在しない。すべての哲学者は左のフォークを取ろうとするが誰も右のフォークを置くことがないため、すべての哲学者の動作がこの状態で止まる。(dead lock) これが起こることを Gears Agda で検出したい。

6.3　SPIN によるモデル検査

SPIN(Simple Promela INterpreter) [16] とは一般的にモデル検査に使用されるツールである。これは使用記述言語 PROMELA(Process Meta Language) による記述を元にプログラムが取る状態を網羅し、モデル検査を行うことができる。

今回 Gears Agda でのモデル検査と比較するために、SPIN での DPP プログラムのモデル検査に必要なコードの一部をソースコード 6.1 に載せる。

```
1 proctype Philosopher(byte id) {
2 Think:
3  if
4  :: atomic { fork[id] == 0 -> fork[id] = id + 1; };
5  :: atomic { fork[(id + 1)%N] == 0 -> fork[(id + 1)%N] = id + 1; };
6  fi;
7 One:
```

```
8  if
9  :: atomic { fork[id] == id + 1 -> fork[(id + 1)%N] == 0 -> fork[(id + 1)%N] = id +
       1; nr_eat++; };
10 :: atomic { fork[id] == 0 -> fork[(id + 1)%N] == id + 1 -> fork[id] = id + 1;
       nr_eat++; };
11 fi;
12 Eat:
13 d_step { nr_eat--; fork[(id + 1)%N] = 0; }
14 fork[id] = 0;
15 goto Think;
16 }
```

ソースコード 6.1: SPIN にて DPP をモデル検査する際のコードの一部

コードを簡単に説明する。哲学者が Think の状態から食事をしようとしだした際の状態の変化が 4 行目と 5 行目になっている。fork の状態を配列で管理している。0 である状態が誰もその fork を持っていない状態である。ここでは、5 人目の人が右手にあるフォークを取ろうとした際に配列の最初を取ることが 5 行目に記述されている。

左手のフォークを取る動作も同じように記述する。SPIN ではこのコードを元にプログラムが取りうる状態全てを網羅し、モデル検査を行うことができる。

図 6.2 はこの Promela から作成された状態遷移図になる。SPIN ではコードからプログラムの状態遷移図を出力することができる他、プログラムの step 実行など幅広くモデル検査を行うことができる。

6.4　Gears Agda によるモデル検査の流れ

今回作成した Gears Agda による DPP のモデル検査の流れは以下のようになっている。

1. Gears Agda にて DPP の実装を行う
2. プログラムが実行中に取りうる状態の網羅をする State List を作成する
3. 上で実装したものを使用しつつ Meta DataGear を構築する Meta CodeGear を記述する
4. 上記 Meta CodeGear で作成された meta data を元に、dead lock を判定する Meta CodeGear を記述する
5. 状態を網羅した State List にある State を一つずつ Meta CodeGear に実行させて、dead lock している状態がないか確認する

先行研究では網羅した状態データを State DB に格納していた。しかし今回の Gears Agda では State List にて代替とした。これは List で行った方が Agda での実装がしやすい点にある。本来は branced tree をデータ構造に持って State を作成するほう正しい。

