2024 年度 卒業論文

Bachelor's Thesis

GearsAgda による Red Black Tree の証明付き実装に関する研究

Implementation with proof of Red-Black-Tree using GearsAgda

琉球大学工学部工学科知能情報コース

205718C　森　逸汰

指導教員　　河野　真治

要旨

当研究室では，GearsOS を開発しており，信頼性を向上させることが現在の課題である．GearsOS では，RedBlackTree の採用が予定されており，これを数学的な証明を用いて検証することで，GearsOS の信頼性向上を図る．

GearsAgda では，HoareLogic を Agda 上に実装することで，BinarySearchTree の検証を可能にした．これを，RedBlackTree に拡張していきたい．拡張するためには，Invariant(ループや再帰で不変の条件) を見つけ出し，それを保持したまま処理を行う実装を記述する必要がある．また，GearsOS は CbC 言語で記述されており，再帰呼び出しができないなどの特徴があるため，それに直接対応した GearsAgda を用いて検証を行う必要がある．

本論文では，Agda での記述方法をはじめ，GearsAgda において Invariant を用いた RedBlackTree の証明付き実装を目指す．

Abstract

Our laboratory has developed GearsOS, and our current task is to improve the reliability of GearsOS. GearsOS will adopt RedBlackTree, and we will verify this by using mathematical proofs.

GearsAgda enables verification of BinarySearchTree by implementing HoareLogic on Agda. We would like to extend this to RedBlackTree. To extend it, we need to write an implementation that finds Invariants (conditions that are invariant in loops and recursion) and processes them while preserving them. In addition, since GearsOS is written in the CbC language and has features such as not being able to perform recursive calls, it is necessary to perform verification using GearsAgda, which is directly compatible with it.

In this paper, we aim to provide a proven implementation of RedBlackTree using Invariant in GearsAgda, including how to describe it in Agda.

目次

図目次

表目次

ソースコード目次

第 1 章

プログラムの信頼性

1.1　背景と目的

昨今ではプログラミング技術の波及により，世界中で様々なアプリケーションが作成され，使用されている．しかしながら，そのアプリケーションの安定性やデータの安全性などは十分であるとは言えない．十分な信頼性を持たないアプリケーションは，システム障害や個人情報の流出などの問題を引き起こす可能性がある．このことからも OS を含むアプリケーションは，高い信頼性を持つことが望ましい．

信頼性を高める手法として，テストやモデル検査などが考えられるが，数学的な証明を行うことでも信頼性を向上させることができる．テストやモデル検査が，特定の条件やケースを対象とするのに対し，数学的な証明は，プログラム全体の挙動や性質を対象とする．これにより，数学的な証明は，テストやモデル検査と比べ，より高い信頼性を示すことができる．

本研究室では，CbC (Continuation based C) を採用した GearsOS を開発している．GearsOS では，ファイルシステムやデータベースの設計において，二分探索木である RedBlackTree を採用することが予定されている．この，RedBlackTree を数学的に証明することで，ファイルシステムやデータベースの信頼性が向上することが考えられる．

本研究では，GearsOS に採用される RedBlackTree を GearsAgda を用いて検証し，証明付き実装を行うことで，GearsOS の信頼性を高めることが目的である．

1.2 論文の構成

本論文は以下の章で構成される．第 1 章のプログラムの信頼性に続き，第 2 章の基礎概念では，本論文を読むにあたって必要な基礎概念や，Agda の記述方法について述べる．第 3 章では，Invariant を用いて RedBlackTree を証明する手法について述べる．第 4 章では，Invariant を用いた RedBlackTree を実装し解説する．第 5 章では，find 操作を実装，実行し，その結果を考察していく．最後に，第 6 章で本論文のまとめと今後の展望について述べる．

第 2 章

基礎概念

本章では，本論文を読み進めるにあたって必要となる知識，基礎概念を解説していく．

2.1　CbC

CbC とは，Continuation based C を略称したものであり，本研究室で開発している GearsOS を構成する言語である．CbC 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．また，dataGear をデータの単位，code Gear を処理の単位として記述する．ノーマルレベルとメタレベルの切り分けが容易などの利点がある．

2.2　GearsOS

GearsOS とは，本研究室で開発している信頼性の保障を目的とした OS である．CbC で記述された軽量継続を基本とする構成になっている．現在は，形式手法による証明を目的とする「GearsAgda」, ユーザーレベルタスクマネジメントの実装を目的とする「GearsOS」, スタンドアロン OS の開発を目的とする「x.v6」 の 3 つ GearsOS の開発が進んでいる．ファイルシステムとデータベースでは RedBlackTree の採用が予定されており，本研究における信頼性向上の対象になっている．CbC で記述されているため，ノーマルレベルとメタレベルの切り分けが容易に行えるなどの利点がある．

2.3 BinarySearchTree

BinarySearchTree とは，ファイルシステムなどで使われる木構造の一種である．ノード，リーフ，ルートの 3 つの要素で成り立っている．ノード間の関係は家系図に見立てた用語で表現される．例えば，あるノードから見た上のノードは親ノードであり，下につながるノードは子ノードと呼ぶ．また，key と value を持ち，key の大小関係が 左の子の *key* < 親 *key* 右の子の *key* であるという制約がある．

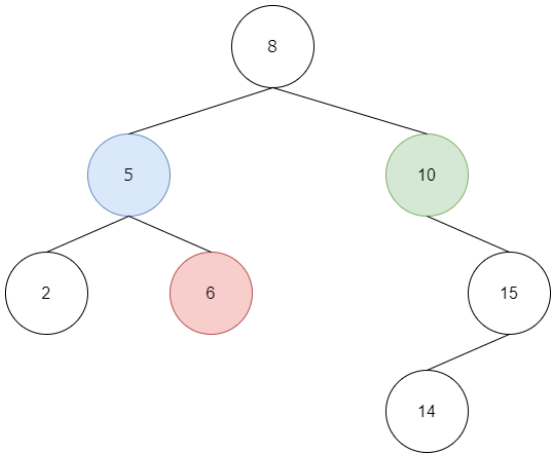

図 2.1 BinarySearchTree におけるノード関係

図 2.1 を見ると前述した制約が満たされていることがわかる．また，赤いノードから見た青のノードは親子関係であり，赤ノードから見た緑ノードは叔父に当たることがわかる．本研究の対象である RedBlackTree の元となる木構造となっている．

