平成30年度 卒業論文

Agda と継続を用いたプログラムの検証

琉球大学工学部情報工学科

145750B 外間 政尊

指導教員 河野 真治

目 次

図 目 次

ソースコード目次

第1章　ソフトウェアの信頼性の保証

動作するソフトウェアは高い信頼性を持つことが望ましい。そのためにはソフトウェアが期待される動作をすることを保証する必要がある。期待される動作は仕様と呼ばれ、自然言語や論理によって記述される。

ソフトウェアの検証手法にはモデル検査と定理証明がある。モデル検査とはソフトウェアの全ての状態を数え上げ、その状態について仕様が正しいことを確認する。モデル検査器には Spin [1] や、モデルを状態遷移系で記述する NuSMV [2]、C 言語/C++ を記号実行する CBMC [3] などが存在する。

定理証明では。ソフトウェアが満たすべき仕様を論理式で記述し、その論理式が常に正しいことを証明する。定理証明支援器には依存型で証明を行なう Agda [4] や Coq [5] などが存在する。

当研究室では検証の単位として CodeGear、DataGear という単位を用いてソフトウェアを記述する手法を提案している。また、 CodeGear 、 DataGear という単位を用いてプログラミングする言語として Countinuation based C [6] を開発している。Continuation based C (CbC) は C 言語と似た構文を持つ言語である。

本論文では Agda で使ってその仕様の一部を証明した。

第2章　Continuation based C

Continuation based C (以下 CbC) は当研究室で開発しているプログラミング言語である。

CbC では処理を CodeGear 、処理に使うデータを DataGear と呼ばれる単位で記述しプログラムを構成する。

本章は CbC について説明する。

2.1　CodeGear と DataGear

本研究室では検証しやすいプログラムの単位として CodeGear と DataGear という単位を用いるプログラミングスタイルを提案している。

CodeGear　とはプログラムの処理部分である。

DataGear は CodeGear で扱うデータの単位であり、処理に必要なデータが全て入っている。 CodeGear の入力となる DataGear を Input DataGear と呼び、出力は Output DataGear と呼ぶ。

CodeGear は他の CodeGear と継続することでプログラムを構成する。図 2.1 のように Input DataGear を受け取り、 CodeGear で Output DataGear に変更を加えることでプログラムを記述していく。

図 2.1: CodeGear と DataGear

2.2　CbC での CodeGear と DataGear

CbC での簡単な記述例をソースコード 2.1、流れを図 2.2 に示す。CbC では CodeGear を __code キーワードを指定する。その際、Input DataGear は関数の引数として定義される。 CodeGear は継続で次の CodeGear に遷移するために関数末尾で goto キーワードの後に CodeGear 名と Input DataGear を指定する必要がある。

ソースコード 2.1 では　cs0、 cs1 が CodeGear で a+b が cs0 の Output DataGear であり、cs1 の Input DataGear である。

ソースコード 2.1: CodeGear の継続の例

```
__code cs0(int a, int b){
  goto cs1(a+b);
}

__code cs1(int c){
  goto cs2(c);
}
```

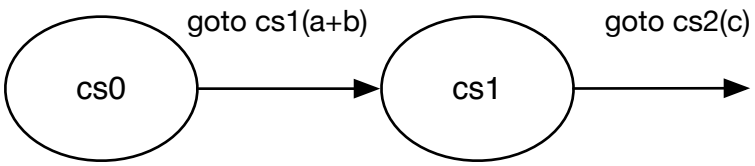

図 2.2: ソースコード 2.1 の流れ

CbC の継続は、呼び出し元の情報を持たずに処理を続ける。この継続を軽量継続と呼ぶ。ソースコード 2.1 は cs0 から cs1 へ継続した後、 cs0 には戻らずに次の継続に指定された CodeGear へ継続する。

CbC でのループ処理の記述例をソースコード 2.2、流れを図 2.3 に示す。軽量継続では関数呼び出しのスタックは存在しないが、ソースコード 2.2 のように計算中の値を DataGear で持つことでループ処理を行なうこともできる。

ソースコード 2.2: 階乗を求める CbC プログラムの例

```
__code print_factorial(int prod)
{
  printf("factorial = %d\n",prod);
  exit(0);
}

__code factorial0(int prod, int x)
{
  if ( x >= 1) {
    goto factorial0(prod*x, x-1);
  }else{
    goto print_factorial(prod);
  }

}

```

```
17 __code factorial(int x)
18 {
19   goto factorial0(1, x);
20 }
21
22 int main(int argc, char **argv)
23 {
24   int i;
25   i = atoi(argv[1]);
26
27   goto factorial(i);
28 }
```

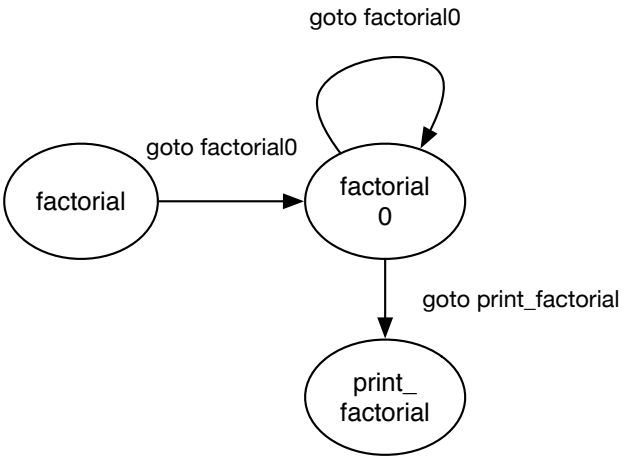

図 2.3: 階乗を求める CbC プログラムの流れ

2.3 メタ計算

メタ計算 (リフレクション) とは、対象とするレベルとメタなレベルを分離し、対象レベルでの推論や計算に関するメタな情報をメタレベルで明示的に記述し操作する。

メタ計算 (自己反映計算) [7] とはプログラムを記述する際に通常の処理と分離し、他に記述しなければならない処理である。例えばプログラム実行時のメモリ管理やスレッド管理、資源管理等の計算がこれに当たる。

2.4 Context

CbC で DataGear を扱う際、 Context と呼ばれる接続可能な CodeGear、 DataGear のリスト、Temporal DataGear のためのメモリ空間等を持っている Meta DataGear であ

る。CbC で必要な CodeGear 、 DataGear を参照する際は Context を通してアクセスする必要がある。

2.5 stub CodeGear

CodeGear が必要とする DataGear を取り出す際、 Context を通す必要がある。しかし、 Context を直接扱えるようにするのは信頼性を損なう。そのため CbC では Context を通して必要なデータを取り出して次の Code Gear に接続する stub CodeGear を定義している。CodeGear は stub CodeGear を介してのみ必要な DataGear へアクセスすることができる。 stub CodeGear は CodeGear 毎に生成され、次の CodeGear へと接続される。stub CodeGear は CodeGear の Meta CodeGear に当たる。

2.6 CbC による Interface の記述と継続

CodeGear は通常の関数と比べ、細かく分割されるためメタ計算をより柔軟に記述できる。 CodeGear 、 DataGear にはそれぞれメタレベルとして、 Meta CodeGear 、 Meta DataGear が存在する。