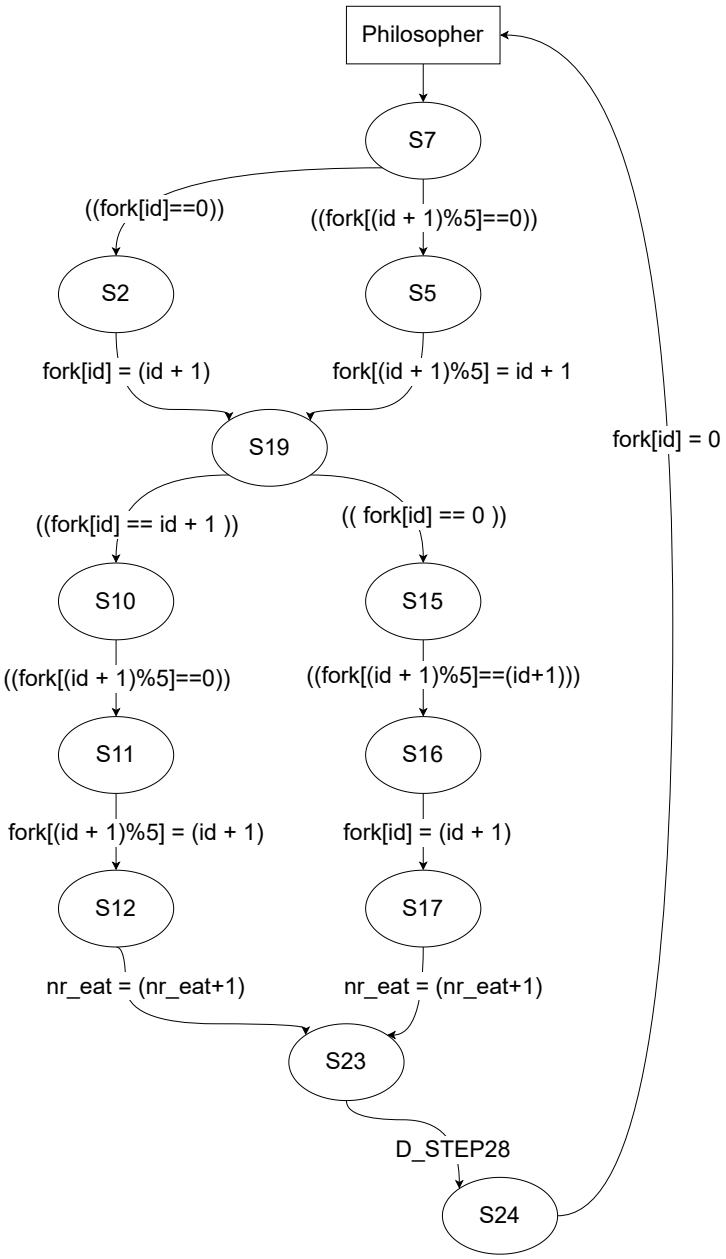

図 6.2: SPIN にて作成した状態遷移図

dead lock の定義として、全てのプロセスが他のプロセスの実行終了待ちをしている。かつ新たに状態の分岐が作成できないものとした。そこで step 実行してもプロセスがすべて変動がなく、かつ State の分岐が作成されなかったものを dead lock としている。

最後について、今回は状態の網羅を一度終了してから dead lock の有無を検証している。しかし、これは今回のプログラムが有限オートマトンであることは事前に明らかであるためにできたことである。検証したいプログラムが無限の状態を作成する場合は、State を作成するたびにそれに対して dead lock や live lock などのモデル検査をすることもできる。モデル検査中に意図しない動きがあった場合に停止するようにすることで、無限の状態を作成するプログラムであっても、モデル検査ができると考える。他にも、プログラムが取る状態を制限することで有限な状態を作成するようにし、モデル検査をすることもできる。 (bounded model checking)

6.5　Gears Agda による DPP の実装

DPP の記述の主要部分を示し、説明する。

```
1 data Code : Set where
2    C_putdown_rfork : Code
3    C_putdown_lfork : Code
4    C_thinking : Code
5    C_pickup_rfork : Code
6    C_pickup_lfork : Code
7    C_eating : Code
```

ソースコード 6.2: Gears Agda での DPP の 哲学者の状態

```
1 record Phi : Set where
2   field
3     pid : ℕ
4     right-hand : Bool
5     left-hand : Bool
6     next-code : Code
7 open Phi
```

ソースコード 6.3: Gears Agda での DPP の プロセス

```
1 record Env : Set where
2   field
3     table : List ℕ
4     ph : List Phi
5 open Env
```

ソースコード 6.4: Gears Agda での DPP の DataGear

ソースコード 6.2 は前述した哲学者の状態を書き記して、哲学者が今行おうとしている動作を網羅している。

ソースコード 6.3 は哲学者一人ずつの環境を持っている。pid はその哲学者がどこに座っているかの識別子で、right / left hand はフォークを手に持っているかを格納している。next-code は次に行う動作を格納している。