2.4 RedBlackTree

RedBlackTree とは，GearsOS に採用される予定のあるバランスした二分探索木の一種である．BinarySearchTree と key の大小関係性は同じであるが，それぞれのノードに赤と黒の色の概念を持ち以下の 4 つの制約を持つ．

1. 各ノードは赤か黒の色を持つ
2. 赤のノードは赤のノードを子に持たない
3. リーフノードはすべて黒である

4. 任意のノードについて，そのノードから子孫の葉までの道に含まれる黒いノードの数は，選んだ葉によらず一定である．

図 2.2 を見ると，上記の制約が満たされていることを確認することができる．

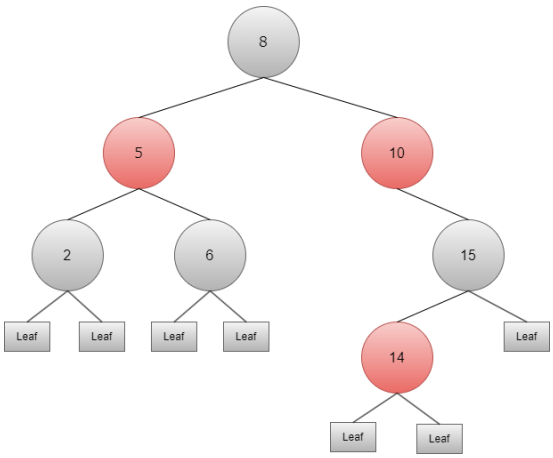

図 2.2　RedBlackTree におけるノードの制約

特に 4 つ目の制約により，RedBlackTree がある程度の平衡性を持っていることが確認できる．挿入・削除・検索といった操作は最悪のケースでは木の高さに比例した計算時間をようするが，RedBlackTree は黒の深さで木の高さが平衡しているため，最悪のケースにおける計算量が，データの挿入・削除・検索のいずれにおいても，データ構造のうちで最善のものの一つであるという利点がある．また，木の深さがバランスしていない場合は，親と叔父の色 を見て木を回転させるという操作が必要になる．

2.5　Agda

Agda とは，定理証明支援器であり，関数型言語である．Agda では依存型という型システムを持ち，Curry-Howard 同型対応により，命題と型付ラムダ計算が 1 対 1 で対応するため，Agda で書かれたプログラムの証明を書くことができる．本節では，本論文を読み進めるにあたって必要な Agda の実装について，ソースコードを例に出しながら解説していく．

2.5.1 Data 型の実装

Data 型とは，複数の要素をもつ集合である．集合が持つ要素ごとに実装を施す必要がある．例として自然数の実装を考える．自然数とは，ペアノの公理から 0 と後者関数 (Successor) から成り立つ．0 とは一番初めの数であり，前者が存在しない数，後者関数とは前者の数に 1 を足したものを出力する関数を意味する．したがって，すべての自然数は，0 と自然数に 1 を足したもので定義できる．これを Agda で実装するとソースコード 2.1 のようになる．

ソースコード 2.1 Agda における Data 型の実装

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

1 行目では ℕ という名前の型を定義している．ℕ は 2 つの要素を持っていることが見て取れる．

2 行目から 3 行目では要素それぞれの実装をおこなっている．3 行目の ℕ → ℕ は，型 ℕ を一つ受け取り，型 ℕ を返すという意味になっている．これは後者関数の実装である．

以上から，ペアノの公理に基づく自然数を Agda で実装できていることが確認できる．

2.5.2 関数の実装

Agda でも他の言語同様に関数を実装できる．今回は例として足し算の実装を挙げ解説していく．足し算はペアノの公理により，n+0 と n+suc を示すことで定義できる．

ソースコード 2.2 Agda における足し算の実装

```
1 plus : ℕ → ℕ → ℕ
2 plus n zero = n
3 plus n (suc m) = suc (plus n m)
4
5 -- plus 2 (suc 2) = suc (plus 2 2)
6 -- * plus 2 2 = plus 2 (suc 1) ....
7 -- plus 2 (suc 2) = suc (suc (suc (plus 2 0)))
```

ソースコード 2.2 について解説していく．

Agda では，-を 2 つ書くことでコメントアウトが書ける．ここでは，plus 関数を展開する様子を記述している．

1 行目では関数を定義している．Agda で関数を定義する際には:(セミコロン) の前に関数名を記述し，後ろに受け取る型と返す型を記述する．この場合では plus という名前の関数は，型 $\mathbb{N}$ を 2 つ受け取り，型 $\mathbb{N}$ を返すという意味になる．

2 行目と 3 行目では，それぞれの場合における動作を記述している．今回の例では，型 $\mathbb{N}$ の定義により，足す数が 0 である場合と suc である場合の 2 パターンが存在する．これらをそれぞれ記述する必要がある．

2 行目では，足す数が 0 である場合を記述しており，n+0 は n であるため，そのまま n を返す．

3 行目では，足す数が suc である場合を記述している．ここでは，suc m の suc を外に出すことで，再帰的に足し算を記述している．例えば 2+3 を考えてみる．2+3 はこの場合 plus 2 (suc 2) = suc (plus 2 2) と書くことができる．次に，plus 2 2 の部分は plus 2 (suc 1) と書ける．最終的には，plus 2 (suc 2) = suc (suc (suc (plus 2 + 0))) となり，2 行目の n+0 の形に帰着する．したがって，n+suc は停止性を示すことができ，再帰的に定義できることが確認できる．

また，Agda では_(アンダースコア) を用いることで引数を受け取ることができ，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．その例をソースコード 2.3 に示す．

