令和2年度 卒業論文

Continuation based C による RedBlackTree の Hoare Logic を用いた検証

琉球大学工学部工学科知能情報コース

175707H 氏名 上地 悠斗

指導教員 : 河野 真治

目 次

図 目 次

表 目 次

ソースコード目次

第1章　はじめに

OS やアプリケーションの信頼性を高めることは重要な課題である。信頼性を高める為にはプログラムが仕様を満たした実装を検証する必要がある。具体的には「モデル検査」や「定理証明」などが検証手法としてあげられる。

当研究室では Continuation based C (CbC) という言語を開発している。CbC とは、C 言語からループ制御構造とサブルーチンコールを取り除き、継続を導入した C 言語の下位言語である。その為、それを実装した際のプログラムが正確に動作するのか検証を行いたい。

1.1　研究目的

仕様に合った実装を実施していることの検証手法として Hoare Logic が知られている。Hoare Logic は事前条件が成り立っているときにある計算 (以下コマンド) を実行した後に、事後条件が成り立つことでコマンドの検証を行う。この定義が CbC の実行を継続するという性質と相性が良い。

CbC では実行を継続するため、ある関数の実行結果は事後条件になるが、その実行結果が遷移する次の関数の事前条件になる。それを繋げていくため、個々の関数の正当性を証明することと接続の健全性について証明するだけでプログラム全体の検証を行うことができる。

CbC ではループ制御構造を取り除いているため、CbC にてループが含まれるプログラムを作成した際の検証を行う必要がある。先行研究では CbC における WhileLoop の検証を行なっている。

Agda が変数への再代入を許していない為、ループが存在し、かつ再代入がプログラムに含まれる RedBlackTree の検証を行いたい。

これらのことから、CbC に対応するように Agda で RedBlackTree を記述し、Hoare Logic により検証を行うことを目指す。

1.2　論文の構成

本論文は以下の流れで構成されている。

- 第 1 章は, 本研究の背景と目的を述べる
- 第 2 章は, 検証を行う CbC について述べる

- 第 3 章は, 証明に使用する言語である Agda について述べる
- 第 4 章は, 検証手法である Hoare Logic について述べる
- 第 5 章は, Agda を Continuation style で記述する方法について述べる
- 第 6 章は, Red Back Tree について述べる
- 第 7 章は、Agda での Red Black Tree の実装方法について述べる
- 第 8 章は、Red Black Tree の Hoare Logic を用いた検証について述べる
- 第 9 章は, 本研究におけるまとめと今後の課題について述べる

第2章　Continuetion based C

2.1　Continuation based C

CbC とはC 言語からループ制御構造とサブルーチンコールを取り除き、継続を導入したC 言語の下位言語である。継続呼び出しは引数付き goto 文で表現される。また、CodeGear を処理の単位、DataGear をデータの単位として記述するプログラミング言語である。CbC のプログラミングでは DataGear を CodeGear で変更し、その変更を次の CodeGear に渡して処理を行う。

2.2　Code Gear / Data Gear

CbC では、検証しやすい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。CodeGear の入力となる DataGear を Input DataGear と呼び、出力は Output DataGear と呼ぶ。

CodeGear 間の移動は継続を用いて行われる。継続は関数呼び出しとは異なり、呼び出した後に元のコードに戻れず、次の CodeGear へ継続を行う。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。

2.3　Meta Code Gear / Meta Data Gear

プログラムの記述する際は、ノーマルレベルの計算の他に、メモリ管理、スレッド管理、資源管理等を記述しなければならない処理が存在する。これらの処理をメタ計算と呼ぶ。

メタ計算は OS の機能を通して処理することが多く、信頼性の高い記述が求められる。そのため、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 は CbC 上でのメタ計算で、通常の CodeGear を実行する際に必要なメタ計算を分離するための単位である。CodeGear を実行する前後や DataGear の大枠として Meta Gear が存在している。