ソースコード 6.4 が DataGear になる。ph は前もって定義した一人の哲学者のプロセスの List になる。List になっている理由は、哲学者が複数人いるためである。そのため実行時に List から一人ずつ取り出して実行をしていく。

table はテーブルに置いてあるフォークの状態のことで、pid が 1 の人の右側にあるフォークが List の最初にあり、pid が 1 の人の左側にあるフォーク、すなわち pid が 2 の人の右側にあるフォークがその次の List に格納されていくようになっている。また、自然数の List になっているが、その場所のフォークがテーブルの上にある場合は自然数の 0 が、誰かが所持している場合はその人の pid が格納されるようになっている。

```
1 init-table : {n : Level} {t : Set n} → ℕ → (exit : Env → t) → t
2 init-table n exit = init-table-loop n 0 (record {table = [] ; ph = []}) exit where
3   init-table-loop : {n : Level} {t : Set n} → (redu inc : ℕ) → Env → (exit : Env
        → t) → t
4   init-table-loop zero ind env exit = exit env
5   init-table-loop (suc redu) ind env exit = init-table-loop redu (suc ind) record env
        {
6     table = 0 :: (table env)
7     ; ph = record {pid = redu ; left-hand = false ; right-hand = false ; next-code =
          C_thinking } :: (ph env) } exit
```

ソースコード 6.5: Gears Agda での DPP の DataGear の init

ソースコード 6.5 が入力から DataGear を作成する CodeGear になる。ここでは哲学者の人数を自然数で受け取り、人数分の List Phi と table を一つずつ作成し env を作成している。また、最初の哲学者の状態は思考することであるため、next-code には C_thinking を格納している。

```
1 code_table : {n : Level} {t : Set n} → Code → ℕ → Phi → Env → (Env → t) → t
2 code_table C_putdown_rfork = putdown-rfork-c
3 code_table C_putdown_lfork = putdown-lfork-c
4 code_table C_thinking = thinking-c
5 code_table C_pickup_rfork = pickup-rfork-c
6 code_table C_pickup_lfork = pickup-lfork-c
7 code_table C_eating = thinking-c
```

ソースコード 6.6: Gears Agda での DPP の step 実行

Agda では並列実行を行うことができない。そのため step 単位の実行を一つずつ行うことで並列実行をしていることとする。

この際に Env にある List Phi の中身を展開しながら一つずつ行動を処理していく。

```
1 pickup-lfork-c : {n : Level} {t : Set n} → ℕ → Phi → Env → (Env → t) → t
2 pickup-lfork-c ind p env exit = pickup-lfork-p (suc ind) [] (table env) p env exit
      where
3   pickup-lfork-p : {n : Level} {t : Set n} → ℕ → (f b : List ℕ) → Phi → Env → (
        Env → t) → t
```

```
4   pickup-lfork-p zero f [] p env exit with table env
5   ... | [] = exit env
6   ... | 0 :: ts = exit record env{ph = ((ph env) ++ (record p{left-hand = true ; next
        -code = C_eating} :: [])); table = ((pid p) :: ts)}
7   ... | (suc x) :: ts = exit record env{ph = ((ph env) ++ p :: [])}
8   pickup-lfork-p zero f (0 :: ts) p env exit = exit record env{ph = ((ph env) ++ (
        record p{left-hand = true ; next-code = C_eating} :: [])); table = (f ++ ((pid
        p) :: ts))}
9   pickup-lfork-p zero f ((suc x) :: ts) p env exit = exit record env{ph = ((ph env)
        ++ p :: [])}
10  pickup-lfork-p (suc ind) f [] p env exit = exit env
11  pickup-lfork-p (suc ind) f (x :: ts) p env exit = pickup-lfork-p ind (f ++ (x ::
        [])) ts p env exit
```

ソースコード 6.7: Gears Agda での DPP の左のフォークを取る記述

ソースコード 6.7 が step 実行をした際に哲学者が左側のフォークを取る記述になる。

左側のフォークを取る際には table の index は pid の次の値になっている。図 6.1 を見ると直感的に理解ができる。

最後の哲学者は一番最初のフォークを参照する必要がある。フォークの状態を確認し、値が 0 である場合はフォークがテーブルの上にあるのでそれを自分の pid に書き換える。次に left-hand を true に書き換えて手に持つフォークの状態が 0 以外、すなわち他の哲学者がその場所のフォークを取得している場合は状態を変化させずに処理を続ける。このように左のフォークを取る記述をした。

右側のフォークを取る場合は引数の部分を 1 足さずにそのまま受け取る。比較するべき table の List と哲学者の List は一致しているため、pickup_lfork のように最後の哲学者が最初のフォークを参照することもない。

