GearsAgdaによる Red Black Tree の検証

森　逸汰 河野真治
琉球大学工学部
Itta Mori Shinji KONO
Faculty of Engineering, University of the Ryukyus

Abstract

GearsAgda では Hoare Logic を Agda 上に実装することよにより BinaryTreen の検証を可能のした。これを RedBlack Tree さらに、その木に対する並行実行の検証に拡張したい。invariant を拡張することにより、Red Black Tree の検証を行う。並行実行は、Model 検査を Geas Agda 上で形式化することにより実現する。これについて考察を行う。

In GearsAgda, verification of Binary Trees was made possible by implementing Hoare Logic on Agda. We want to extend this to the verification of RedBlack Trees and concurrent execution on these trees. By expanding the invariant, we can verify RedBlack Trees. Concurrent execution can be achieved by formalizing Model Checking on GearsAgda. Considerations need to be made for this.

1　検証された Red Black Tree の重要性

OS を含むアプリケーションの信頼性は、当然、OS のアプリケーションの両方の信頼性を上げる必要がある。信頼性を上げる手法には、テストやモデル検査などがあるが、数学的な証明を行うことが基本である。

最近では、定理証明を高階直観論理あるいは高階関数型言語で行うことができる。これは、Curry Howard 対応 [5] と呼ばれる命題と型の対応、そして、推論と関数の対応があるためである。実際には、Curry Howard 対応が、高階直観論理の意味を決めるのを主導していくことになる。例えば Coq [2] や Isabel HOL[7] そして、本論文で使用する Agda [6] が知られている。

一般的なプログラムにはループや再帰呼び出しがあり、メモリや並行実行などと関連している。メモリや並行実行あるいは I/O は、プログラムの関数としての性質とは直接は関係しないので、副作用と呼ばれることもある。これらを理論的に扱う手法として Monad がある。Monad [3, 9] は通常の関数にメタ情報を扱う構造を付加する方法になる。

OS を含むアプリケーションは、証明を意識した手法でプログラミングされるべきであり、一つの方法は、すべてをリストのような特定の型変数を含むデータ構造 (関手) として定義して、その map としてプログラムを記述することである。これにより圏論的な証明手法 (交換図) が使えるようになる。この方法は有効だが一般的とは言えない。例えば、バランス木の一種である Red Black Tree の insert 操作は map ではなくて、構造変化させる操作なので、その操作の正しさは関手による方法で書くことは自明にはならない。

このような場合に有効なのは invariant (ループや再帰で不変な条件/命題) を見つけることであり、古典的には Hoare Logic として知られている。この論文では、実際に、簡単な while program の証明と、二分木、そして Red Black Tree の invariant について考察する。

GearsAgda [12] は、Gears OS[11] に採用されている CbC (Continuation based C)[10] を Agda で記述する方法である。これを用いることにより、CbC に直接対応した Hoare Logic よりも柔軟な証明法を使うことができる。例えば、プログラムの停止性は、CbC の実行単位である Code Gear に対して、ループで減少する自然数を対応させることで容易に示すことができる。これは、Code Gear の接続子 (loop connector) のようなものになる。

また、並列実行も Code Gear 単位のシャッフルと考えることにより、モデル検査的な証明が可能になる。この方についても考察する。

2 CbC に証明を付け加えた GearsAgda

CbC は goto 文中心に記述する言語で、**__code** という単位で実行される。この実行単位は有限な時間 (OS の tick など) で実行することが想定されている。つまり、不定な loop は goto 文の外で組み合わされる。

CbC は LLVM[8]/GCC[4] で実装されている。コンパイラの Basic block に相当すると考えてもよい。C form では例えば以下のように記述する。ここでは変数は record Env に格納されている。

```
__code whileLoop(Env *en, __code next(Env *en)) {
  if ( 0 >= en->varn )  goto next(en);
   else {
    en->varn = en->varn - 1;
    en->vari = en->vari + 1;
    goto whileLoop(en, next);
  }
}
```