ソースコード 2.3 Agd における二項演算子の実装

```
1 _+_ : ℕ → ℕ → ℕ
2 n + zero = n
3 n + (suc m) = suc (plus n m)
```

二項演算子を実装することで，後述する _∧_ などを用いたコードの可読性を向上させることができる．

2.5.3 場合分けの書き方

Agda では，with 文を使用することで簡単に場合分けを行うことができる．これにより，様々な場合に応じた柔軟なコーディングが可能になっている．今回は，二つの自然数を比較し，大きいほうを出力する関数を解説する．

ソースコード 2.4 Agda における場合分けの書き方

```
1 Compare : ℕ → ℕ → ℕ
```

```
2 Compare x y with <-cmp x y
3 Compare x y | tri< a ¬ b ¬ c = y
4 ... | tri≃ ¬ a b ¬ c = x
5 ... | tri> ¬ a ¬ b c = x
```

ソースコード 2.4 を解説する．

1 行目では関数の実装を行っている． 2 つの自然数を受け取り，自然数を返すという意味になる．

2 行目では，with 文を使用することで場合分けを行っている．二つの自然数を比較する際には，普通 $<-cmp$ 文を使用する．まとめると，受け取った二つの自然数 x,y を比較するという意味になっている．

3 行目から 5 行目では，比較した結果の場合分けに応じた処理を記述している．tri の後に続く a , b , c にはそれぞれ，$<, \equiv, >$ の意味がある．¬ には否定の意味があることから，場合分けに応じて a,b,c のうち 1 つだけが真になるようになっている．また，... | と記述することで，入力が同じ場合に記述を省略することができる．3 行目から 5 行目の入力はすべて同じであり，2 通りの書き方ができることを示している．

3 行目では，a が真であり， $x<y$ の場合の処理を記述すればよい．したがって，大きいほうを返せばいいので y を返す．

4 行目では，b が真であり，$x \equiv y$ の場合の処理を記述すればよい．したがって，どちらを返しても同じなので x を返す．

5 行目では，c が真であり，$x>y$ の場合の処理を記述すればよい．したがって，大きいほうを返せばよいので x を返す．

2.6 GearsAgda

GearsAgda とは，GearsOS の一種であり，CbC の概念を取り入れた Agda の記述方法である．これを用いることで，CbC に直接対応した柔軟な証明を行うことができる．CbC の仕様を満たしている GearsAgda で書かれた証明付きのコードは，CbC にコンパイルすることが可能である．そのため，GearsAgda で証明することができれば，直接的に CbC で実装することが可能である．

2.6.1 GearsAgda の記述方法

GearsAgda は CbC の概念を取り入れているため，再帰的処理が行えず，軽量継続を用いて処理を記述する必要がある．ここでは，2.5.2 で実装した足し算の関数を GearsAgda で記述し，比較しながら解説していく．

ソースコード 2.5 GearsAgda における足し算の実装

```
1  plus-case : {l : Level} {t : Set l} → ℕ → ℕ →
2          (next : ℕ → ℕ → t) → (exit : ℕ → t) → t
3  plus-case x y next exit with y
4  ... | zero = exit x
5  ... | (suc y) = next (suc x) y
6
7  {-# TERMINATING #-}
8  Loop-plus : {l : Level} {t : Set l} → ℕ → ℕ →
9          (exit : ℕ → t) → t
10 Loop-plus x y exit = plus-case x y (λ x y → Loop-plus x y exit) exit
11
12 plus-gears : ℕ → ℕ → ℕ
13 plus-gears x y = Loop-plus x y (λ x → x )
```

ソースコード 2.5 について解説する．

2.5.2 で実装した足し算のコードと比べ，記述量が約 3 倍になっていることが確認できる．これは，GearsAgda が再帰処理を行えず，自己呼び出しによるループのみで足し算を行う必要があり，場合分け部分とループ部分を構成する必要があるからである．

1 行目と 2 行目は関数の定義部分になっており，plus-case 関数では，次の遷移先を決める処理が記述されている．自然数 2 つと next,exit を受け取り，t を返す．ここでいう t とは不定の型であり，t を返すためには next か exit を必ず呼び出す必要がある．これが CbC の goto 文を用いた軽量継続に当たり，GearsAgda では，next か exit を必ず呼び出さなければ型が一致しないため，継続によって関数が機能していることが確認できる．

4 行目と 5 行目では，それぞれの遷移先における条件を記述している．with 文により y が zero である場合，ループを exit を呼び出して x を返す．y が (suc y) の場合，next を呼び出して，y から 1 引き，x に 1 を足す．これらにより，next を繰り返すたびに y が減り x が増え，y が zero になった時にループが停止し，x を返すことで足し算の結果となることが確認できる．

8 行目から 10 行目では簡単なループを記述している．x,y と exit を受け取り，plus-case を呼び出している．plus-case に渡す引数の next に Loop-plus 自身を渡すことで，ループを構築している．Loop-plus の最後に返す型は t であり，Loop-plus の定義部分により，exit を経由することでしか t を得ることができないため，exit になるまで next をループすし，exit を必ず経由してからループを抜けることが確認できる．

12 行目から 13 行目では，関数の実行部分を記述している．Loop-plus を呼び出すことで足し算が開始される．Loop-plus は x y と exit を引数に持つ．このとき exit にある ($\lambda\ x \rightarrow x$) は受け取った値をそのまま返すという意味になっており，exit した際に出た x をそのまま返すという意味になっている．

以上により，GearsAgda で記述された足し算を実装することができた．GearsAgda での実装は，通常の実装に比べ煩雑になってしまう．しかし，CbC の概念に寄り添った形である GearsAgda は，CbC へとコンパイルすることが容易に可能だと考えられ，GearsOS の信頼性を向上と密接に関係している．