CbC で実装していくうちに、stub CodeGear の記述が煩雑になることが分かった。そのため 既存の実装を モジュールとして扱うため Interface という仕組みを導入した。

Interface は DataGear に対して何らかの操作 (API) を行う CodeGear とその CodeGear で使われる DataGear の集合を抽象化した メタレベルの DataGear として定義されている。

例として CbC による Stack Interface のソースコード 2.3, 2.4 がある。Stack への push 操作に注目して見ると、実態は SingleLinkedStack の push であることが 2.4 で分かる。実際の SingleLinkedStack の push では Stack を指定する必要があるが、Interface で実装した Stack では push 先の Stack が stackImpl として扱われている。この stackImpl は $Stack->push$ で呼ばれた時の Stack と同じになる。これにより、 ユーザーは実行時に Stack を指定する必要がなくなる。また、ユーザーが誤って異なる Stack を指定することを防ぐこともできる。

このように Interface 記述をすることで CbC で通常記述する必要がある一定の部分を省略し呼び出しが容易になる。

ソースコード 2.3: CbC での Stack-Interface の定義

```
1 typedef struct Stack<Type, Impl>{
2         union Data* stack;
3         union Data* data;
4         union Data* data1;
5
6         __code whenEmpty(...);
```

```
7         __code clear(Impl* stack,__code next(...));
8         __code push(Impl* stack,Type* data, __code next(...));
9         __code pop(Impl* stack, __code next(Type* data, ...));
10        __code pop2(Impl* stack, __code next(Type* data, Type* data1,
      ...));
11        __code isEmpty(Impl* stack, __code next(...), __code whenEmpty
      (...));
12        __code get(Impl* stack, __code next(Type* data, ...));
13        __code get2(Impl* stack, __code next(Type* data, Type* data1,
      ...));
14        __code next(...);
15 } Stack;
```

ソースコード 2.4: CbC での Stack-Interface の実装

```
1  Stack* createSingleLinkedStack(struct Context* context) {
2      struct Stack* stack = new Stack();
3      struct SingleLinkedStack* singleLinkedStack = new SingleLinkedStack()
       ;
4      stack->stack = (union Data*)singleLinkedStack;
5      singleLinkedStack->top = NULL;
6      stack->push = C_pushSingleLinkedStack;
7      stack->pop  = C_popSingleLinkedStack;
8      stack->pop2  = C_pop2SingleLinkedStack;
9      stack->get  = C_getSingleLinkedStack;
10     stack->get2  = C_get2SingleLinkedStack;
11     stack->isEmpty = C_isEmptySingleLinkedStack;
12     stack->clear = C_clearSingleLinkedStack;
13     return stack;
14 }
```

第3章　定理証明支援器 Agda での証明

型システムを用いて証明を行うことができる言語として Agda [4] が存在する。Agda は依存型という型システムを持つ。依存型とは型も第一級オブジェクトとする型システムで、依存型を持っている言語では型を基本的な操作に制限なしに使用できる。型システムは Curry-Howard 同型対応により命題と型付きラムダ計算が一対一で対応する。

本章では Agda で証明をするために必要な要素について説明し、Natural Deduction での証明とそれに対応して Agda 使って証明ができることを示す。

3.1　Agda の文法

Agda はインデントに意味を持つため、きちんと揃える必要がある。また、スペースの有無は厳格にチェックされる。なお、 -- の後はコメントである。

Agda のプログラムでは全てモジュール内部に記述されるため、まずはトップレベルにモジュールを定義する必要がある。トップレベルのモジュールはファイル名と同一になる。

通常のモジュールをインポートする時は `import` キーワードを指定する。また、インポートを行なう際に名前を別名に変更することもでき、その際は `as` キーワードを用いる。モジュールから特定の関数のみをインポートする場合は `using` キーワードの後に関数名を、関数の名前を変える時は `renaming` キーワードを、特定の関数のみを隠す場合は `hiding` キーワードを用いる。なお、モジュールに存在する関数をトップレベルで用いる場合は `opestack` に対する操作を定義しており、`n` キーワードを使うことで展開できる。モジュールをインポートする例をリスト 3.1 に示す。

ソースコード 3.1: Agda におけるモジュールのインポート

```
1 import Data.Nat                    -- import module
2 import Data.Bool as B              -- renamed module
3 import Data.List using (head)      -- import Data.head function
4 import Level renaming (suc to S) -- import module with rename suc to S
5 import Data.String hiding (_++_) -- import module without _++_
6 open import Data.List              -- import and expand Data.List
```

Agda における型指定は : を用いて行う。

例えば、変数 x が型 A を持つ、ということを表すには `x : A` と記述する。

データ型は、代数的なデータ構造で、その定義には `data` キーワードを用いる。`data` キーワードの後に `data` の名前と、型、 `where` 句を書きインデントを深くした後、値に

コンストラクタとその型を列挙する。例えば Bool 型を定義するとリスト 3.2 のようになる。Bool はコンストラクタ `true` と `false` を持つデータ型である。Bool 自身の型は `Set` であり、これは Agda が組み込みで持つ「型集合の型」である。Set は階層構造を持ち、型集合の集合の型を指定するには Set1 と書く。

ソースコード 3.2: Agda におけるデータ型 Bool の定義

```
1 data Bool : Set where
2   true  : Bool
3   false : Bool
```

関数の定義は、関数名と型を記述した後に関数の本体を = の後に記述する。関数の型には → 、 または-> を用いる。

例えば引数が型 `A` で返り値が型 `B` の関数は `A -> B` のように書ける。また、複数の引数を取る関数の型は `A -> A -> B` のように書ける。この時の型は `A -> (A -> B)` のように考えられる。Bool 変数 `x` を取って true を返す関数 `f` はリスト 3.3 のようになる。

ソースコード 3.3: Agda における関数定義

```
1 f : Bool -> Bool
2 f x = true
```

引数は変数名で受けることもでき、具体的なコンストラクタを指定することでそのコンストラクタが渡された時の挙動を定義できる。これはパターンマッチと呼ばれ、コンストラクタで case 文を行なっているようなもので例えば Bool 型の値を反転する `not` 関数を書くとリスト 3.4 のようになる。

ソースコード 3.4: Agda における関数 not の定義

```
1 not : Bool -> Bool
2 not  true = false
3 not false = true
```

パターンマッチでは全てのコンストラクタのパターンを含まなくてはならない。例えば、Bool 型を受け取る関数で `true` の時の挙動のみを書くことはできない。なお、コンストラクタをいくつか指定した後に変数で受けると、変数が持ちうる値は指定した以外のコンストラクタとなる。例えばリスト 3.5 の not は x には true しか入ることは無い。なお、マッチした値以外の挙動をまとめて書く際には _ を用いることもできる。

ソースコード 3.5: Agda におけるパターンマッチ

```
1 not : Bool -> Bool
2 not false = true
3 not x     = false
```