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。

2.4 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。以下はフィボナッチ数列の n 番目を求める CbC と C 言語のソースコードである。

Listing 2.1: plus

```
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 }
```

Listing 2.2: plus

```
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 }
```

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。

Listing 2.3: plus

```
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
17 100000e87: e8 98 ff ff ff         callq   -104 <_fin>
18 100000e8c: 48 8b 55 f0           movq    -16(%rbp), %rdx
19 100000e90: 48 8b 45 e8           movq    -24(%rbp), %rax
20 100000e94: 48 01 c2             addq    %rax, %rdx
21 100000e97: 48 8b 45 e8           movq    -24(%rbp), %rax
22 100000e9b: 48 01 c2             addq    %rax, %rdx
23 100000e9e: 48 8b 4d f0           movq    -16(%rbp), %rcx
24 100000ea2: 48 8b 45 e8           movq    -24(%rbp), %rax
25 100000ea6: 48 01 c1             addq    %rax, %rcx
26 100000ea9: 48 8b 45 f8           movq    -8(%rbp), %rax
27 100000ead: 48 83 e8 02           subq    $2, %rax
28 100000eb1: 48 89 ce             movq    %rcx, %rsi
29 100000eb4: 48 89 c7             movq    %rax, %rdi
30 100000eb7: e8 96 ff ff ff         callq   -106 <_fib>
31 100000ebc: 90                nop
32 100000ebd: c9                leave
33 100000ebe: c3                retq
```

Listing 2.4: plus

```
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)
```

```
100000e5e: 48 89 75 f0            movq    %rsi, -16(%rbp)
100000e62: 48 89 55 e8            movq    %rdx, -24(%rbp)
100000e66: 48 83 7d f8 00         cmpq    $0, -8(%rbp)
100000e6b: 75 0c                  jne     12 <_fib+0x27>
100000e6d: 48 8b 45 f0            movq    -16(%rbp), %rax
100000e71: 48 89 c7               movq    %rax, %rdi
100000e74: e8 ab ff ff ff         callq   -85 <_fin>
100000e79: 48 83 7d f8 01         cmpq    $1, -8(%rbp)
100000e7e: 75 0c                  jne     12 <_fib+0x3a>
100000e80: 48 8b 45 e8            movq    -24(%rbp), %rax
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
```

第3章　定理証明支援系言語 Agda

Agda とは定理証明支援器であり、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ことが可能である。また、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ため Agda では記述したプログラムを証明することができる。

3.1　Agda の基本

3.1.1　関数の実装

本節では Agda の基本事項について 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。

Listing 3.1: plus の実装

```
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
```

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 つの変数が x, y を受け取り、自然数を返すという定義になる。

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 三項演算子の実装

_ (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、三項演算子を実装することができる。以下に、三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。

Listing 3.2: 三項演算子を用いた plus の実装

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

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三項演算子を使用して記述する。

3.1.3 Agda におけるラムダ計算

λ

3.1.4 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。例として既出で Data 型である $\mathbb{N}$ の実装を ソースコード 3.3 に示す。

Listing 3.3: Nat

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

実装から、$\mathbb{N}$ 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 $\mathbb{N}$ のみであるが、suc は $(n : \mathbb{N}) \rightarrow \mathbb{N}$ である。つまり、suc 自体の型は $\mathbb{N}$ であるが、そこから $\mathbb{N}$ 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。

3.1.5 パターンマッチ

3.1.6 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。

Listing 3.4: And

```
1 record _∧_ (A B : Set) : Set where
2   field
3     p1 : A
4     p2 : B
```

また、Agda の関数定義では_(アンダースコア) で囲むことで三項演算子を定義することができる。

これを使用して三段論法を定義することができる。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。

Listing 3.5: syllogism

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

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 Agda で使用するもの

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。

Listing 3.6: 入力を省略する Agda コードの例

```
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
```

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。

第4章　Hoare Logic

4.1　Hoare Logic

Hoare Logic4.2 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、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。