第 3 章

提案手法

本章では，RedBlackTree を GearsAgda で証明付き実装するため，以下の手順を提案する．

1. BinarySearchTree を応用した，RedBlackTree の実装
2. Invariant を見つける

3.1　BinarySearchTree を応用した，RedBlackTree の実装

BinarySearchTree と RedBlackTree は共通点が多いため，比較的簡単である BinarySearchTree を実装し，それを RedBlackTree の実装に応用する．変更点としては，ノードそれぞれが色を持つこと，黒の深さでバランスすること，バランスするときに木が回転することなどが考えられる．

3.2　Invariant を見つける

Red Black Tree の正しさを GearsAgda で証明するためには，RedBlackTree の Invariant（ループや再帰で不変な条件/命題）を見つけることが有効であると考えられる．例えば，あるツリーに対して RedBlackTree であるための Invariant を渡す．その後，様々な操作を行った後にツリーが Invariant を持っていることを証明することで，操作後のツリーは RedBlackTree であるといえる．したがって，GearsAgda で RedBlackTree を証明付き実装するためには，Invariant をみつけることが不可欠である．

第 4 章

RedBlackTree と Invariant の実装

本章では，RedBlackTree とその Invariant を GearsAgda を用いて実装していく．

4.1　RedBlackTree の基本的な実装

第 3 章 1 節で説明した通り，BinarySearchTree をもとに RedBlackTree の基本的な部分を実装していく．BinarySearchTree は leaf と node の二つの要素から構成されており，node には以下の 4 つの要素を持つ．

1. key : 自然数であり，この値によって木構造を決定する．
2. value : node に格納する値である．型は任意で決めることができる．
3. left-child : 該当ノードから見た左側に持つ子供ノード．制約により，親の Key の値よりも小さい Key を持つ．
4. right-child : 該当ノードから見た右側に持つ子供ノード．制約により，親の Key の値よりも大きい Key を持つ．

これらの要素を含む Data 型を GearsAgda で記述することで，基本的な BinarySearchTree の構造を示すことができる．

ソースコード 4.1　BinarySearchTree の基本的な実装

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

ソースコード 4.1 について解説する．

1 行目では Data 型の実装に基づき，bt 型を定義している．ここに書かれている (A : Set n) とは，任意の型を 1 つ受け取るという意味になり，value の持つ型を指定することができる．前述した通り，BinarySearchTree は leaf と node の二つの要素からなるので，直下に要素ごとの処理を記述する必要がある．

2 行目では，leaf を定義している．leaf は Key も Value も持たないため，ただ bt A を返すだけ と記述するだけになることが確認できる．

3 行目と 4 行目では node について定義している．node は必要な 4 つの要素を入力に受け取り，bt 型を返すという実装になっていることが確認できる．

BinarySearchTree の基本的な実装を元に色の概念を付け加えることで，RedBlackTree の基本的な実装を行うことができる．ここでは，Value に色を組で渡すことで色の概念を付け加える．

ソースコード 4.2　RedBlackTree の基本的な実装

```
1 data Color : Set where
2     Red : Color
3     Black : Color
4
5 RBTreeTest : bt (Color ∧ ℕ)
6 RBTreeTest = node 8 《 Black , 200 》 (node 5 《 Red , 100 》 (_) (_))
      (node 10 《 Red , 300 》 (_) (_))
```

ソースコード 4.2 について解説する．

1 行目から 3 行目では，色の概念を追加するために Color という名前の Data 型を定義している．要素には Red と Black の二つがあり，二つともただ Color を返すだけになっている．

5 行目では，実際に簡単な RedBlackTree を記述している．前述した通り，bt に与える Value の値の部分を色と自然数の組にして渡している．今回は例として自然数を用いているが，ここの型は任意に決めることができる．

6 行目では，RBtreeTest の処理内容を記述している．ここでは例として，第 2 章 4 節で説明した図 2.2 のルートノードから見た木構造を実装している．BinarySearchTree での value に相当する部分が，色と自然数の組として記述されていることが確認できる．また，左の子に Key が 5 で赤の子，右の子に Key が 10 で赤の子が記述されており，これは第 2 章 4 節で説明した図 2.2 の木構造と一致する．Agda では，_ (アンダースコア) を使用することでコンパイラが推論し，記述を省略できる．本来，親ノードから見た孫ノードが入るが，木構造全体を記述すると冗長になるため省略している．

以上により，GearsAgda にて RedBlackTree を記述し木構造を実装できることが確認できた．

4.2　Invariant の実装

第 3 章 2 節で説明した通り，本研究では Invariant を中心に証明を進めていく．本節では，特に重要になる以下の Invariant についての実装を解説していく．

1. RBtreeInvariant : 対象の木が RedBlackTree であることを示す
2. stackInvariant : 辿った木を積む stack が辿った順に構成されていることを示す

これらの Invariant は，RedBlackTree における操作の正当性を示すのに重要な Invariant となっている．

4.2.1　RBtreeInvariant の実装

RBtreeInvariant は，対象の木が RedBlackTree であることを示す Invariant である．これは，RedBlackTree の可能な値全部の集合であり，RedBlackTree の表示的意味論そのものになっている．つまり，全体の集合を型として用意し，この型に該当することを示すことで，対象の木は RedBlackTree であるということを証明することができる．RedBlackTree の取る可能な値は 8 種類あり，それぞれについての実装を記述する必要がある．