Agda にはラムダ式が存在している。ラムダ式とは関数内で生成できる無名の関数であり、`\arg1 arg2 -> function body` のように書くことができる。例えば Bool 型の引数

b を取って not を適用する not-apply をラムダ式で書くとリスト 3.6 のようになる。関数 not-apply をラムダ式を使わずに定義すると not-apply-2 になるが、この二つの関数は同一の動作をする。

ソースコード 3.6: Agda におけるラムダ式

```
not-apply : Bool -> Bool
not-apply = (\b -> not b)   -- use lambda

not-apply : Bool -> Bool
not-appy b = not b          -- not use lambda
```

Agda では特定の関数内のみで利用できる関数を where 句で記述できる。スコープは where 句が存在する関数内部のみであるため、名前空間が汚染させることも無い。例えば自然数 3 つを取ってそれぞれ 3 倍して加算する関数 f を定義するとき、 where を使うとリスト 3.7 のように書ける。これは f' と同様の動作をする。where 句は利用したい関数の末尾にインデント付きで where キーワードを記述し、改行の後インデントをして関数内部で利用する関数を定義する。

ソースコード 3.7: Agda における where 句

```
f : Int -> Int -> Int
f a b c = (t a) + (t b) + (t c)
  where
    t x = x + x + x

f' : Int -> Int -> Int
f' a b c = (a + a + a) + (b + b + b) + (c + c + c)
```

データ型のコンストラクタには自分自身の型を引数に取ることもできる (リスト 3.8)。自然数のコンストラクタは 2 つあり、片方は自然数ゼロ、片方は自然数を取って後続数を返すものである。例えば 0 は zero であり、1 は suc zero に、3 は suc (suc (suc zero)) に対応する。

ソースコード 3.8: Agda における自然数の定義

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

自然数に対する演算は再帰関数として定義できる。例えば自然数どうしの加算は二項演算子+としてリスト 3.9 のように書ける。

この二項演算子は中置関数として振る舞う。前置や後置で定義できる部分を関数名に _ として埋め込んでおくと、関数を呼ぶ時にあたかも前置や後置演算子のように振る舞うことができる。例えば !_ を定義すると ! true のように利用でき、_~ を定義すると false ~ のように利用できる。

また、Agda は再帰関数が停止するかを判別できる。この加算の二項演算子は左側がゼロに対しては明らかに停止する。左側が 1 以上の時の再帰時には `suc n` から `n` へと減っているため、再帰で繰り返し減らすことでいつかは停止する。もし `suc n` のまま自分自身へと再帰した場合、Agda は警告を出す。

ソースコード 3.9: Agda における自然数の加算の定義

```
1 _+_ : Nat -> Nat -> Nat
2 zero  + m = m
3 suc n + m = suc (n + m)
```

次に依存型について見ていく。依存型で最も基本的なものは関数型である。依存型を利用した関数は引数の型に依存して返す型を決定できる。なお、依存型の解決はモジュールのインポート時に行なわれる。

Agda で `(x : A) -> B` と書くと関数は型 A を持つ x を受け取り、B を返す。ここで B の中で x を扱っても良い。例えば任意の型に対する恒等関数はリスト 3.10 のように書ける。

ソースコード 3.10: 依存型を持つ関数の定義

```
1 identity : (A : Set) -> A -> A
2 identity A x = x
3
4 identity-zero : Nat
5 identity-zero = identity Nat zero
```

この恒等関数 `identitiy` は任意の型に適用可能である。実際に関数 `identitiy` を Nat へ適用した例が `identitiy-zero` である。

多相の恒等関数では型を明示的に指定せずとも `zero` に適用した場合の型は自明に `Nat -> Nat` である。Agda はこのような推論をサポートしており、推論可能な引数は省略できる。推論によって解決される引数を暗黙的な引数 (implicit arguments) と言い、変数を`{}` でくくることで表す。

例えば、`identitiy` の対象とする型 `A` を暗黙的な引数として省略するとリスト 3.11 のようになる。この恒等関数を利用する際は特定の型に属する値を渡すだけでその型が自動的に推論される。よって関数を利用する際は `id-zero` のように型を省略して良い。なお、関数の本体で暗黙的な引数を利用したい場合は `{variableName}` で束縛することもできる (`id'` 関数)。適用する場合も `{}`でくくり、`id-true` のように使用する。

ソースコード 3.11: Agda における暗黙的な引数を持つ関数

```
1 id : {A : Set} -> A -> A
2 id x = x
3
4 id-zero : Nat
5 id-zero = id zero
```

```
6
7  id' : {A : Set} -> A -> A
8  id' {A} x = x
9
10 id-true : Bool
11 id-true = id {Bool} true
```

Agda のデータには C における構造体に相当するレコード型も存在する。定義を行なう際は `record` キーワードの後にレコード名、型、`where` の後に `field` キーワードを入れた後、フィールド名と型名を列挙する。例えば x と y の二つの自然数からなるレコード `Point` を定義するとリスト 3.12 のようになる。レコードを構築する際は `record` キーワードの後の `{}` の内部に `fieldName = value` の形で値を列挙していく。複数の値を列挙する際は ; で区切る。

ソースコード 3.12: Agda におけるレコード型の定義

```
1  record Point : Set where
2    field
3      x : Nat
4      y : Nat
5
6  makePoint : Nat -> Nat -> Point
7  makePoint a b = record { x = a ; y = b }
```

構築されたレコードから値を取得する際には `RecordName.fieldName` という名前の関数を適用する (リスト 3.13 内 2 行目) 。なお、レコードにもパターンマッチが利用できる (リスト 3.13 内 5 行目)。レコード内の値は `record oldRecord {field = value ; ... }` という構文を利用し更新することができる。Point の中の x の値を 5 増やす関数 `xPlus5` はリスト 3.13 の 7,8 行目のように書ける。

ソースコード 3.13: Agda におけるレコードの射影、パターンマッチ、値の更新

```
1  getX : Point -> Nat
2  getX p = Point.x p
3
4  getY : Point -> Nat
5  getY record { x = a ; y = b} = b
6
7  xPlus5 : Point -> Point
8  xPlus5 p = record p { x = (Point.x p) + 5}
```

3.2 Natural Deduction

Natural Deduction (自然演繹) は Gentzen によって作られた論理及びその証明システムである。命題変数と記号を用いた論理式で論理を記述し、推論規則により変形することで求める論理式を導く。

Natural Deduction では次のように

$$\begin{array}{c} \vdots \\ A \end{array} \tag{3.1}$$

と書いた時、命題 A を証明したことを意味する。証明は木構造で表わされ、葉の命題は仮定となる。

$$\begin{array}{c} A \\ \vdots \\ B \end{array} \tag{3.2}$$

式 3.2 のように A を仮定して B を導いたとする。この時 A は alive な仮定であり、証明された B は A の仮定に依存していることを意味する。

ここで、推論規則により記号 $\Rightarrow$ を導入する。

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathcal{I}$$

$\Rightarrow \mathcal{I}$ を適用することで仮定 A は dead となり、新たな命題 $A \Rightarrow B$ を導くことができる。A という仮定に依存して B を導く証明から、「A が存在すれば B が存在する」という証明を導いたこととなる。このように、仮定から始めて最終的に全ての仮定を dead とすることで、仮定に依存しない証明を導ける。なお、dead な仮定は `[A]` のように `[]` で囲んで書く。

alive な仮定を dead にすることができるのは $\Rightarrow \mathcal{I}$ 規則のみである。それを踏まえ、natural deduction には以下のような規則が存在する。