Agda は pure fuctional な depedent type language で証明を記述できる。CbC の実行単位 codeGear は以下の形式 Agda form で記述する。常に不定の型 **t** を返すのが特徴になっている。

Agda では C の構造体に対応するのは record で、以下のように定義する。

```
record Env ( c : ℕ ) : Set where
 field
  varn : ℕ
  vari : ℕ
```

これにより、codeGear からは指定された継続 (continuation) を呼ぶしか型的に緩されない。これが、CbC の goto 文に相当する。変数の初期化を行う codeGear は以下のようになる。**record {}** が record の初期化になっている。

```
whileTest : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : Env → t) → t
whileTest c10 next = next (record {varn = c10 ; vari = 0} )
```

ここで、**a → b → c** は、**(a → b) → c** であり、Curry 化に対応している。例えば、仕様は

```
whileTestSpec1 : (c10 : ℕ) →  (e1 : Env c10 ) → vari e1 ≡ c10 → ⊤
whileTestSpec1 _ _ x = tt
```

という形に書ける。ここで、⊤ は、tt というただひとつの値を持つ型

```
data ⊤ : Set where
  tt : ⊤
```

である。これにより、

```
initTest : {l : Level} {t : Set l} → (c10 : ℕ)
   → (Code : (en : Env c10 ) → vari en ≡ c10 → t) → t
initTest c10 next = next (record {vari = c10 ; varn = 0 }) refl
```

という風に初期化が正しく値を設定していることを証明付きで書ける。ここで

```
data _≡_ {A : Set } (x y : A) : Set where
  refl {x : A} → x ≡ x
```

で、x = x つまり、自分自身は等しいという自然演繹の等式の推論になっている。この等しさは、λ 項の既約項同士の単一化なので、かなり複雑な操作になる。なので、一般的には等式の証明は自明にはならない。Agda では式変形をサポートしているので、少し見やすくすることが可能になってる。

実際に whilteTestSpec1 を検証するには

```
test0 : {l : Level} {t : Set l} → (c10 : ℕ) → ⊤
test0 c10 = initTest c10 ( λ en eq → whileTestSpec1 c10 en eq )
```

とする。initTest は値を提供している。Agda では証明は証明操作を表す λ 項なので値になっている。

仕様は複雑なプログラムの動作の結果として満たされるものなので、プログラムの内部に記述する必要がある。

initTest の **vari en ≡ c10** の証明は、その場で refl で行われている。whilteTestSpec1 はそれを受け取っているだけになっている。test1 での en は「任意の Env record」なので、**vari en ≡ c10** の証明は持っていない。

3 古典的な手法

プログラムを検証する方法としては、Hoare Logic [5, 1] が知られている。これは、プログラムを command で記述し、前提に成立する条件 Pre と 実行後に成立する条件 Post との

```
{Pre} command {Post}
```

の三つ組で条件を command 毎に記述していく方法である。command を例えばアセンブラ的な命令にすれば、プログラムの正しさを証明できる。loop の停止性の証明は、command 対してそれぞれ別に行う必要がある。

この方法では command 毎に Hoare logic の Soundness を定義する必要がある。実際に Hoare logic を Agda で実装した例がある [1]。

4 GearsAgda でのプログラム検証手法

プログラムの仕様はそのまま Agda の変数として持ち歩いているが、loop に関する証明を行うにはいくつか問題がある。

普通に while 文を書くと、Agda が警告を出す。

```
{-# TERMINATING #-}
whileLoop : {l : Level} {t : Set l} → Env → (Code : Env → t) → t
whileLoop env next with lt 0 (varn env)
whileLoop env next | false = next env
whileLoop env next | true =
   whileLoop (record {varn = (varn env) - 1 ; vari = (vari env) + 1}) next

test1 : Env
test1 = whileTest 10 ( λ env → whileLoop env ( λ env1 → env1 ))
```

