CbC 言語によるプログラムの検証

145750B 氏名 外間政尊 指導教員：河野 真治

1 研究目的

ソフトウェアの信頼性を保証することは重要である。現在ソフトウェアの信頼性を保証する方法として代表的なものはモデル検査と、定理証明が存在している。モデル検査はソフトウェアの状態をすべて数え上げ、すべての状態で仕様が正しいことを確認する方法である。定理証明はソフトウェアが満たすべき仕様を論理式で記述し、その論理式が恒真であることを証明することである。

当研究室では検証と実装を同一の言語で行える Continuation based C (CbC) 言語を開発している。本研究では、検証や証明に直接使用できる言語として CbC を用いる。

本研究では CbC を用いて RedBlackTree を実装し、 Insert、Delete などの操作の際に RedBlackTree が常にその仕様を満たせているかを検証、証明する。

2 Countinuation based C (CbC)

Continuation based C (CbC) とは、当研究室で開発されているプログラミング言語である。CbC では OS や組み込みソフトウェアを主な対象としている。CbC は C 言語とほぼおなじ構文を持ち、よりアセンブラに近い形でプログラムを記述する。

CbC では C の関数の代わりに CodeSegment を用いて処理を記述する。CodeSegment は値を入力として受取り、出力を行う処理の単位で、それらの状態を goto で遷移して記述する。この goto による処理の遷移を継続と呼ぶ。CbC の CodeSegment を定義するには C とは少し異なり関数定義のの先頭に __code が付く。DataSegment は CodeSegment が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。CodeSegment の入力となる DataSegment は Input DataSegment　出力 を Input DataSegment は関数の引数として定義する。次の CodeSegment に処理を移す際は、goto の後に CodeSegment 名と Input DataDataSegment を指定する。

CbC ではこの継続処理にをメタ計算として定義されていて、実装や環境によって切り替えることできる。検証を行うメタ計算を定義することで、 CodeSegment の仕様を変更せずソフトウェアの検証を行う事ができる。例として CbC による Stack に対する操作のコード示す。

ソースコード 1: CbC による Stack

```
__code pushSingleLinkedStack(struct Single LinkedStack*
      stack,union Data* data, __code next(...)) {
    Element* element = new Element();
    element->next = stack->top;
    element->data = data;
    stack->top = element;
    goto next(...);
}
__code popSingleLinkedStack(struct Single LinkedStack*
     stack, __code next(union Data* data, ...)) {
    if (stack->top) {
        data = stack->top->data;
        stack->top = stack->top->next;
    } else {
        data = NULL;
    }
    goto next(data, ...);
}
```

このコードでは Stack に対する push と pop を定義している。

push や pop は必要があるときに外から呼ばれる。

push では element で新しい要素を作って、次の要素との関係、 push する要素を入れ、Stack の top を書き換えて次の CodeSegment に飛ぶ。

pop では Stack の top に data があればその data を next に入れ、次の CodeSegment に飛ぶ。top に data が無ければ NULL を next の Input Data に入れて次の CodeSegment に飛ぶ。

比嘉 (2016)[1] では CbC における CodeSegment 、DataSegment が部分型で定義できることが示されている。これより、 CbC で Functional に書かれたプログラムは等価な Agda のコードの置き換えることができる。本研究では CbC の代わりに等価な Agda のコードに変換することで証明を行う。

3 Agda

Agda[2] とは定理証明支援器であり依存型を関数プログラミング言語である。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あり、型の型や型を引数に取る関数、値を取って型を返す関数などが記述することができる。

CbC を Agda に変換する場合 DataSegment をレコード型、 CodeSegment は関数型となる。

前項で示した CbC で書かれた Stack の操作を Agda に変換したコードを示す。

ソースコード 2: Agda による Stack

```
record Stack {a t : Set} (stackImpl : Set) : Set  where
field
stack : stackImpl
push  : stackImpl −> a −> (stackImpl −> t) −> t
pop   : stackImpl −> (stackImpl −> Maybe a −> t)
      −> t
open Stack

pushStack : {a t : Set} −> Stack a −> a −> (Stack t
      −> t) −> t
pushStack {a} {t} s0 d next = (push s0) (stack s0) d (\
      s1 −> next (record {stack = s1} ))

popStack : {a t : Set} −> Stack a −> (Stack t −> t)
      −> t
popStack {a} {t} s0 next = (pop s0) (stack s0) (\s1 −>
      next s0)
```

Agda のコードで関数を定義するときは関数名、型を記述した後に関数本体を指定する。関数の型では → または `->` を使い定義する。

関数にはリテラルが存在し、関数を定義せずにその場で値を生成することもできる。これは ラムダ式と呼ばれ、`\arg1 arg2 -> function body` または λ arg1 arg2 → function body のように記述できる。上の例では pushStack の `\s1 -> next (record {stack = s1})` や、 popStack の `\s1 -> next s0` が ラムダ式である。

Agda のレコード型も存在する。定義をする際は record キーワードのあとにレコード名、型、 where の後に field キーワードを入れ、フィールド名 : 型名 と列挙する。レコードを構築する際は record キーワードの後に 内部に fileName = value の形で列挙していく。複数の値を列挙する際は ; で区切る。上の例では `record {stack = s1}` がそれにあたる。

このように CbC のコードを Agda に変換し、証明を行う。

4 RedBlackTree

RedBlackTree とは拡張された二分探索木で、木のバランスを取るための情報として各ノードにそれぞれ赤、黒の色を持っている。また、通常の二分探索木の条件に加えて、各ノードが赤か黒の色を持つ、ルートノードの色は黒、葉ノードの色は黒、赤ノードは二つの黒ノードを子として持つ、ルートから末端のノードへの経路に含まれる黒ノードの数は一定などの条件を持つ。

数値を要素に持つ RedBlackTree の例を以下の図 1 に示す。条件に示されている通り、ルートノードは黒であり、赤ノードは連続していない。加えて各最下位ノードへの経路に含まれる黒ノードの個数は全て 2 である。

本研究で検証する RedBlackTree は非破壊であり、一度構築した木構造は破壊される操作ごとに新しい木が生成される。非破壊である理由は並列実行時のデータ保存である。

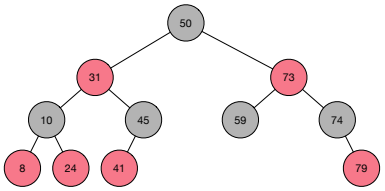

図 1: RedBlackTree の例

5 今後の課題

現段階では CbC で書かれた RedBlackTree の一部を Agda のコードに変換した。今後は CbC での RedBlackTree の Deletion 、Agda での証明を実装していく。また、依存型を導入することで CbC で自身を証明できるようにするなどの課題があるため、今後はこれらの課題に着手していく。

参考文献

[1] 比嘉 健太, 河野 真治. メタ計算を用いた Continuation based C の検証手法, 2016.

[2] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2017/10/24(Tue).

[3] 伊波 立樹, 東恩納 琢偉, 河野 真治. Code Gear 、Data Gear に基づく OS のプロトタイプ, 2016.

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

[5] Welcome to agda ' s documentation! — agda 2.6.0 documentation. `http://agda.readthedocs.io/en/latest/index.html`. Accessed: 2017/10/24(Tue).