- Hypothesis

  仮定。葉にある式が仮定となるため、論理式 A を仮定する場合に以下のように書く。

  A

- Introductions

  導入。証明された論理式に対して記号を導入することで新たな証明を導く。

$$\frac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B} \wedge\mathcal{I}$$

$$\frac{\begin{array}{c}\vdots \\ A\end{array}}{A \vee B} \vee 1\mathcal{I}$$

$$\frac{\begin{array}{c}\vdots \\ B\end{array}}{A \vee B} \vee 2\mathcal{I}$$

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathcal{I}$$

- Eliminations

  除去。ある論理記号で構成された証明から別の証明を導く。

$$\frac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{A} \wedge 1\mathcal{E}$$

$$\frac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{B} \wedge 2\mathcal{E}$$

$$\frac{\begin{array}{ccc} & [A] & [B] \\ \vdots & \vdots & \vdots \\ A \vee B & C & C\end{array}}{C} \vee\mathcal{E}$$

$$\frac{\begin{array}{cc}\vdots & \vdots \\ A & A \Rightarrow B\end{array}}{B} \Rightarrow \mathcal{E}$$

記号 $\vee, \wedge, \Rightarrow$ の導入の除去規則について述べた。natural deduction には他にも $\forall, \exists, \bot$ といった記号が存在するが、ここでは解説を省略する。

それぞれの記号は以下のような意味を持つ

- $\wedge$ conjunction。2 つの命題が成り立つことを示す。$A \wedge B$ と記述すると、 A かつ B と考えることができる。
- $\vee$ disjunction。2 つの命題のうちどちらかが成り立つことを示す。$A \vee B$ と記述すると、 A または B と考えることができる。
- $\Rightarrow$ implication。左側の命題が成り立つ時、右側の命題が成り立つことを示す。$A \Rightarrow B$ と記述すると、 A ならば B と考えることができる。

Natural Deduction では、これまでで説明したような規則を使い証明を行うことができる。

例として Natural Deduction で三段論法の証明を行う。このとき、「A は B であり、 B は C である。よって A は C である」 が証明するべき命題である。

この命題では 「A は B であり」と 「B は C である」の二つの小さい命題に分けられる。この「A は B であり」から、A から B が導出できることが分かり、これは $A \Rightarrow B$ と表せる。また、「B は C である」から、B から C が導出できることが分かる。これも「A は B であり」の時と同様に $B \Rightarrow C$ と表せる。

$$\dfrac{\dfrac{[A]_{(1)} \qquad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(A \Rightarrow B)} \wedge 1\mathcal{E}}{B} \Rightarrow \mathcal{E} \qquad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(B \Rightarrow C)} \wedge 2\mathcal{E}}{\dfrac{C}{A \Rightarrow C} \Rightarrow \mathcal{I}_{(1)}} \Rightarrow \mathcal{E}}{((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)} \Rightarrow \mathcal{I}_{(2)}$$

Natural Deduction では次のような証明木になる。

$$\dfrac{\dfrac{[A]_{(1)} \qquad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(A \Rightarrow B)} \wedge 1\mathcal{E}}{B} \Rightarrow \mathcal{E} \qquad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(B \Rightarrow C)} \wedge 2\mathcal{E}}{\dfrac{C}{A \Rightarrow C} \Rightarrow \mathcal{I}_{(1)}} \Rightarrow \mathcal{E}}{((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)} \Rightarrow \mathcal{I}_{(2)}$$

図 3.1: 自然演繹での三段論法の証明

これにより自然演繹を使って三段論法が証明できた。

3.3 Natural Deduction と 型付き λ 計算

ここでは、 Natural Deduction と型付き λ 計算の対応を定義する。対応は以下の表 3.1 のようになる。

Natural Deduction	型付き λ 計算
A	型 A を持つ変数 x
$A \Rightarrow B$	型 A を取り型 B の変数を返す関数 f
$\Rightarrow \mathcal{I}$	ラムダの抽象化
$\Rightarrow \mathcal{E}$	関数適用
$A \wedge B$	型 A と型 B の直積型 を持つ変数 x
$\wedge\mathcal{I}$	型 A,B を持つ値から直積型へのコンストラクタ
$\wedge 1\mathcal{E}$	直積型からの型 A を取り出す射影 fst
$\wedge 2\mathcal{E}$	直積型からの型 B を取り出す射影 snd

表 3.1: natural deuction と 型付き λ 計算との対応 (Curry-Howard Isomorphism)

この対応をもとに Agda で型付き λ 計算による証明を示す。ここでも先程 Natural Deduction で証明した三段論法を例とする。

ソースコード 3.14: Agda による三段論法の定義と証明

```
1  data × __ (A B : Set) : Set where
2    <_,_> : A → B → A ×  B
3
4  fst : {A B : Set} → A ×  B → A
5  fst < a , _ > = a
6
7  snd : {A B : Set} → A ×  B → B
8  snd < _ , b > = b
9
10
11 f : {A B C : Set} → ((A → B) ×  (B → C)) → (A → C)
12 f = λ  p x → (snd p) ((fst p) x)
```

自然演繹での三段論法の証明は、1 つの仮定から $\wedge 1\mathcal{E}$ と $\wedge 2\mathcal{E}$ を用いて仮定を二つ取り出し、それぞれに $\Rightarrow \mathcal{E}$ を適用した後に仮定を $\Rightarrow \mathcal{I}$ して導出していた。

ここで $\Rightarrow \mathcal{I}$ に対応するのは関数適用である。よってこの証明は「一つの変数から fst と snd を使って関数を二つ取り出し、それぞれを関数適用する」という形になる。これをラムダ式で書くとリスト 3.15 のようになる。仮定 $(A \to B) \times (B \to C)$ と仮定 A から A → C を導いている。

仮定に相当する変数 p の型は $(A \to B) \times (B \to C)$ であり、p からそれぞれの命題を取り出す操作が fst と snd に相当する。fst p の型は $(A \to B)$ であり、snd p の型は $(B \to C)$ である。もう一つの仮定 x の型は A なので、fst p を関数適用することで B が導ける。得られた B を snd p に適用することで最終的に C が導ける。

ソースコード 3.15: Agda における三段論法の証明

```
1 f : {A B C : Set} -> ((A -> B) × (B -> C)) -> (A -> C)
2 f = \p x -> (snd p) ((fst p) x)
```

このように Agda でも自然演繹と同様に証明を記述できる。

第4章　Agda における CbC の表現

前章では Agda の文法について説明した。本章では CbC と対応して CodeGear、DataGear、継続を Agda で表現する。また、 Agda で継続を記述することで得た知見を示す。