{-# TERMINATING #-} は Agda が 停止性が確認できないことを示している。

5 simple while loop と証明付きデータ構造

loop の証明には、性質が保存する invariant (不変条件) と、停止性を示す reduction parameter (loop のたびに減少する自然数) の二つを使う。この時に、invariant を別な条件として使うと、プログラム全体が複雑になる。そこで、データ構造そのものに invariant を付加するとよい。上の例題では、

```
record Env ( c : ℕ ) : Set where
 field
   varn : ℕ
   vari : ℕ
   n+i=c : varn + vari ≡ c
```

とすると良い。この invariant を発見するのは一般的には難しいが、loop の構造とデータ構造、つまり、アルゴリズムとデータ構造から決まる。

whileLoop に invariant の証明を足し、軽量継続で loop を分割すると、停止性を codeGear の段階で Agda が判断する必要がなくなる。

```
whileLoopSeg : {l : Level} {t : Set l} → (c10 :  ℕ ) → (env : Env c10 )
   → (next : (e1 : Env c10 ) → varn e1 < varn env → t)
   → (exit : (e1 : Env c10 ) → vari e1 ≡ c10 → t) → t
whileLoopSeg c10 env next exit with  ( suc zero  ≤? (varn  env) )
whileLoopSeg c10 env next exit | no p = exit env ?
whileLoopSeg c10 env next exit | yes p = next env1  ? where
```

ここでは肝心の証明は **?** で省略している。Agda は、この様に証明を延期することができる。実際のコードでは証明が提供されている。

停止性を示す reducde と、loop 中に invariant が保存することから、loop を Agda で直接に証明することができる。

```
TerminatingLoopS : {l : Level} {t : Set l} (Index : Set ) → ( reduce : Index → ℕ)
   → (loop : (r : Index)  → (next : (r1 : Index)  → reduce r1 < reduce r  → t ) → t)
   → (r : Index) → t
TerminatingLoopS {_} {t} Index reduce loop  r with <-cmp 0 (reduce r)
... | tri≈ ¬a b ¬c = loop r ( λ r1 lt → ⊥-elim (lemma3 b lt) )
... | tri< a ¬b ¬c = loop r ( λ r1 lt1
      → TerminatingLoop1 (reduce r) r r1 (≤-step lt1) lt1 ) where
    TerminatingLoop1 : (j : ℕ) → (r r1 : Index) → reduce r1 < suc j  → reduce r1 < reduce r → t
    TerminatingLoop1 zero r r1 n≤j lt = loop r1 ( λ r2 lt1 → ⊥-
elim (lemma5 n≤j lt1))
    TerminatingLoop1 (suc j) r r1  n≤j lt with <-cmp (reduce r1) (suc j)
    ... | tri< a ¬b ¬c = TerminatingLoop1 j r r1 a lt
    ... | tri≈ ¬a b ¬c = loop r1 ( λ r2 lt1 → TerminatingLoop1 j r1 r2
       (subst ( λ k → reduce r2 < k ) b lt1 ) lt1 )
    ... | tri> ¬a ¬b c =  ⊥-elim ( nat-≤> c n≤j )
```

ここで、**tri≈** などは、二つの自然数を比較した <,=,> の三つの場合を表す data である。

```
<-cmp 0 (reduce r)
```

で、その三つの場合を作っている。そして、

```
... | tri> ¬a ¬b c =
```

という形で受ける。

```
⊥-elim ( nat-≤> c n≤j )
```

は、矛盾の削除則である。

TerminatingLoopS では、loop からの脱出は記述されていない。index が 0 以下になることはありえないので、loop はその前に終了しているはずである。それは、whileLoopSeg での reduce の証明がそれを保証している。つまり脱出条件は、TerminatingLoopS ではなく、whileLoopSeg で記述されている。