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

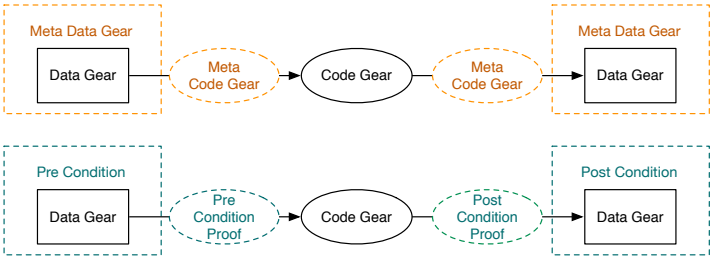
この 3 つ組は Hoare Triple と呼ばれる。

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。

4.2　Hoare による Code Gear の検証

図 4.2 が agda にて Hoare Logic を用いて Code Gear を検証する際の流れになる。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。



第5章　Continuation based C と Agda

5.1　検証手法

本章では検証する際の手法を説明する。CodeGear の引数となる DataGear が事前条件となり、それを検証する為の Pre Condition を検証する為の Meta Gears が存在する。その後、さらに事後条件となる DetaGear も Meta Gears にて検証する。これらを用いて Hoare Logic によりプログラムの検証を行いたい。

5.1.1　CbC 記法で書く agda

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。ソースコード 5.1 が例となるコードである。

Listing 5.1: Agda での CodeGear の例

```
1  record Env : Set where
2    field
3      varx : ℕ
4      vary : ℕ
5  open Env
6
7  plus-com : {l : Level} {t : Set l} → Env → (next : Env → t) → (exit : Env → t) → t
8  plus-com env next exit with vary env
9  ... | zero  = exit (record { varx = varx env ; vary = vary env })
10 ... | suc y = next (record { varx = suc (varx env) ; vary = y })
11
12 {-# TERMINATING #-}
13 plus-p : {l : Level} {t : Set l} → (env : Env) → (exit : Env → t) → t
14 plus-p env exit = plus-com env ( λ env → plus-p env exit ) exit
15
16 plus : ℕ → ℕ → Env
17 plus x y = plus-p (record { varx = x ; vary = y }) ( λ env → env)
```

前述した加算を行うコードと比較すると、不定の型 (t) により継続を行なっている部分が見える。これが Agda で表現された CodeGear となる。

5.1.2 agda による 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。今回はその Meta Gears を Agda による検証の為に用いる。

- 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。これを用いることで、仕様となる制約条件を記述することができる。

- 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。故に、Meta CodeGear は Agda で記述した CodeGear の検証そのものである

第6章　Red Black Tree

6.1　Tree

Tree (木構造) とは、非常に有用なデータ構造である。下図の ○ の部分を node (節) と呼び、top node を root(根) と呼ぶ。特に、根を持つ木構造のことを強調して、Rooted Tree (根付き木) と呼ぶ事がある。

6.2　Binary Tree

各 node からすぐ下に辺で結ばれている node をその node の child または son (子あるいは子供) と呼ぶ。child 側から上の辺を parent (親) と呼ぶ。下図のように、各 node が持つ child が高々 2 つである Tree を Binary Tree (2 分木) と呼ぶ。

6.3　Binary Search Tree

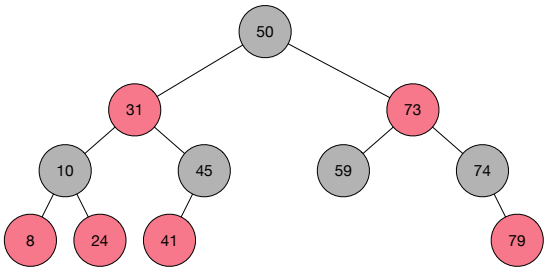
Rooted Binary Tree に対して、以下の制約を持つものを、Binary Search Tree と呼ぶ。
左側の子孫にある要素　＜ 親 ＜ 右側の子孫にある要素