似たような形で哲学者がフォークを置く putdown-lfork/rfork を実装した。思考と食事の実装に関してはそのまま状態を変更することなく進むようにしている。

6.6　Gears Agda による DPP の検証

これまでの実装は一般的な DPP の実装であったため、Code / DataGear の実装であった。ここからは、モデル検査を行うため、Meta DataGear の定義をし、それの操作を行う Meta CodeGear の実装を行っていく。

以下ソースコード 6.8 が Meta DataGear の定義になる。

```
1 record metadata : Set where
2   field
3     num-branch : ℕ
4     wait-list : List ℕ
5 open metadata
6
7 record MetaEnv : Set where
8   field
9     DG : List Env
```

```
10    meta : metadata
11    deadlock : Bool
12    is-done : Bool
13    is-step : Bool
14 open MetaEnv
15
16 record MetaEnv2 : Set where
17   field
18    DG : List (List Env)
19    metal : List MetaEnv
20 open MetaEnv2
```

ソースコード 6.8: Gears Agda で DPP のモデル検査を行うための Meta DataGear

この Meta DataGear の説明をすると、metadata は状態の分岐数を持っておく num-brunch がある。MetaEnv はもとの DataGear を保持する DG(DataGear の省略) となる。meta には前述した metadata を持っており、他には、deadlock しているかのフラグである deadlock 、最後の 2 つは後に必要になるフラグである。その state が step 実行済みなのかのフラグである is-step、その state がモデル検査済なのかのフラグである is-done フラグを持っている。

MetaEnv2 は 1 つの state である MetaEnv の List を metal で持てる。加えて今まで実行していた DataGear を DG で持てる。

次に Meta DataGear を作成する Meta CodeGear の説明をする。

1. その状態から分岐できる状態数をカウントする Meta CodeGear
2. Wait List を作成する Meta CodeGear

以下の ソースコード 6.9 が分岐できる状態数をカウントする Meta CodeGear となる。

```
1 check-deadlock-metaenv : {n : Level} {t : Set n} → MetaEnv2 → (exit : MetaEnv2 → t
      ) → t
2 check-deadlock-metaenv meta2 exit = search-brute-force-envll-p [] (metal meta2) (λ e
      → exit record meta2{metal = e}) where
3   search-brute-force-envll-p : {n : Level} {t : Set n} → (f b : List (MetaEnv)) → (
        exit : List (MetaEnv) → t) → t
4   search-brute-force-envll-p f [] exit = exit f
5   search-brute-force-envll-p f b@(metaenv :: bs) exit with DG metaenv
6   ... | [] = search-brute-force-envll-p f bs exit
7   ... | (env :: envs) = brute-force-search env (λ e0 → search-brute-force-envll-p (f
         ++ ( record metaenv{meta = record (meta metaenv){num-branch = (length e0)} }
        :: [])) bs exit )
```

ソースコード 6.9: 状態の分岐数をカウントする Meta DataGear の定義

実際にやっていることは、MetaEnv2 から state を取り出し、分岐を見る関数に遷移させている。その結果の List の length を meta データとしている。

つまり、この Meta CodeGear の実装にあたって新しい実装はほとんど行っていない。

Wait List の 作成も同じように取り出した state を step させて、そこで一致する next-code を状態が変わっていないとして Wait List に入れている。

Wait List について、Thinking と Eathing の状態に関しては状態が変わる可能性がある。これを Wait List に入れなければ Wait List のみで dead lock が検知できると考えられる。しかし、DPP 以外の他のプログラムをモデル検査する際に、一つ一つ next-code の設定を行うのは煩雑であると考えた。そのため、step した際に状態が変化しないものを Wait List にいれた。これと分岐がない場合という条件にて、dead lock を検知する。これにより Meta CodeGear の作成が簡易化された。そのため、Thinking と Eathing のプロセスも Waithing List に入るようになっている。

deadlock の検出方法として、上記の 2 つの Meta CodeGear で作成した meta データを使用する。

そして「num-brunch の値が 1 であり、wait List の数がプロセス数と一致する」ということは、「その状態から別の状態に遷移することができない」として、この状態を dead lock であると定義した。