4.1　Agda での CodeGear 、 DataGear 、 継続の表現

DataGear はレコード型で表現できるため、 Agda のレコード型をそのまま利用して定義してく。記述は 4.1 のようになる。

ソースコード 4.1: Agda における DataGear の定義

```
1 record ds0 : Set where
2   field
3     a : Int
4     b : Int
5
6 record ds1 : Set where
7   field
8     c : Int
```

CodeGear は DataGear を受け取って DataGear を返すという定義であるため、$I \to O$ を内包する CodeGear 型のデータ型 (4.2) を定義する。

ソースコード 4.2: Agda における CodeGear 型の定義

```
1 data CodeSegment {l1 l2 : Level} (I : Set l1) (O : Set l2) : Set (l ⊔ l1
    ⊔ l2) where
2   cs : (I → O) → CodeSegment I O
```

CodeGear 型を定義することで、 Agda での CodeGear の本体は Agda での関数そのものと対応する。

CodeGear の実行は CodeGear 型から関数本体を取り出し、レコード型を持つ値を適用することに相当する。

CbC での軽量継続は

- 次に実行する CodeGear を指定する
- CodeGear に渡す DataGear を指定する

- 現在実行している CodeGear から制御を指定された CodeGear へと移す

の機能を持っている。
この機能を満たす関数はソースコード 4.3 として定義されている。

ソースコード 4.3: Agda における goto の定義

```
goto : {l1 l2 : Level} {I : Set l1} {O : Set l2}
   → CodeSegment I O → I → O
goto (cs b) i = b i
```

goto は CodeGear よりも一つ Level が上の Meta CodeGear にあたり、次に実行する CodeGear 型を受け取り、Input DataGear、 Output DataGear を返す。型になっている。

4.2 Agda での Stack、 Tree の実装

ここでは Agda での Stack 、 Tree の実装を示す。
Stack の実装を以下のソースコード 4.4 で示す。実装は SingleLinkedStack という名前で定義されている。定義されている API は push を例に残りは省略する。残りのの実装は付録に示す。

ソースコード 4.4: Agda における Stack の実装

```
record SingleLinkedStack {n : Level } (a : Set n) : Set n where
  field
  top : Maybe (Element a)
open SingleLinkedStack

pushSingleLinkedStack : {n m : Level } {t : Set m } {Data : Set n} →
   SingleLinkedStack Data → Data → (Code : SingleLinkedStack Data → t)
    → t
pushSingleLinkedStack stack datum next = next stack1
  where
    element = cons datum (top stack)
    stack1  = record {top = Just element}

-- Basic stack implementations are specifications of a Stack

singleLinkedStackSpec : {n m : Level } {t : Set m } {a : Set n} →
   StackMethods {n} {m} a {t} (SingleLinkedStack a)
singleLinkedStackSpec = record {
                                  push = pushSingleLinkedStack
                                ; pop  = popSingleLinkedStack
                                ; pop2 = pop2SingleLinkedStack
```

```
19                                    ; get  = getSingleLinkedStack
20                                    ; get2 = get2SingleLinkedStack
21                                    ; clear = clearSingleLinkedStack
22                                  }
23
24 createSingleLinkedStack : {n m : Level } {t : Set m } {a : Set n} →
       Stack {n} {m} a {t} (SingleLinkedStack a)
25 createSingleLinkedStack = record {
26                               stack = emptySingleLinkedStack ;
27                               stackMethods = singleLinkedStackSpec
28                             }
```

Element は SingleLinkedStack で扱われる要素の定義で、現在のデータ datum と次のデータを Maybe 型という値の存在が不確かな場合の型で包み、自身で再帰的に定義している。Maybe 型では値が存在する場合は Just 、 存在しない場合は　Nothing を返す。

SingleLinkedStack 型では、この Element の top 部分のみを定義している。

Stack に対する push 操作では stack と push する element 型の datum を受け取り、datum の next に現在の top を入れ、 stack の top を受け取った datum に切り替え、新しい stack を返すというような実装をしている。

Tree の実装 (以下のソースコード 4.5) は RedBlackTree という名前で定義されている。定義されている API は put 以後省略する。残りのの実装は付録に示す。

ソースコード 4.5: Agda における Tree の実装

```
1  record TreeMethods {n m : Level } {a : Set n } {t : Set m } (treeImpl :
       Set n ) : Set (m Level.⊔ n) where
2    field
3      putImpl : treeImpl → a → (treeImpl → t) → t
4      getImpl  : treeImpl → (treeImpl → Maybe a → t) → t
5  open TreeMethods
6
7  record Tree  {n m : Level } {a : Set n } {t : Set m } (treeImpl : Set n )
       : Set (m Level.⊔ n) where
8    field
9      tree : treeImpl
10     treeMethods : TreeMethods {n} {m} {a} {t} treeImpl
11     putTree : a → (Tree treeImpl → t) → t
12     putTree d next = putImpl (treeMethods ) tree d (\t1 → next (record {
       tree = t1 ; treeMethods = treeMethods} ))
13     getTree : (Tree treeImpl → Maybe a → t) → t
14     getTree next = getImpl (treeMethods ) tree (\t1 d → next (record {
       tree = t1 ; treeMethods = treeMethods} ) d )
15 open Tree
```

```
16
17 record Node {n : Level } (a k : Set n) : Set n where
18   inductive
19   field
20     key   : k
21     value : a
22     right : Maybe (Node a k)
23     left  : Maybe (Node a k)
24     color : Color {n}
25 open Node
26
27 leafNode : {n : Level } {a k : Set n}  → k → a → Node a k
28 leafNode k1 value = record {
29   key   = k1 ;
30   value = value ;
31   right = Nothing ;
32   left  = Nothing ;
33   color = Red
34   }
35 record RedBlackTree {n m : Level } {t : Set m} (a k : Set n) : Set (m
     Level.⊔ n) where
36   field
37     root : Maybe (Node a k)
38     nodeStack : SingleLinkedStack  (Node a k)
39     compare : k → k → CompareResult {n}
40 open RedBlackTree
41
42 putRedBlackTree : {n m : Level } {a k : Set n} {t : Set m} →
     RedBlackTree {n} {m} {t} a k → k → a → (RedBlackTree {n} {m} {t} a
     k → t) → t
43 putRedBlackTree {n} {m} {a} {k}  {t} tree k1 value next with (root tree)
44 ...                                  | Nothing = next (record tree {root =
     Just (leafNode k1 value) })
45 ...                                  | Just n2  = clearSingleLinkedStack (
     nodeStack tree) (\ s → findNode tree s (leafNode k1 value) n2 (\
     tree1 s n1 → insertNode tree1 s n1 next))
46
47 -- 以下省略
```

Node 型は key と value 、 Color と Node の rihgt 、 left の情報を持っている。Tree を構成する末端の Node は leafNode 型で定義されている。

RedBlackTree 型は root の Node 情報と Tree に関する計算をする際に、そこまでの Node の経路情報を保持するための nodeStack と 比較するための compare を持っている。

Tree の put 操作では tree 、 put するノードのキーと値 (k1、value) を引数として受け取り、Tree の root に Node が存在しているかどうかで場合分けしている。Nothing が返ってきたときは RedBlackTree 型の tree 内に定義されている root に受け取ったキーと値を新しいノードとして追加する。Just が返ってきたときは root が存在するので、経路情報を積むために nodeStack を初期化し、受け取ったキーと値で新たなノードを作成した後、ノードが追加されるべき位置までキーの値を比べて新しい Tree を返すというような実装になっている。