つまり、TerminatingLoopS は loop の接続を表す connector と考えることができる。

実際の証明は

```
proofGearsTermS : (c10 : ℕ ) → ⊤
proofGearsTermS c10 =
   TerminatingLoopS (Env c10) ( λ env → varn env) ( λ n2 loop
     → whileLoopSeg c10 n2 loop ( λ ne pe → whileTestSpec1 c10 ne pe ) )
       record { varn = 0 ; vari = c10 ; n+i=c = refl }
```

というようになる。最初の初期化の証明と同じように、プログラムは値の部分にあり実際に実行されていて、それが仕様を満たしている証明を **whileTestSpec1 c10 ne pe** が受け取っている。loop 変数が whileLoopSeg の軽量継続を停止性込みで接続している。

6 Hoare Logic との比較

Hoare Logic では、command と条件の書き方を規定して、その規定の中で Logic の Soundness を示している。条件の接続の段階で証明が必要であり、さらに Soundness での汎用的な証明が必要になる。command による記述はアセンブラ的になる。

GearsAgda では、command ではなく、GearAgda の形式を満たして入れば良い。codeGear 中で dataGear の持つ条件を証明することになる。Soundness は connector に閉じていて、比較的容易に証明される。さらに、? で証明を省略してもコード自体の実行は可能である。

この二つは、基本的に平行していて、Hoare Logic を理解していれば、GearsAgda を理解することは問題ない。また、Agda を知っていれば、これが Hoare Logic だというように説明することも可能である。

これが可能になっているのは、GearsAgda の軽量継続による分解であり、任意の関数呼び出しては、それに合わせて、TerminatingLoopS に相当するものを記述する必要がある。

ただし、GearsAgda 自体が軽量継続による制限から、アセンブラ的 (コンパイラの基本単位) になっているので、一般的な人向けのプログラミング言語のような可読性はない。

7 binary tree

二分木では、要素は自然の key によって順序付られている。普通のプログラミングでは、その順序付けは明示されていない。GearsAgda で、それを invariant として記述することができる。

GearsAgda は、軽量継続による制約により再帰呼び出しはしないことになっている。これにより、CbC との直接の対応が可能になっている。なので、二分木への挿入 insert は、

find による挿入点の探索と、**stack** の構成
stack をほどきながら木を置換していく操作

の二段階の構成になる。Haskell などでは、工夫された高階関数が使われるので stack は明示されない。

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

data treeInvariant {n : Level} {A : Set n} : (tree : bt A) → Set n where
    t-leaf : treeInvariant leaf
    t-single : (key : ℕ) → (value : A) →  treeInvariant (node key value leaf leaf)
    t-right : {key key  : ℕ} → {value value  : A} → {t  t  : bt A} → (key < key )
      → treeInvariant (node key  value  t  t )
      → treeInvariant (node key value leaf (node key  value  t  t ))
    t-left  : {key key  : ℕ} → {value value  : A} → {t  t  : bt A} → (key < key )
      → treeInvariant (node key value t  t )
      → treeInvariant (node key  value  (node key value t  t ) leaf )
    t-node  : {key key  key  : ℕ} → {value value  value  : A} → {t  t  t  t  : bt A}
      → (key < key ) → (key  < key )
      → treeInvariant (node key value t  t )
      → treeInvariant (node key  value  t  t )
      → treeInvariant (node key  value  (node key value t  t ) (node key  value  t  t ))
```

これが二分木のデータ構造と invariant の実装になる。これは、invariant というよりは、順序付された二分木の可能な値全部の集合であり、二分木の表示的意味論そのものになっている。

ここで、**(key < key)** は、Agda の型であり、そこには順序を示す data 構造が配る。つまり、二分木の順序は木の構成時に証明する必要がある。

さらに、stack が木をたどった順に構成されていることと、木が置き換わっていることを示す invariant が必要である。