ソースコード 4.3　RBtreeInvariant の実装

```
1  data RBtreeInvariant {n : Level} {A : Set n} : (tree : bt (Color ∧ A
       )) → Set n where
2      rb-leaf : RBtreeInvariant leaf
3      rb-single : {c : Color} → (key : ℕ) → (value : A) →
           RBtreeInvariant (node key ⟪ c , value ⟫ leaf leaf)
4      rb-right-red : {key key₁ : ℕ} → {value value₁ : A}
5         → {t t₁ : bt (Color ∧ A)} → key < key₁
6         → black-depth t ≡ black-depth t₁
7         → RBtreeInvariant (node key₁ ⟪ Black , value₁ ⟫ t t₁)
8         → RBtreeInvariant (node key ⟪ Red , value ⟫ leaf (node key₁ ⟪
              Black , value₁ ⟫ t t₁))
9      rb-right-black : ?
10     rb-left-red : ?
11     rb-left-black : {key key₁ : ℕ} → {value value₁ : A}
```

```
12        → {t t₁ : bt (Color ∧ A)} → key < key₁ → {c : Color}
13        → black-depth t ≡ black-depth t₁
14        → RBtreeInvariant (node key₁ ⟪ c , value₁ ⟫ t t₁)
15        → RBtreeInvariant (node key ⟪ Black , value ⟫ (node key₁ ⟪ c ,
              value₁ ⟫ t t₁) leaf)
16    rb-node-red : {key key₁ key₂ : ℕ} → {value value₁ value₂ : A} →
          {t₁ t₂ t₃ t₄ : bt (Color ∧ A)}
17        → key < key₁ → key₁ < key₂
18        → black-depth (node key ⟪ Black , value ⟫ t₁ t₂) ≡ black-
              depth (node key₂ ⟪ Black , value₂ ⟫ t₃ t₄)
19        → RBtreeInvariant (node key ⟪ Black , value ⟫ t₁ t₂)
20        → RBtreeInvariant (node key₂ ⟪ Black , value₂ ⟫ t₃ t₄)
21        → RBtreeInvariant (node key₁ ⟪ Red , value₁ ⟫ (node key ⟪
              Black , value ⟫ t₁ t₂) (node key₂ ⟪ Black , value₂ ⟫ t₃ t
              ₄))
22    rb-node-black : ?
```

ソースコード 4.3 を解説していく．

agda では「？」と記述することで，該当箇所の記述を省略することができる．ここでは，全体を記述すると冗長になってしまうため，最低限の説明が必要な部分を除き，記述を省略している．

1 行目では，RBtreeIvariant 型を定義している．入力として，RedBlackTree を受け取る．

2 行目では rb-leaf を定義しており，これは leaf の Invariant を示している．RBtreeInvariant の引数に leaf を渡すことで実装している．

3 行目では rb-single を定義しており，これはノードの左右に leaf がついている木の最小単位のようなノードを示している．RBtreeInvaritant の引数に (node key⟪c , value ⟫ leaf leaf) を渡すことで実装している．

4 行目から 8 行目までは，rb-right-red を定義しており，これはノードの右側に子ノード，左側には leaf があるようなノードの Invariant を示している．また，親ノードは赤と黒の 2 パターン存在するので，それぞれ別に定義する必要があり，4 行目から 8 行目は赤の親ノード，9 行目は黒の親ノードを定義している．

5 行目では，$key<key_1$ を入力として受けるように記述されている．これは，この Invariant を示すためには，親子間の Key の大小関係を必ず明示しなければいけないという意味になり，Key の制約を満たす場合のみを通すことから，この Invariant を示すことで，Key の正当性を示すことが可能になる．

6 行目では，親ノードからみた孫ノードの黒の深さが同じであることを入力として受けている．これは Key の大小関係と同じように，この Invariant を示すことで，黒の深さの正当性を示すことが可能になる．

7 行目では，子ノードの色が黒である Invariant を入力として受けている．rb-right-red では，親は赤ノードであるため，必ず子ノードは黒であるという条件を含めていることが確認できる．

11 行目から 15 行目では，rb-left-black を定義しており，これはノードの右側に leaf，左側には子ノードがあるようなノードの Invariant を示している．これは，前述した rb-right-red,rb-right-black の左右の子を入れ替えたものであり，本質的な構造は似ているため，説明を省略する．ただし，親ノードが黒のとき，子ノードは赤と黒の 2 パターンが存在し，どちらの色でも可能という意味で《c , value 》と記述されている．

16 行目から 21 行目では，rb-node-red を定義しており，これは左右それぞれに子ノードを持つようなノードの Invariant を示している．また，親ノードは赤と黒の 2 パターン存在するので，それぞれ別に定義する必要があり，16 行目から 21 行目は赤の親ノード，22 行目からは黒の親ノードを定義している．

以上のように，RedBlackTree の制約を Invariant の入力に含めることで，これを示した際に RedBlackTree の正当性を証明することができる．

4.2.2　stackInvariant の実装

GearsOS のファイルシステムとデータベースでは，非破壊的な RedBlackTree を採用することが予定されている．非破壊的とは，操作を行う際に木を上書きせず，コピーして木を構築することを意味し，読み込みと書き込みを同時に行うことができるなどの利点がある．非破壊的な RedBlackTree を構築するためには stack を使う必要がある．

stackInvariant とは，木の操作を行う際に辿った木を積む stack が，辿った順に構成されていることを示す Invariant である．つまり，この Invariant を示すことで，対象の stack が積んできた木の履歴を見れるような形になっている．これを実装することにより，木をバランスさせる際などに stack を見て，ひとつ前の木に戻り操作を行うなどの実装が可能になる．

ソースコード 4.4　stackInvariant の実装

```
1 data stackInvariant {n : Level} {A : Set n} (key : ℕ) : (top orig :
      bt A) → (stack : List (bt A)) → Set n where
2     s-nil : {tree0 : bt A} → stackInvariant key tree0 tree0 (tree0 ::
```

```
        [])
3   s-right : (tree tree0 tree₁ : bt A) → {key₁ : ℕ } → {v1 : A }
        → {st : List (bt A)}
4       → key₁ < key
5       → stackInvariant key (node key₁ v1 tree₁ tree) tree0 st
6       → stackInvariant key tree tree0 (tree :: st)
7   s-left : (tree₁ tree0 tree : bt A) → {key₁ : ℕ } → {v1 : A } →
         {st : List (bt A)}
8       → key < key₁
9       → stackInvariant key (node key₁ v1 tree₁ tree) tree0 st
10      → stackInvariant key tree₁ tree0 (tree₁ :: st)
```

ソースコード 4.4 について解説する．

1 行目では，stackInvariant の型を定義している．入力は，key,top,orig,stack の 4 つを受け取る．top と orig は両方とも bt 型であり，top は stack の一番上にある木，orig は一番下にある木を意味する．GearsAgda において，stack は List 型を用いて定義する．List 型は，任意の型を中に持ち，List (bt A) の書くことで木を List の中に持たせることができる．List 型は， (tree0 :: []) のように記述することができ，「::」で要素同士を並べていく．「[]」は空であることを意味し，List の終わりであることを示す．