4.3 Agda における Interface の実装

Agda 側でも CbC 側と同様に interface を実装した。interface は record で列挙し、ソースコード 4.6 のように紐付けることができる。CbC とは異なり、 Agda では型を明記する必要があるため record 内に型を記述している。

例として Agda で実装した Stack 上の interface (ソースコード 4.6) を見る。Stack の実装は SingleLinkedStack(ソースコード**??**) として書かれている。それを Stack 側から interface を通して呼び出している。

ここでの interface の型は Stack の record 内にある pushStack や popStack などで、実際に使われる Stack の操作は StackMethods にある push や pop である。この push や pop は SingleLinkedStack で実装されている。

ソースコード 4.6: Agda における Interface の定義

```
1  open import Level renaming (suc to succ ; zero to Zero )
2  module AgdaInterface where
3
4  data Maybe {n : Level } (a : Set n) : Set n where
5    Nothing : Maybe a
6    Just    : a → Maybe a
7
8  record StackMethods {n m : Level } (a : Set n ) {t : Set m }(stackImpl :
       Set n ) : Set (m Level.⊔ n) where
9    field
10     push : stackImpl → a → (stackImpl → t) → t
11     pop  : stackImpl → (stackImpl → Maybe a → t) → t
12     pop2 : stackImpl → (stackImpl → Maybe a → Maybe a → t) → t
13     get  : stackImpl → (stackImpl → Maybe a → t) → t
14     get2 : stackImpl → (stackImpl → Maybe a → Maybe a → t) → t
15     clear : stackImpl → (stackImpl → t) → t
16 open StackMethods
17
18 record Stack {n m : Level } (a : Set n ) {t : Set m } (si : Set n ) : Set
       (m Level.⊔ n) where
```

```
19   field
20     stack : si
21     stackMethods : StackMethods {n} {m} a {t} si
22   pushStack :  a → (Stack a si → t) → t
23   pushStack d next = push (stackMethods ) (stack ) d (\s1 → next (record
       {stack = s1 ; stackMethods = stackMethods } ))
24   popStack : (Stack a si → Maybe a  → t) → t
25   popStack next = pop (stackMethods ) (stack ) (\s1 d1 → next (record {
      stack = s1 ; stackMethods = stackMethods }) d1 )
26   pop2Stack :  (Stack a si → Maybe a → Maybe a → t) → t
27   pop2Stack next = pop2 (stackMethods ) (stack ) (\s1 d1 d2 → next (
      record {stack = s1 ; stackMethods = stackMethods }) d1 d2)
28   getStack :  (Stack a si → Maybe a  → t) → t
29   getStack next = get (stackMethods ) (stack ) (\s1 d1 → next (record {
      stack = s1 ; stackMethods = stackMethods }) d1 )
30   get2Stack :  (Stack a si → Maybe a → Maybe a → t) → t
31   get2Stack next = get2 (stackMethods ) (stack ) (\s1 d1 d2 → next (
      record {stack = s1 ; stackMethods = stackMethods }) d1 d2)
32   clearStack : (Stack a si → t) → t
33   clearStack next = clear (stackMethods ) (stack ) (\s1 → next (record {
      stack = s1 ; stackMethods = stackMethods } ))
34
35 open Stack
```

interface を通すことで、実際には Stack の push では stackImpl と何らかのデータ a を取り、 stack を変更し、継続を返す型であったのが、 pushStack では 何らかのデータ a を取り stack を変更して継続を返す型に変わっている。

また、 Tree でも interface を記述した。

ソースコード 4.7: Tree Interface の定義

```
1  record TreeMethods {n m : Level } {a : Set n } {t : Set m } (treeImpl :
      Set n ) : Set (m Level.⊔ n) where
2    field
3      putImpl : treeImpl → a → (treeImpl → t) → t
4      getImpl  : treeImpl → (treeImpl → Maybe a → t) → t
5  open TreeMethods
6
7  record Tree  {n m : Level } {a : Set n } {t : Set m } (treeImpl : Set n )
       : Set (m Level.⊔ n) where
8    field
9      tree : treeImpl
10     treeMethods : TreeMethods {n} {m} {a} {t} treeImpl
11     putTree : a → (Tree treeImpl → t) → t
```

```
12      putTree d next = putImpl (treeMethods ) tree d (\t1 → next (record {
        tree = t1 ; treeMethods = treeMethods} ))
13      getTree : (Tree treeImpl → Maybe a → t) → t
14      getTree next = getImpl (treeMethods ) tree (\t1 d → next (record {
        tree = t1 ; treeMethods = treeMethods} ) d )
15
16  open Tree
```

interface を記述することによって、データを push する際に予め決まっている引数を省略することができた。また、 Agda で interface を記述することで CbC 側では意識していなかった型が、明確化された。

4.4 継続を使った Agda における Test , Debug

Agda ではプログラムのコンパイルが通ると型の整合性が取れていることは保証できるが、必ずしも期待した動作をするとは限らない。そのため、本研究中に書いたプログラムが正しい動作をしているかを確認するために行なった手法を幾つか示す。

今回は実験中にソースコード 4.8 のような Test を書いた。この Test では Stack をターゲットにしていて、 Stack に 1、 2 の二つの ℕ 型のデータを push した後、 pop2 Interface を使って Stack に入っている 1、 2 が Stack の定義である First in Last out の通りに 2、1 の順で取り出せるかを確認するために作成した。

ソースコード 4.8: Agda におけるテスト

```
1   -- after push 1 and 2, pop2 get 1 and 2
2
3   testStack02 : {m : Level } →  ( Stack  ℕ (SingleLinkedStack ℕ) → Bool {
        m} ) → Bool {m}
4   testStack02 cs = pushStack createSingleLinkedStack 1 (\s → pushStack s 2
        cs)
5
6
7   testStack031 : (d1 d2 : $\mathbb{N}$ ) → Bool {Zero}
8   testStack031 2 1 = True
9   testStack031 _ _ = False
10
11  testStack032 : (d1 d2 : Maybe ℕ) → Bool {Zero}
12  testStack032  (Just d1) (Just d2) = testStack031 d1 d2
13  testStack032  _ _ = False
14
15  testStack03 : {m : Level } → Stack  ℕ (SingleLinkedStack ℕ) → ((Maybe
        ℕ) → (Maybe ℕ) → Bool {m} ) → Bool {m}
```

```
testStack03 s cs = pop2Stack s (\s d1 d2 → cs d1 d2 )

testStack04 : Bool
testStack04 = testStack02 (\s → testStack03 s testStack032)

testStack05 : testStack04 ≡ True
testStack05 = refl
```