```
data stackInvariant {n : Level} {A : Set n}  (key : ℕ) : (top orig : bt A)
      → (stack  : List (bt A)) → Set n where
    s-nil :  {tree0 : bt A} → stackInvariant key tree0 tree0 (tree0 :: [])
    s-right :  {tree tree0 tree  : bt A} → {key  : ℕ } → {v1 : A } → {st : List (bt A)}
       → key  < key  →  stackInvariant key (node key  v1 tree  tree) tree0 st
       → stackInvariant key tree tree0 (tree :: st)
    s-left :  {tree  tree0 tree : bt A} → {key  : ℕ } → {v1 : A } → {st : List (bt A)}
       → key  < key  →  stackInvariant key (node key  v1 tree  tree) tree0 st
       → stackInvariant key tree  tree0 (tree  :: st)

data replacedTree  {n : Level} {A : Set n} (key : ℕ) (value : A)
     : (before after : bt A ) → Set n where
    r-leaf : replacedTree key value leaf (node key value leaf leaf)
    r-node : {value  : A} → {t t  : bt A}
      → replacedTree key value (node key value  t t ) (node key value t t )
    r-right : {k : ℕ } {v1 : A} → {t t1 t2 : bt A}
        → k < key →  replacedTree key value t2 t
        →  replacedTree key value (node k v1 t1 t2) (node k v1 t1 t)
    r-left : {k : ℕ } {v1 : A} → {t t1 t2 : bt A}
```

```
      → key < k →  replacedTree key value t1 t
      →  replacedTree key value (node k v1 t1 t2) (node k v1 t t2)
```

木の構造は同じ順序を持っていても、同じ形にはならない。この replacedTree は、そういう場合が考慮されていない。

```
findP : {n m : Level} {A : Set n} {t : Set m} → (key : ℕ) → (tree tree0 : bt A )
   → (stack : List (bt A))
   →  treeInvariant tree ∧ stackInvariant key tree tree0 stack
   → (next : (tree1 : bt A) → (stack : List (bt A))
     → treeInvariant tree1 ∧ stackInvariant key tree1 tree0 stack
     → bt-depth tree1 < bt-depth tree   → t )
   → (exit : (tree1 : bt A) → (stack : List (bt A))
     → treeInvariant tree1 ∧ stackInvariant key tree1 tree0 stack
     → (tree1 ≡ leaf ) ∨ ( node-key tree1 ≡ just key )  → t ) → t
findP key leaf tree0 st Pre _ exit = exit leaf st Pre (case1 refl)
findP key (node key  v1 tree tree ) tree0 st Pre next exit with <-cmp key key
findP key n tree0 st Pre _ exit | tri≈ ¬a refl ¬c = exit n st Pre (case2 refl)
... | tri< a ¬b ¬c = next tree (tree ∷ st)
       ? , ?   ?
... | tri> ¬a ¬b c = next tree  (tree  ∷ st)
       ? , ?   ?
```

ここでも、実際の証明は? と省略している。ここで、木の深さを loop の停止条件として使っている。

```
replaceNodeP : {n m : Level} {A : Set n} {t : Set m} → (key : ℕ)
   → (value : A) → (tree : bt A)
   → (tree ≡ leaf ) ∨ ( node-key tree ≡ just key )
   → (treeInvariant tree ) → ((tree1 : bt A) → treeInvariant tree1
   →  replacedTree key value (child-replaced key tree) tree1 → t) → t
```

repace のプログラムはさらに煩雑なので型だけを示す。

最終的に、これらを loop connector で接続して証明付きのプログラムが完成する。