2 行目では，stack の最小単位である s-nil を定義している．これは，top と orig に受け取る木が同じであるといった特徴がある．

3 行目から 6 行目では，s-right を定義している． 5 行目を見ると，入力で stackInvariant を受け取っている．これは，一個前の木で示された stackInvariant であり，s-right では，この木の右側の子を stack に積んだことを示す． 4 行目を見ると，key の大小関係を入力として受け取っており，対象の Key よりも一個前の key_1 が小さく，木の順序性を保っていることが確認できる． 6 行目では，一個前の stackInvariant が示している stack の st に，新たな木を積んだことを示す stack を (tree :: st) と表していることが確認できる．

7 行目から 10 行目では，s-left を定義している．s-left では右側ではなく左側の子を stack に積んだことを示す Invariant になっている．その他の大まかな実装は s-right と同じであるため，説明を省略する．

以上により，一番下の木 orig から，対象の木 top までを辿った木を積む stack の正当性を示す Invariant を定義することができた．

第 5 章

findRBT の実行

ここでは，RedBlackTree の基本的な操作である Find の実装について説明していく．find は，指定された key を持つ node を探す操作になる．

5.1　findRBT の実装

find の操作は，第 4 章で実装した Invariant を保有しながら操作を行うことで，木の正当性を証明することができる．また，GearsAgda での実装となるため，第 2 章 6 節で説明した通り，軽量継続を用いた実装を行う必要がある．

軽量継続は Loop を用いて行うので，この関数の停止条件を決める必要がある．前述したソースコード 2.5 の足し算の例では，y が 0 になることで exit し，関数が停止するよう定義してある．ここでは，対象の key を持つノードが見つからないまま leaf に達してしまった時と対象の key を持つノードが見つかった時の 2 つを停止条件とする．

また,findRBT にはさまざまな関数や記法が使われているため, それらの機能を表 5.1 にまとめる．

表 5.1 findRBT に使用される関数

名前	機能
RBtreeLeftDown	親ノードからみた左の子の RBtreeInvariant を得る関数
RBtreeRightDown	親ノードからみた右の子の RBtreeInvariant を得る関数
case1	∨ で与えられた要素の左の値を取り出すことができる.
case2	∨ で与えられた要素の右の値を取り出すことができる.
proj1	∧ で与えられた要素の左の値を取り出すことができる.
proj2	∧ で与えられた要素の右の値を取り出すことができる.
depth-1<	自然数の大小関係を証明する関数.3 パターン存在する.

次に, 実装したソースコードを以下に示す.

ソースコード 5.1 findRBT の実装

```
1  findRBT : {n m : Level} {A : Set n} {t : Set m}
2       → (key : ℕ)
3       → (tree tree0 : bt (Color ∧ A) )
4       → (stack : List (bt (Color ∧ A)))
5       → RBtreeInvariant tree ∧ stackInvariant key tree tree0 stack
6       → (next : (tree1 : bt (Color ∧ A) )
7            → (stack : List (bt (Color ∧ A)))
8            → RBtreeInvariant tree1 ∧ stackInvariant key tree1 tree0
                 stack
9            → bt-depth tree1 < bt-depth tree
10           → t )
11      → (exit : (tree1 : bt (Color ∧ A))
12           → (stack : List (bt (Color ∧ A)))
13           → RBtreeInvariant tree1 ∧ stackInvariant key tree1 tree0
                 stack
14           → (tree1 ≡ leaf ) ∨ ( node-key tree1 ≡ just key )
15           → t )
16      → t
17 findRBT key leaf tree0 stack inv next exit
18  = exit leaf stack inv (case1 refl)
19 findRBT key (node key₁ value left right) tree0 stack inv next exit with
       <-cmp key key₁
20 findRBT key (node key₁ value left right) tree0 stack inv next exit |
```

```
        tri< a ¬ b ¬ c
21   = next left (left :: stack) ⟪ RBtreeLeftDown left right (_∧ _.proj1
         inv) , s-left a (_∧ _.proj2 inv) ⟫ depth-1<
22 findRBT key n tree0 stack inv _ exit | tri≃ ¬ a refl ¬ c
23   = exit n stack inv (case2 refl)
24 findRBT key (node key₁ value left right) tree0 stack inv next exit |
        tri> ¬ a ¬ b c
25   = next right (right :: stack) ⟪ RBtreeRightDown left right (_∧ _.
         proj1 inv) , s-right c (_∧ _.proj2 inv) ⟫ depth-2<
```

ソースコード 5.1 について解説する．

1 行目から 16 行目では，findRBT の関数を定義している．findRBT では，7 つの入力を受け取り，型 t を返すという意味になっていることがわかる．6 行目と 11 行目を見ると，next と exit が定義されていることから，GearsAgda を用いた軽量継続を行っていることが確認できる．

5 行目では，Invariant を受け取っている．ここでは，RBtreeInvariant と stackInvariant が組となって渡されており，top である tree についての Invariant であることが確認できる．

6 行目と 11 行目では，軽量継続の next と exit についての記述がある．両者とも，木と stack とそれぞれの Invariant の組を受け取るところまでは同じである．違いがあるのは，9 行目と 14 行目である．next の 9 行目を見ると，木の高さを比較しており，その段階で見ている木よりも次の木の高さが低いことを示している．これは，次の木に進むにつれて leaf に近づいているという証明を受け取るという意味になる．次に，exit の 14 行目を見ると停止条件が指定されていることが確認できる．前述した通り，その段階で見ている木が leaf の場合もしくは，見ている木の key が一致する場合が停止条件になっていることが確認できる．これらは，or で記述されており，どちらかを満たす場合停止するという意味になる．