上の Test では、02 が 2 つのデータを push し、 03 で 二つのデータを pop する pop2 を行っている。それらをまとめて記述したものが 04 で 2 回 push し、 2 つのデータを pop する動作が正しく行われていれば 04 は True を返し、 05 で記述された型通りに互いに等しくなるため 05 が refl でコンパイルが通るようになる。今回は、 pop2 で取れた値を確認するため 03 の後に 031、 032 の二つの作成した。032 では 03 で扱っている値が Maybe であるため、 031 のような比較をする前に値が確実に存在していることを示す補題である。 032 を通すことで 031 では 2 つの値があり、かつ ℕ 型である事がわかる。031 では直接取れた値が 2、 1 の順番で入っているかを確認している。

この Test でエラーが出なかったことから、 Stack に 1、2 の二つのデータを push すると、 push した値が Stack 上から消えることなく push した順番に取り出せることが分かる。

また、継続を用いて記述することで関数の Test を書くことで計算途中のデータ内部をチェックすることができた。

ここでの $ は をまとめる糖衣構文で、 $ が書かれた一行を でくくることができる。

ソースコード 4.9 のように関数本体に記述してデータを返し、C-c C-n (Compute normal form) で関数を評価すると関数で扱っているデータを見ることができる。また、途中の計算で受ける変数名を変更し、 Return 時にその変更した変数名に変えることで、計算途中のデータの中身を確認することができる。評価結果はソースコード内にコメントで記述した。

ソースコード 4.9: Agda におけるテスト

```
test31 = putTree1 {_} {_} {ℕ} {ℕ} (createEmptyRedBlackTreeℕ ℕ ) 1 1
  $ \t → putTree1 t 2 2
  $ \t → putTree1 t 3 3
  $ \t → putTree1 t 4 4
  $ \t → getRedBlackTree t 4
  $ \t x → x

-- Just
-- (record
-- { key = 4
-- ; value = 4
-- ; right = Nothing
```

```
13 -- ; left = Nothing
14 -- ; color = Red
15 -- })
```

今回、この手法を用いることで複数の関数を組み合わせた findNode 関数内に異常があることが分かった。

第5章　Agda による CbC の証明

前章では Agda で CodeGear や DataGear の定義を示した。また、 CbC のコードを Agda にマッピングし等価なコードを生成できることを示した。本章では前章で生成した Interface の Stack や Tree のコードを使い Agda で Interface を経由したコードでの証明が可能であることを示す。

5.1　Agda による Interface 部分を含めた Stack の部分的な証明

ここでの証明とは Stack の処理が特定の性質を持つことを保証することである。
Stack の処理として様々な性質が存在する。例えば、

- Stack に push した値は存在する
- Stack に push した値は取り出すことができる
- Stack に push した値を pop した時、その値は Stack から消える
- どのような状態の Stack に値を push しても中に入っているデータの順序は変わらない
- どのような状態の Stack でも、値を push した後 pop した値は直前に入れた値と一致する

などの性質がある。

本セクションでは「どのような状態の Stack でも、値を push した後 pop した値は直前に入れた値と一致する」という性質を証明する。

まず始めに不定状態の Stack を定義する。ソースコード 5.1 の stackInSomeState 型は中身の分からない抽象的な Stack を表現している。ソースコード 5.1 の証明ではこの stackInSomeState に対して、 push を 2 回行い、 pop2 をして取れたデータは push したデータと同じものになることを証明している。

ソースコード 5.1: 抽象的な Stack の定義と push− >push− >pop2 の証明

```
1 stackInSomeState : {l m : Level } {D : Set l} {t : Set m } (s :
     SingleLinkedStack D ) → Stack {l} {m} D {t}  ( SingleLinkedStack  D )
2 stackInSomeState s =  record { stack = s ; stackMethods =
     singleLinkedStackSpec }
3
4 push→push→pop2 : {l : Level } {D : Set l} (x y : D ) (s :
     SingleLinkedStack D ) → pushStack (stackInSomeState s)  x (\s1 →
     pushStack s1 y (\s2 → pop2Stack s2 (\s3 y1 x1 →
5    (Just x ≡ x1) ∧ (Just y ≡ y1))))
6 push→push→pop2 {l} {D} x y s = record { pi1 = refl ; pi2 = refl }
```

この証明では stackInSomeState 型の s が抽象的な Stack で、そこに x 、 y の 2 つのデータを push している。また、 pop2 で取れたデータは y1 、 x1 となっていて両方が Just で返ってくるかつ、 x と x1 、 y と y1 がそれぞれ合同であることが仮定として型に書かれている。

この関数本体で返ってくる値は $x \equiv x1$ と $y \equiv y1$　のため record でまとめて refl で推論が通る。これにより、抽象化した Stack に対して push 、 pop を行うと push したものと同じものを受け取れることが証明できた。

5.2 Agda による Interface 部分を含めた Binary Tree の部分的な証明と課題

ここでは Binary Tree の性質の一部に対して証明を行う。Binary Tree の性質として挙げられるのは次のようなものである

- Tree に対して Node を Put することができる。
- Tree に Put された Node は Delete されるまでなくならない。
- Tree に 存在する Node とその子の関係は必ず「右の子 > Node」かつ「Node > 左の子」の関係になっている。
- どのような状態の Tree に値を put しても Node と子の関係は保たれる
- どのような状態の Tree でも値を Put した後、その値を Get すると値が取れる。

ここで証明する性質は
!! と書かれているところはまだ記述できていない部分で ? としている部分である。

ソースコード 5.2: Tree Interface の証明

```
redBlackInSomeState : { m : Level } (a : Set Level.zero) (n : Maybe (Node a ℕ)) {t : Set m} → RedBlackTree {Level.zero} {m} {t} a ℕ
redBlackInSomeState {m} a n {t} = record { root = n ; nodeStack = emptySingleLinkedStack ; compare = compare2 }

putTest1 :{ m : Level } (n : Maybe (Node ℕ ℕ))
         → (k : ℕ) (x : ℕ)
         → putTree1 {_} {_} {ℕ} {ℕ} (redBlackInSomeState {_} ℕ n {Set Level.zero}) k x
         (\ t → getRedBlackTree t k (\ t x1 → check2 x1 x  ≡ True))
putTest1 n k x with n
...  | Just n1 = lemma2 ( record { top = Nothing } )
   where
     lemma2 : (s : SingleLinkedStack (Node ℕ ℕ) ) → putTree1 (record { root = Just n1 ; nodeStack = s ; compare = compare2 }) k x λ ( t →
          GetRedBlackTree.checkNode t k λ ( t1 x1 → check2 x1 x ≡ True) (root t))
     lemma2 s with compare2 k (key n1)
     ... |  EQ = lemma3 {!!}
        where
           lemma3 : compare2 k (key n1) ≡  EQ → getRedBlackTree {_} {_} {ℕ} {ℕ} {Set Level.zero} ( record {  root = Just ( record {
               key   = key n1 ; value = x ; right = right n1 ; left  = left n1 ; color = Black
               } ) ; nodeStack = s ; compare = λ  x1 y → compare2 x1 y  } ) k ( \ t x1 → check2 x1 x  ≡ True)
           lemma3 eq with compare2 x x | putTest1Lemma2 x
           ... | EQ | refl with compare2 k (key n1)  | eq
           ...              | EQ | refl with compare2 x x | putTest1Lemma2 x
           ...                    | EQ | refl = refl
     ... |  GT = {!!}
     ... |  LT = {!!}

...  | Nothing =  lemma1
   where
     lemma1 : getRedBlackTree {_} {_} {ℕ} {ℕ} {Set Level.zero} ( record {  root = Just ( record {
               key   = k ; value = x ; right = Nothing ; left  = Nothing ; color = Red
        } ) ; nodeStack = record { top = Nothing } ; compare = λ  x1 y → compare2 x1 y  } ) k
```

```
31          ( \ t x1 → check2 x1 x  ≡ True)
32      lemma1 with compare2 k k | putTest1Lemma2 k
33      ... | EQ | refl with compare2 x x | putTest1Lemma2 x
34      ...                 | EQ | refl = refl
```