```
insertTreeP : {n m : Level} {A : Set n} {t : Set m} → (tree : bt A)
    → (key : ℕ) → (value : A) → treeInvariant tree
    → (exit : (tree repl : bt A)
       → treeInvariant repl ∧ replacedTree key value tree repl → t ) → t
insertTreeP {n} {m} {A} {t} tree key value P0 exit =
  TerminatingLoopS (bt A ∧ List (bt A) )
   { λ  p → treeInvariant (proj1 p)
     ∧ stackInvariant key (proj1 p) tree (proj2 p) }
  ( λ  p → bt-depth (proj1 p))   tree , tree ∷ []      P0 , s-nil
    $ λ  p P loop → findP key (proj1 p)  tree (proj2 p) P
      ( λ  t s P1 lt → loop   t ,  s    P1 lt )
    $ λ  t s P C → replaceNodeP key value t C (proj1 P)
    $ λ  t1 P1 R → TerminatingLoopS (List (bt A) ∧ bt A ∧ bt A )
       { λ  p → replacePR key value  (proj1 (proj2 p))
         (proj2 (proj2 p)) (proj1 p)  ( λ  _ _ _ → Lift n ⊤ ) }
         ( λ  p → length (proj1 p))   s ,   t , t1
           record { tree0 = tree ; ti = P0
             ; si = proj2 P ; ri = R ; ci = lift tt }
    $  λ  p P1 loop → replaceP key value  (proj2 (proj2 p)) (proj1 p) P1
       ( λ  key value {tree1} repl1 stack P2 lt
         → loop   stack ,   tree1 , repl1      P2 lt )
    $ λ  tree repl P →  exit tree repl
        RTtoTI0 _ _ _ _ (proj1 P) (proj2 P) , proj2 P
```

このプログラムは順序付きの二分木の invariant と、それが置換されている invariant を返すので、そこから、必要な仕様をすべて導出することができる。例えば、木がソートされていること、置換したもの以外は保存されていることなどである。

8　red black tree

赤黒木は、バランスした二分木の実装の一つである。木のノードに赤と黒の状態を持たせ、黒のノードの個数を左右でバランスさせる。これをそのまま invariant として記述する。この時、木の深さは b-depth で直接的に記述できる。

```
data RBTree {n : Level} (A : Set n) : (key : ℕ) → Color
     → (b-depth : ℕ) → Set n where
   rb-leaf     : (key : ℕ) → RBTree A key Black 0
   rb-single   : (key : ℕ) → (value : A) → (c : Color) →  RBTree A key c 1
   t-right-red : (key : ℕ) {key  : ℕ} → (value : A) → key < key  → {d : ℕ }
     → RBTree A key  Black d → RBTree A key Red d
   t-right-black : (key : ℕ) {key  : ℕ} → (value : A) → key < key
     → {c : Color} → {d : ℕ }→ RBTree A key  c d → RBTree A key Black (suc d)
   t-left-red  : (key  : ℕ) { key : ℕ} → (value : A) → key < key  → {d : ℕ}
     → RBTree A key Black d
     → RBTree A key  Red d
   t-left-black  : (key  : ℕ) {key : ℕ} → (value : A) → key < key   → {c : Color} → {d : ℕ}
     → RBTree A key  c d
     → RBTree A key  Black (suc d)
   t-node-red  : (key  : ℕ) { key key  : ℕ} → (value : A)
     → key < key  → key  < key    →  {d : ℕ}
     → RBTree A key Black d
     → RBTree A key  Black d
     → RBTree A key  Red d
   t-node-black  : (key  : ℕ) {key key  : ℕ} → (value : A)
     → key < key  → key  < key   →  {c c1 : Color} {d : ℕ}
     → RBTree A key c d
     → RBTree A key  c1 d
     → RBTree A key  Black (suc d)
```

かなり複雑な操作だが、ここでは非破壊的な赤黒木を使うので stack が必須となる。double linked な木にして stack を使わない手法もあるが破壊的な操作になるし、純関数型のデータ構造は循環構造を扱えないので、double link にするのはかなり難しい。循環構造は GC とかでも問題があり、実際のプログラミングでもバグが起きやすい部分となる。これを取り扱うにはポインタを数字で置き換えることが必須となる。

赤黒木では最大三つのノードを見てバランスを判断する。必要があれば、stack を使って一つ上の木に戻ってバランスに必要な木の回転を行う。なので、replaceTree には回転を扱う部分を追加する必要がある。さらに、stack には、二つのノードを明示する invariant が必要になる。