17 行目から 25 行目では，実際の処理の内容について記述されている．これらは，4 つの場合分けがなされており，1 つは木が leaf だった場合，残り 3 つはその段階で見ている木の key と指定された key の大小関係による場合分けである．

17 行目では，見ている木が leaf である場合の処理が記述されている．findRBT の 2 つ目の引数を見ると leaf になっていることが確認できる．停止条件により，leaf の場合は exit に遷移する．case1 と書くことで or の一つ目の条件を抽出することができ，refl はそれらが同値であることを示すことから，それが満たされていることを証明している．

19 行目では，with 文を使用することで場合分けをし，key の大小関係によって次の遷移

先を決定していることが確認できる．20,21 行目と 24,25 行目での操作は，左右の差しかないため，説明を省略する．

20 行目と 21 行目では，指定された key がその段階で見ている key_1 より小さい時の処理が記述されている．つまり，指定された key を持つノードは key の制約により，必ず左側の木に存在していることになり，次の遷移先が左の子供になることがわかる．21 行目を見ると，次の遷移先が left になっており，stack の top に left が積まれていることが確認できる．Invariant の組は，RBtreeLeftDown 関数と s-left から導出している．どちらも，left に遷移する際の前の状態（現在見ている状態）の Invariant が入力として必要である．これは，すでに inv として入力で受け取っているため (_∧ _.proj1 inv) のように記述することで，組の左側を入力として渡している．

22 行目と 23 行目では，見ている木の key が一致する場合について記述されている．これは停止条件であることから，exit に遷移する．23 行目を見ると，見ている木と stack をそのまま返し，停止条件の 2 つ目を抽出し，それが満たされていることを証明している．

以上により，GearsAgda による Invariant を保有した Find の操作を実装することができた．

5.2 findRBT の実行方法

findRBT を実行するためには，第 2 章 6 節で述べた通り GearsAgda の記述方法に則って実装していく必要がある．ここでは，findRBT をテスト実行する findTest 関数をループコネクターでつなげ，軽量継続を用いた実行を可能にする．すべてのコードを解説すると冗長になってしまうことから，理解するために必要な関数や記述方法を表 5.2 にまとめる．

表 5.2 findTest に使用される関数

名前	機能
TerminatingLoopS	様々な関数を接続するループコネクターの役割を担う．
testRBTree0 testRBI0	図 2.2 で示した RedBlackTree を GearsAgda 上で表した関数． testRBTree0 の RBtreeInvariant を意味する．
result	record 型である，C 言語でいう構造体である．find の結果を格納するために使用す

次に，実装したソースコードを以下に示す．

ソースコード 5.2 findTest の実装

```
1  findTest : {n m : Level} {A : Set n } {t : Set m }
2    → (key : ℕ)
3    → (tree0 : bt (Color ∧ A))
4    → RBtreeInvariant tree0
5    → (exit : (tree1 : bt (Color ∧ A))
6         → (stack : List (bt (Color ∧ A)))
7         → RBtreeInvariant tree1 ∧ stackInvariant key tree1 tree0
              stack
8         → (tree1 ≡ leaf ) ∨ ( node-key tree1 ≡ just key ) → t )
9    → t
10 findTest {n} {m} {A} {t} k tr0 rb0 exit = TerminatingLoopS (bt (
       Color ∧ A) ∧ List (bt (Color ∧ A))) {λ p → RBtreeInvariant (
       proj1 p) ∧ stackInvariant k (proj1 p) tr0 (proj2 p) } (λ p →
       bt-depth (proj1 p)) ⟪ tr0 , tr0 :: [] ⟫ ⟪ rb0 , s-nil ⟫
11         $ λ p P loop → findRBT k (proj1 p) tr0 (proj2 p) P (λ t s
               P1 lt → loop ⟪ t , s ⟫ P1 lt )
12         $ λ tr1 st P2 0 → exit tr1 st P2 0
13
14 findRBTreeTest : result
15 findRBTreeTest = findTest 14 testRBTree0 testRBI0
16         $ λ tr s P 0 → (record {tree = tr ; stack = s ; ti = (proj1
               P) ; si = (proj2 P) ; exit = 0})
```

ソースコード 5.2 について解説する．

5 行目を見ると，exit の条件が指定されている．これは，FindRBT と同じものになっている．TerminatingLoopS には停止条件が含まれていないためここで記述する必要がある．

10 行目を見ると，TerminatingLoopS が呼び出されていることがわかる．ここでの操作は複雑であるが, Invariant として RBtreeInvariant と stackInvariant を渡しており, Invariant を保有しながら Loop していることが確認できる．

11 行目では，ループの next 部分を記述しており，findRBT を遷移先に指定していることが確認できる．

12 行目では，exit を呼び出しループが終了した際の処理を記述している．ここでは，findRBT と findTest の終了条件を同じものに実装しているので，受け取った値をそのまま exit の引数として与えることができる．

14 行目では，findRBTreeTest を実行する関数を定義している．ここでは，result 型を指定することで，関数の結果を構造体のような形で持つことができる．

15 行目では，findTest 関数を呼び出し，実際に検索する key の値と木と木の Invariant を引数として渡している．この例では，key の値が 14 のノードを testRBTree0 の中から検索するという意味になる．

16 行目では，受け取った値を result 型に格納している．ここでは，検索結果の木，stack，RBtreeInvariant，stackInvariant，停止条件を保持している．