この Tree の証明では、不定状態の Tree を redBlackInSomeState で作成し、その状態の Tree に一つ Node を Put し、その Node を Get することができるかを証明しようとしたものである。

しかし、この証明は Node を取得する際に Put した Node が存在するか、 Get した Node が存在するのか、などの条件や、 Get した Node と Put した Node が合同なのか、key の値が等しい場合の eq は合同と同義であるのか等の様々な補題を証明する必要があった。今回この証明では Put した Node と Get した Node が合同であることを記述しようとしていたがそれまでの計算が異なるため完全に合同であることを示すことが困難であった。

今後の研究では 5.3 で説明する Hoare Logic を元に証明を行おうと考えている。

5.3 Hoare Logic

Hoare Logic [8] とは Tony Hoare によって提案されたプログラム正しさを推論する手法である。図 5.1 のように、 P を前状態 (Pre Condition) 、C を処理 (Command) 、 Q を後状態 (Post Condition) とし、$\{P\}\ C\ \{Q\}$ のように考えたとき、プログラムの処理を「前状態を満たした後、処理を行い状態が後状態に変化する」といった形で考える事ができる。

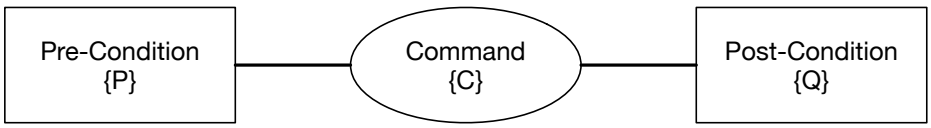

図 5.1: hoare logic の流れ

このとき、後状態から前状態を推論することができればそのプログラムは部分的に正しい動きをすることを証明することができる。

この Hoare Logic の前状態、処理、後状態を CodeGear、 input/output の DataGear が表 5.2 のように表せるのではないか考えている。

この状態を当研究室で提案している CodeGear、 DataGear の単位で考えると Pre Condition が CodeGear に入力として与えられる Input DataGear、Command が CodeGear、Post Condition を Output DataGear として当てはめることができると考えている。

今後の研究では CodeGear、 DataGear、継続の概念を Hoare Logic に当てはめ Agda に当てはめ、幾つかの実装を証明していく。

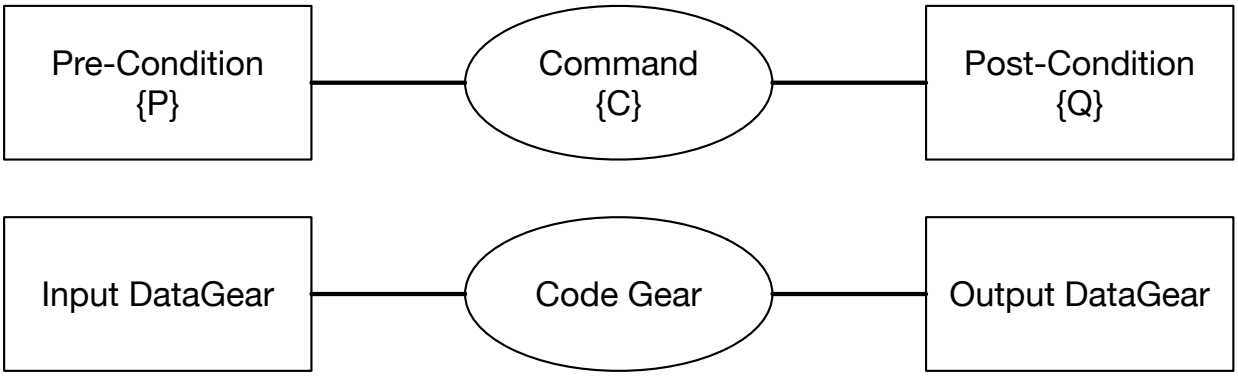

図 5.2: cbc と hoare logic

第6章　まとめ

本研究では CodeGear、 DataGear を用いたプログラミング手法を使い、Agda で Interface を用いたプログラムを実装し、証明を記述した。これにより、 CbC で記述した時には細かく分かっていなかった Interface の型が明確になった。また、 Hoare Logic を CodeGear 、 DataGear と対応させることで Hoare Logic ベースで証明が進められると考えている。今回の研究中に継続を利用することで得られた知見は、今後の研究で大いに役立つと考える。

今後の課題としては、CbC 上での RedBlackTree の実装や、Agda 上での RedBlackTree の実装と証明がある。また、 CodeGear、DataGear をベースにした Hoare Logic を Agda で実装する。Agda 定義した Hoare Logic を使い、いくつかの証明を実際に記述し、書き方を確立するなどが考えられる。他にも、タスクスケジューラの実装を Agda に移し、SynchronizedQueue の同期、非同期の検証などが考えられる。

参考文献

[1] Spin - formal verification. http://spinroot.com/spin/whatispin.html. Accessed: 2018/02/09(Fri).

[2] Nusmv home page. http://nusmv.fbk.eu/. Accessed: 2018/02/09(Fri).

[3] The cbmc homepage. http://www.cprover.org/cbmc/. Accessed: 2018/02/09(Fri).

[4] The agda wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed: 2018/2/9(Fri).

[5] Welcome! — the coq proof assistant. https://coq.inria.fr/. Accessed: 2018/02/09(Fri).

[6] 徳森 海斗, 河野 真治. LLVM Clang 上の Continuation based C コンパイラ の改良, 2015.

[7] 博靖菅野, 二郎田中. 非標準理論とその応用：メタ推論とリフレクション. 情報処理, Vol. 30, No. 6, jun 1989.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, Vol. 12, pp. 576–580, 1969.

[9] 宮城 光希, 河野 真治. CbC 言語による OS 記述, 2017.

[10] 健太比嘉, 真治河野. Verification method of programs using continuation based c. 情報処理学会論文誌プログラミング（PRO）, Vol. 10, No. 2, pp. 5–5, feb 2017.

[11] Ats-pl-sys. http://www.ats-lang.org/. Accessed: 2018/02/09(Fri).

[12] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[13] Welcome to agda ' s documentation! — agda 2.6.0 documentation. http://agda.readthedocs.io/en/latest/index.html. Accessed: 2018/2/9(Fri).

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

[15] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

謝辞

本研究の遂行，また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深く感謝したします。また、一年間共に研究を行い、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みなさんに感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2018 年 3 月
外間政尊