6.4　RedBlackTree

RedBlackTree (または赤黒木) とは平衡 2 分探索木の一つである。2 分探索木の点にランクという概念を追加し、そのランクの違いを赤と黒の色で分け、以下の定義に基づくように木を構成した物である。図では省略しているが、値を持っている点の下に黒色の空の葉があり、それが外点となる。

1. 各点は赤か黒の色である。
2. 点が赤である場合の親となる点の色は黒である。
3. 外点 (葉。つまり一番下の点) は黒である。
4. 任意の点から外点までの黒色の点はいずれも同数となる。

参考となる図 6.4 を以下に示す。上記の定義を満たしていることが分かる。



6.5 Left Learing Red Black Tree

Left Learing Red Black Tree とは Red Black Tree の変形である。Red Black Tree の 仕様を満たしながら、実装が容易である。

以下の図のように、赤色の node は parent から見て左の node にしか 現れない Red Black Tree となる。これにより、パターンマッチの分岐を減らす事ができ、実装が容易になる。

本来の Red Black Tree の実装は困難であるため、本論では Red Black Tree の仕様を満たしている Left Learning Red Black Tree を検証する。

第7章　Red Black Tree の実装

7.1　Agda による Red Black Tree の 実装

通常は再起処理を使用して Red Black Tree を行う。しかし、今回は CbC で実装された Red Black Tree を検証するので、下図の手順を元に実装を行う。

上記に示した手順通りに Agda で記述すると以下のようなソースコードになる。

以上のように Tree の基本操作である insert, find, delete の実装を行った。

第8章　Red Black Tree の検証

Input Data Gear が Pre Condition を、Output Data Gear が Post Condition を 満たしているか検証することで Hoare Logic に当てはめる。

以下の要素を検証するための Meta Code Gear を実装する。

そして、Meta Code Gear から 生成される Meta Data Gear が Pre / Post Condition を満たしているのか確認することで、関数一つ一つに対して Hoare Logic を用いた検証を行う

第9章　今後の課題

9.1　今後の課題

今後の課題として、以下が挙げられる。RedBlackTree の基本操作として insert や delete が挙げられる。通常は、再代入などを用いて実装を行うと思われるが、Agda が変数への代入を許していないため、操作後の RedBlackTree を再構成するように実装を行う必要がある。その際にどこの状態の検証を行うかが課題になっている。

先行研究にて、個々の Code Gear の条件を書いてそれを接続することは Agda で実装されている。しかし、接続された条件が健全であるか証明されていない。

証明されていない部分というのは、プログラム全体はいくつかの Code Gear の集まりだが、Code Gear 実行後の事後条件が正しく次に実行される Code Gear の事前条件として成り立っているか、それが最初からプログラムの停止まで正しく行われているかという部分である。

今後はこの接続された条件の健全性の証明から行っていく。

謝辞

感謝します。

2021 年 2 月
上地　悠斗

参考文献

[1] Hoare logic - **並列信頼研** mercurial repository, **http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/**, Accessed: 2020/09/10.

[2] 外間政尊, "Continuation based c での hoare logic を用いた仕様記述と検証," M.S. thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2019.

[3] C. A. R. Hoare, "An axiomatic basis for computer programming," Commun. ACM, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 0001-0782. DOI: **10.1145/363235.363259**. [Online]. Available: **https://doi.org/10.1145/363235.363259**.

[4] The agda wiki, **http://wiki.portal.chalmers.se/agda/pmwiki.php**, Accessed: 2020/09/10.

[5] Welcome to agda's documentation! — agda latest documentation, **http://agda.readthedocs.io/en/latest/**, Accessed: 2020/09/10.

[6] A. Stump, Verified Functional Programming in Agda. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2016, ISBN: 978-1-97000-127-3.

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

[8] 徳森海斗, "Llvm clang 上の continuation based c コンパイラ の改良," M.S. thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2016.

[9] 渡邉, **データ構造と基本アルゴリズム**, 2000.