以下のソースコード 6.10 が dead-lock を検知する関数となる。

複雑なことは何もしておらず、単純に state 毎の num-brunch の値を見て、wait-list の数がプロセス数と一致していた場合に deadlock のフラグを立ち上げている。これが、Gears Agda におけるアサーションになっている。

```
judge-deadlock-metaenv : {n : Level} {t : Set n} → MetaEnv2 → (exit : MetaEnv2 → t
    ) → t
judge-deadlock-metaenv meta2 exit = judge-deadlock-p [] (metal meta2) (λ e → exit
    record meta2{metal = e} ) where
  judge-deadlock-p : {n : Level} {t : Set n} → (f b : List (MetaEnv)) → (exit :
      List (MetaEnv) → t) → t
  judge-deadlock-p f [] exit = exit f
  judge-deadlock-p f (metaenv :: bs) exit with num-branch (meta metaenv)
  ... | suc (suc branchnum) = judge-deadlock-p (f ++ (metaenv :: []) ) bs exit
  ... | zero = judge-deadlock-p (f ++ (metaenv :: []) ) bs exit
  ... | suc zero with (DG metaenv )
  ... | [] = judge-deadlock-p (f ++ (metaenv :: []) ) bs exit
  ... | p :: ps with <-cmp (length (wait-list (meta metaenv))) (length (ph p))
  ... | tri< a ¬b ¬c = judge-deadlock-p (f ++ (metaenv :: []) ) bs exit
  ... | tri> ¬a ¬b c = judge-deadlock-p (f ++ (metaenv :: []) ) bs exit
  ... | ≈ tri ¬a b ¬c = judge-deadlock-p (f ++ (record metaenv{deadlock = true} ::
      []) ) bs exit
```

ソースコード 6.10: DPP での dead lock を検知する Meta CodeGear

ここから前述した通り、State List を作成する。

そのまま実行するだけでは無限に実行されるのみで停止しないと思われる。しかし、分岐後に step 実行後の state を保存している State List に同じ State が存在しないかを確かめる。存在していた場合はそれを追加せず、存在しなかった場合にのみ State を追加する。

Agda には 2 つの record が等しいか確かめる関数などは存在せず、ソースコード 6.11 のように record の中身を一つずつ一致しているか確認する。こちらはそのまま掲載すると長いので、実際のコードに手を加えて省略している。実際のコードは付録にて参照で

きる。

```
1  exclude-same-env : {n : Level} {t : Set n} → (env1 env2 : MetaEnv2) → (exit :
       MetaEnv2 → t) → t
2  exclude-same-env env1 env2 exit = loop-target-metaenv (metal env1) env2 exit where
3    eq-pid : {n : Level} {t : Set n} → MetaEnv2 → (phl1 phl2 : List Phi) → (p1 p2 :
         Phi) → (exit : MetaEnv2 → t) → (loop : Bool → t) → t
4    eq-lhand : {n : Level} {t : Set n} → MetaEnv2 → (phl1 phl2 : List Phi) → (p1 p2
         : Phi) → (exit : MetaEnv2 → t ) → ( loop : Bool → t) → t
5    eq-rhand : {n : Level} {t : Set n} → MetaEnv2 → (phl1 phl2 : List Phi) → (p1 p2
         : Phi) → (exit : MetaEnv2 → t ) → ( loop : Bool → t) → t
6    eq-next-code : {n : Level} {t : Set n} → MetaEnv2 → (phl1 phl2 : List Phi) → (p1
          p2 : Phi) → (exit : MetaEnv2 → t ) → ( loop : Bool → t) → t
7    eq-env : {n : Level} {t : Set n} → MetaEnv2 → ( e1 e2 : List Phi) → (exit :
         MetaEnv2 → t) → (loop : Bool → t) → t
8
9    loop-metaenv : {n : Level} {t : Set n} → MetaEnv → (origin : MetaEnv2) → (f b :
         List MetaEnv) → (exit : MetaEnv2 → t) → t
10
11   loop-target-metaenv : {n : Level} {t : Set n} → (env1 : List MetaEnv) → ( env2 :
         MetaEnv2) → (exit : MetaEnv2 → t) → t
12   loop-target-metaenv [] metaenv2 exit = exit metaenv2
13   loop-target-metaenv (metaenv :: metaenvl) metaenv2 exit = loop-metaenv metaenv
         metaenv2 [] (metal metaenv2) (λ e → loop-target-metaenv metaenvl e exit)
14
15   loop-metaenv me origin f [] exit = exit (record origin{metal = me :: (metal origin)
          }) -- not found & add state to origin
16   loop-metaenv me origin f (x :: metaenvl) exit with DG me
17   ... | [] = exit origin
18   ... | env1 :: envl with DG x
19   ... | [] = loop-metaenv me origin (f ++ (x :: [])) metaenvl exit
20   ... | env2 :: history = eq-env origin (ph env1) (ph env2) (λ exit-e → exit record
         origin{metal = f ++ (record x{is-done = boolor (is-done me) (is-done x); is-
         step = boolor (is-step me) (is-step x) } :: metaenvl)}) (λ e → loop-metaenv me
          origin (f ++ (x :: [])) metaenvl exit )
21
22   eq-env origin [] [] exit loop = exit origin -- true
23   eq-env origin [] (x :: pl2) exit loop = loop false
24   eq-env origin (p1 :: pl1) [] exit loop = loop false
25   eq-env origin (p1 :: pl1) (p2 :: pl2) exit loop = eq-pid origin pl1 pl2 p1 p2 exit
         (λ e → loop e) -- prototype
26
27   eq-pid origin pl1 pl2 p1 p2 next1 exit1 with <-cmp (pid p1) (pid p2)
28   ... | tri< a ¬b ¬c = exit1 false
29   ... | tri> ¬a ¬b c = exit1 false
30   ... | ≈ tri ¬a b ¬c = eq-lhand origin pl1 pl2 p1 p2 next1 exit1
31   .
32   .
33   .
```