以上により，findTest を実装することができる．

5.3 実行結果

ここでは，前述した findTest の実行結果を述べ，解説していく．

実行結果 5.3 Key の値 14 の findTest 実行結果

```
1  record
2  { tree = node 14 《 Red , 1400 》 leaf leaf
3  ; stack =
4      node 14 《 Red , 1400 》 leaf leaf
5      :: node 15 《 Black , 1500 》 (node 14 《 Red , 1400 》 leaf leaf)
           leaf
6      :: node 10 《 Red , 1000 》 leaf (node 15 《 Black , 1500 》 (node
           14 《 Red , 1400 》 leaf leaf) leaf)
7      :: node 8 《 Black , 800 》
8      (node 5 《 Red , 500 》 (node 2 《 Black , 200 》 leaf leaf)
9       (node 6 《 Black , 600 》 leaf leaf))
10     (node 10 《 Red , 1000 》 leaf
11      (node 15 《 Black , 1500 》 (node 14 《 Red , 1400 》 leaf leaf)
            leaf))
12     :: []
13 ; ti = rb-single 14 1400
14 ; si =
15     s-left
16     (s≤s(s≤s (s≤s(s≤s(s≤s (s≤s(s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s
           ≤s (s≤s z≤ n))))))))))))))
17     (s-right (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s
           z≤ n))))))))))))
18      (s-right (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s z≤ n
```

```
          ))))))))) s-nil))
19 ; exit = case2 refl
20 }
```

実行結果 5.3 について解説していく．

1 行目を見ると，record と書かれており，record 型で実行結果を受け取っていることがわかる．2 行目からが実行結果の内容になっており，それぞれ result 型の要素を見ることができる．

2 行目を見ると，指定した key である 14 の値を持つ node を検索することに成功していることが確認できる．

3 行目から 12 行目では，key が 14 である node を見つけるまでに辿った木が stack に保存されていることが確認できる．これにより，非破壊的な RedBlackTree の操作を行うことができる．

13 行目では，検索結果である node の RBtreeInvariant を確認することができる．これにより，検索結果の木が RedBlackTree であることを証明することができる．また，操作を行った後でも，RedBlackTree であることを証明することができ，この Invariant から必要な要素を導出することも可能になる．

14 行目から 18 行目では，stackInvariant を確認することができる．これは，stack にどの様に木を積んだかを確認することができ，stack の木の順序性を証明することができる．この例だと，stack の始まりから，右，右，左の順番で辿ったという証明を持っていることになる．key の大小を比較を s≤s のような形で記述している．

19 行目を見ると，case2 refl となっており，key が一致したことでループが停止したことが確認できる．

次に，今回の木に存在しない値である 1 を指定して実行してみる．

実行結果 5.4　Key の値 1 の findTest 実行結果

```
1 record
2 { tree = leaf
3 ; stack =
4     leaf
5     :: node 2 ⟪ Black , 200 ⟫ leaf leaf
6     :: node 5 ⟪ Red , 500 ⟫ (node 2 ⟪ Black , 200 ⟫ leaf leaf)
7     (node 6 ⟪ Black , 600 ⟫ leaf leaf)
8     :: node 8 ⟪ Black , 800 ⟫
9     (node 5 ⟪ Red , 500 ⟫ (node 2 ⟪ Black , 200 ⟫ leaf leaf)
```

```
10      (node 6 《 Black , 600 》 leaf leaf))
11     (node 10 《 Red , 1000 》 leaf
12      (node 15 《 Black , 1500 》 (node 14 《 Red , 1400 》 leaf leaf)
            leaf))
13     :: []
14 ; ti = rb-leaf
15 ; si =
16     s-left (s≤s (s≤s z≤ n))
17     (s-left (s≤s (s≤s z≤ n)) (s-left (s≤s (s≤s z≤ n)) s-nil))
18 ; exit = case1 refl
19 }
```

特に注目するべき点は，2 行目と 18 行目である．2 行目では key の値 1 を持つ node を見つけることができずに末端に到達したため，leaf が結果として帰ってきていることが確認できる．また，18 行目では case1 refl となっており，末端の leaf まで到達してしまったからループが停止したことが確認できる．

これにより，findRBT は指定した key が存在している場合その木の情報を返し，key が存在していない場合 leaf を返して停止することが確認できた．また，Invariant もこの情報の中に含まれており，返した木の正しさを証明しているが確認できる．これは，findRBT の実装内容を満たしており，GearsAgda において RedBlackTree の find 操作を証明付き実装できたことを示している．

第 6 章

まとめと今後の展望

本論文では，GearsAgda を用いたプログラムを数学的に証明する方法と RedBlackTree の正当性を Invariant を用いて証明する実装について述べた．GearsAgda を用いて再帰処理を使用しない記述を実装することで，GearsOS に使用されている CbC 言語に直接対応するような証明を書くことが可能になった．また，Invariant を定義し，それらを保有しながら処理を行うことにより，数学的な証明を用いて RedBlackTree の信頼性を向上させることができた．この Invariant からは，さまざまな要素を導出できることから，今後の研究において応用的な処理を記述する際にも役立つと考える．

今後の課題として，Insert，delete のなどの操作を実装することが挙げられる．現段階で証明されている操作は find しかなく，これでは実用的な RedBlackTree とは言えない．Insert の実装は大きな枠組みは既に完成しており，証明部分を記述していく段階である．しかし，Insert などの木構造を変化させる操作では，木がバランスする動作を記述する必要があり，子供から見た祖父や叔父の色を見ながら場合わけをする必要がある．これにより，場合わけの数が多く実装が難しくなっているのが現状である．Insert を実装することができれば，似たようなアルゴリズムで delete を実装できるため，まずは Insert を簡単に記述する方法を探しつつ，着実に実装を進めていく必要がある．

今後の展望として，GearsAgda のコードを CbC のコードに変換することが挙げられる．GearsAgda は CbC に直接対応した記述方法であるため，理論上コンパイルすることが可能である．CbC 言語は C 言語とアセンブラの中間に位置しており，コーディングが困難であることから，GearsAgda を用いてコーディングできることが望ましい．これが可能になることで，CbC での記述が GearsAgda ベースで行えるようになり，信頼性のさらなる向上につながると考える．

謝辞

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

2024 年 2 月

森 逸汰

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] The agda wiki, http://wiki.portal.chalmers.se/agda/pmwiki. php, Accessed: 2023/09/10.

[4] Welcome to agda’ s documentation! — agda latest documentation, http : / / agda . readthedocs . io / en / latest/, Accessed: 2023/09/10.

[5] 上地悠斗, ”GearsAgda による Left Learning Red Black Tree の検証” 琉球大学工学部工学科知能情報コース 2020