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

Verification of red-black tree implemented in Continuation based C using Hoare Logic

学籍番号 175706H 氏名 上地 悠斗
指導教員 : 河野 真治

要 旨

当研究室にて Continuation based C[1] (以下 CbC) なる C 言語の下位言語に当たる言語を開発している。先行研究 [2] にて Floyd-Hoare Logic[3](以下 Hoare Logic) を用いてその検証を行なった。本稿では、先行研究にて実施されなかった CbC における RedBlackTree の検証を Hoare Logic を用いて検証することを目指す。

Abstract

We are developing a language called Continuation based C[1] (CbC), which is a lower language of the C. In a previous study[2] , Floyd-Hoare Logic[3] (Hoare Logic) was used to validate it. In this paper, we aim to use Hoare Logic to validate the red-black tree in CbC, which was not performed in previous studies.

1 研究目的

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されていること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挙げられる。当研究室では 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。この言語の信用性を検証したい。

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 RedBlack-Tree を記述し、Hoare Logic により検証を行うことを目指す。

2 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。

2.1 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。

2.2 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。

3 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 プログラムの読み方

本節では Agda の基本事項について ソースコード 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。

ソースコード 1: plus

```
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。

3.2 Data 型

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 ℕ の実装を ソースコード 2 に示す。

ソースコード 2: Nat

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

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。

3.3 Record 型

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 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。

ソースコード 3: 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 4 を以下に示す。

ソースコード 4: 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。

4 Code Gear に合わせた Agda

検証を行うために、Agda のコードも CbC に合わせて記述を行う必要がある。実際に以下がコードとなる。

CbC の特徴である、変数を継続して実行するために、必要な変数は Envc に格納する。コードに (next : Envc → t) と (exit : Envc → t) を引数に受け取っている。これで次の遷移先を引数として受け取る事で、実行を継続していることを示す。= の後は next Envc もしくは exit Envc となっていることからも実行を継続している事が分かる。

5 Hoare Logic

Hoare Logic1 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。

5.1 Hoare による Code Gear の検証

図 1 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。

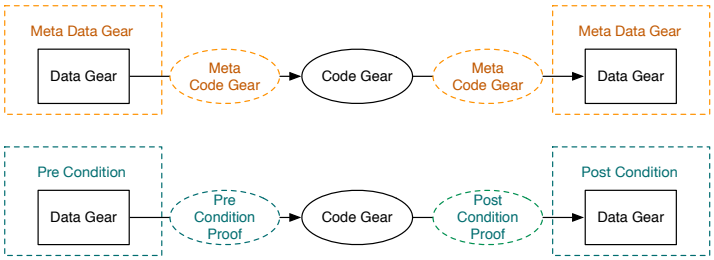

Figure 1: CodeGear、DataGear での Hoare Logic

6 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 2 を以下に示す。上記の定義を満たしていることが分かる。

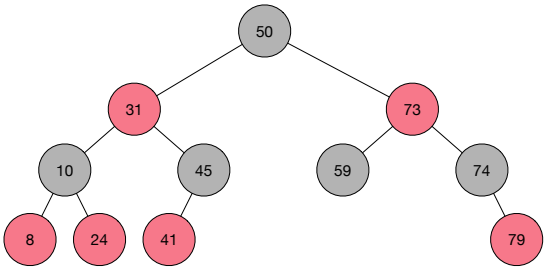

Figure 2: RedBlackTree の一例

7 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。

7.1 CbC 記法で書く agda

CbC プログラムの検証をするに当たり、Agda コードも CbC 記法で記述を行う。つまり継続渡しを用いて記述する必要がある。ソースコード 5 が例となるコードである。

ソースコード 5: Agda での CodeGear の例

```
1 plus : {l : Level} {t : Set l} → (x y : ℕ) → (next : ℕ -> t) -> t
2 plus x zero next = next x
3 plus x (suc y) next = plus (suc x) y next
4
5 -- plus 10 20
6 --   λ next → next 30
```

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。

7.2 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

8 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。

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.