ソースコード 6.11: 重複している state を除外する Meta CodeGear

MetaEnv2 を受け取ってその中身の state を比較するが、そこまで展開する必要がある。loop-metaenv と loop-target-metaenv、 eq-env にてそれを行っている。

更に 15 行目の loop-metaenv では State List 内に見つからなかった場合に State List に State を追加し、次の state の一致を探索するように記述されている。

実際に state の一致を見ているのは 22 行目の eq-env 関数で、一致している State が見

つかった場合には追加せずにこちらも次の State を探索するように記述されている。

State が保持している値がそれぞれ正しいのかは eq-pid のように一致を見て分岐させている。その値が一致している場合には別の値を見て、一致していない場合は eq-env に遷移して State List にある次の State との一致を見るようにしている。他の値である、lhand や rhand なども eq-pid と同じように記述している。

これらにて、まだ分岐を見ていない 1 つの State の分岐を確かめる。発生した分岐を step 実行させる。step 実行して作成された state が State List に存在していないかを確かめる。これを繰り返すことで State List を作る。State List に存在している全ての state の分岐を見たということは、出現する State を全て網羅することができたと言える。

あとは State List を dead lock の検査を行う Meta CodeGear に与えるとどの state が dead lock しているかを検証することができる。

6.7　Gears Agda による live lock の検証方法について

live lock とは、状態が変動しているが、その状態はループしており、実行が進んでいないことを指す。

DPP にて例を上げると、(上の哲学者のフローだと) dead lock が発生するため、フローを変更したとする。それは以下のような動作を追加する。「右手にフォークを持っているが、左手のフォークがすでに取られている場合に右手のフォークを即座に置いて固定時間待つ」とする。これで解決すると思われるが、全員が同時に食事をしようとした場合、右手のフォークを取ったり置いたりを永遠に繰り返すことになる。これを live lock という

つまり、この状態を検知するなら、その state が持っている history まで考慮する必要がある。つまり、今回の場合だと state の loop ができた際にコマンドが一巡しているかを確認すると良い。

history が loop していた際に、誰も eathing していない場合を live lock していると定義する。

今回のは DPP であるため Eating が挙げられているが、他の実装であれば、どのコマンドが実行されるのが正常な動きなのかを決める。それが loop 中に存在しているかを確認することで live lock を検知できる。

6.8　Gears Agda でのモデル検査の評価

SPIN で行った DPP のモデル検査に比べると、Gears Agda の方がコードも長く、制約が多いため難解に思える。しかし、Gears Agda ではプログラムの実装も含んでいる。

さらに、Gears Agda での実装をしたあとのモデル検査を行う際の Meta CodeGear の流れは変化しないため、他のプログラムに適応できることを考えると比較的容易であると言える。

加えて、Gears Agda は CbC へコンパイルされ、高速に実行されることを踏まえると、いかに Gears Agda が検証に向いたプログラムであるか理解できる。

第7章　まとめと今後の展望

本論文では Gears Agda による形式手法を用いたプログラムの検証について述べた。そこで、定理証明については Invariant を用いて定理証明を行った。モデル検査については Gears Agda にて dead lock を検知できるようになった。

実際に Invariant を用いることで、プログラムに与える入力とその出力に対して条件を与え、Hoare Logic による検証を行えるようになった。これにより、いままでより容易に Gears Agda にて検証が進められるようになった。

また、先行研究での課題にて、CbC で開発、検証を行いたいと考えている Gears OS[17] の並列動作の検証があった。これもモデル検査により、Gears Agda 内で並列動作に対する検証も行えるようになった。

7.1　今後の課題

今後の課題としてモデル検査による検証、定理証明による検証、Gears Agda の今後の展望について述べる。モデル検査においては、有向グラフからなる有限オートマトンの遷移を全自動探索することと、live lock や LTTL も用いたアサーションなどの検証 [18] を行いたい。加えて、State List のデータ構造を balanced tree にすることで、並列にモデル検査が行えるようになると考えられる。これにより、状態が膨大になるモデル検査に対応できる。

定理証明においては、Red Black Tree の検証を行いたいと思っている。これで検証を行ったものをモデル検査や Gears OS のファイルシステムなどに転用できると考えている。

Gears Agda の展望について、CbC に変換することが挙げられる。CbC はアセンブラに近いため記述が複雑かつ困難であるが、Gears Agda で記述したものを CbC に変換できるようになれば、その点を解決できる。更に、Gears Agda で検証を行ったプログラムであるため、プログラムの信頼性もある。加えて、モデル検査の実行には速度が求められるが、CbC で高速に実行できるようになることが期待される。

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と学科システムを保守してくれたシステム管理チームに感謝致します。私の研究において苦楽を共にした Manjaro Linux とその開発者には頭が上がりません。さらに、ゼミの前においしいご飯で元気づけてくれた Hally's Cafe に感謝しております。最後に、有意義な時間を共に過ごした理工学研究科工学専攻の学友、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。

2023 年 3 月
上地 悠斗

参考文献

[1] cbc-gcc - 並列信頼研 mercurial repository. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/. Accessed: 2020/2/9(Sun).

[2] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pp. 1–2, New York, NY, USA, 2009. ACM.

[3] 政尊外間, 真治河野. Gearsos の agda による記述と検証. Technical Report 5, 琉球大学大学院理工学研究科情報工学専攻, 琉球大学工学部情報工学科, may 2018.

[4] Edmund M. Clarke, Jr. In *Model Checking, Second Edition*, 2018.

[5] 東恩納琢偉. Gears os でモデル検査を実現する方法について. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2021.

[6] Tokumori Kaito and Kono Shinji. Implementing continuation based language in llvm and clang. *LOLA 2015, Kyoto*, July 2015.

[7] The agda wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[8] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.

[9] agda-stdlib. https://github.com/agda/agda-stdlib. Accessed: 2023/2/1(Wed).

[10] 比嘉健太. メタ計算を用いた continuation based c の検証手法. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2017.

[11] Hoare logic in agda2. https://github.com/IKEGAMIDaisuke/HoareLogic. Accessed: 2020/2/9(Sun).

[12] Hoare logic - 並列信頼研 mercurial repository. http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/. Accessed: 2020/2/9(Sun).

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, Vol. 12, No. 10, p. 576–580, October 1969.

[14] 渡邉. データ構造と基本アルゴリズム, 2000.

[15] 伊波立樹. Gears os の並列処理. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2018.

[16] Spin - formal verification. http://spinroot.com/. Accessed: 2023/2/1(Wed).

[17] 宮城光希, 河野真治. Codegear と datagear を持つ gears os の設計. 第 59 回プログラミング・シンポジウム予稿集, 第 2018 巻, pp. 197–206, jan 2018.

[18] 広瀬健. コンピュータ基礎理論ハンドブック 形式的モデルと意味論, 1994.