```
data rbstackInvariant2 {n : Level} {A : Set n}  {key : ℕ} {c : Color} {d : ℕ}
      (orig : RBTree A key c d ) :
      {k1 k2 d1 d2 : ℕ} {c1 c2 : Color} (parent : RBTree A k1 c1 d1)
      (grand : RBTree A k2 c2 d2) Set n where
   s-head :  rbstackInvariant2 orig ? orig
   s-right : rbstackInvariant2 orig ? ? → rbstackInvariant2 orig ? ?
   s-left : rbstackInvariant2 orig ? ? → rbstackInvariant2 orig ? ?
```

replaceTree は、RBTree をそのまま使うとかなり煩雑になる。

まず、

```
findRBP : {n m : Level} {A : Set n} {t : Set m} → (key : ℕ) {key1 d d1 : ℕ}
    → {c c1 : Color}
    → (tree : RBTree A key c d ) (tree1 : RBTree A key1 c1 d1 )
    →  rbstackInvariant tree key1
    → (next : {key0 d0 : ℕ} {c0 : Color} → (tree0 : RBTree A key0 c0 d0 )
      → rbstackInvariant tree key1 → rbt-depth A tree0 < rbt-
depth A tree1 → t )
    → (exit : {key0 d0 : ℕ} {c0 : Color} → (tree0 : RBTree A key0 c0 d0 )
      → rbstackInvariant tree key1
      → (rbt-depth A tree ≡ 0 ) ∨ ( rbt-key A tree ≡ just key )  → t ) → t
```

findRBP で置き換える部分までの stack を構成する。この時に、脱出条件として、ノードのキーが等しいか、leaf であることを要求する。

```
replaceRBP : {n m : Level} {A : Set n} {t : Set m}
    → (key : ℕ) → (value : A) → {key0 key1 key2 d0 d1 d2 : ℕ}
      {c0 c1 c2 : Color}
    → (orig : RBTree A key1 c1 d1 ) → (tree : RBTree A key1 c1 d1 )
      ( repl : RBTree A key2 c2 d2 )
    → (si : rbstackInvariant orig key1)
    → (ri : replacedTree key value (RB→bt A tree) (RB→bt A repl))
    → (next : ℕ → A → {k1 k2 d1 d2 : ℕ} {c1 c2 : Color}
      → (tree1 : RBTree A k1 c1 d1 ) (repl1 : RBTree A k2 c2 d2 )
      → (si1 : rbstackInvariant orig k1)
      → (ri : replacedTree key value (RB→bt A tree1) (RB→bt A repl1))
      → rbsi-len orig si1 < rbsi-len orig si → t )
    → (exit : {k1 k2 d1 d2 : ℕ} {c1 c2 : Color} (tree1 : RBTree A k1 c1 d1 )
      → (repl1 : RBTree A k2 c2 d2 )
      → (ri : replacedTree key value (RB→bt A orig) (RB→bt A repl1))
      → t ) → t
```

replaceRBP では、木をバランスさせながら、replacedTree を作成していく。

9 Invariant を基本としたコード生成

binary tree でも red black tree でも、invariant は表示的意味論つまり、木のすべての可能な場合になる。これを入力とすることによりかなりの部分が自動的に導出される。この時に、出力する invariant はすべて構成する必要がある。これは、残念ながらやさしいとはいえない場合がある。しかし、それほど難しい処理にはならない。ただ、場合の数が多い。

残念ながら、invariant が間違っている場合もありえる。その場合のコードの修正は自明ではない。しかし、影響を小さくした変更は可能ではある。

10 より効率を重視したコードの扱い

バランス木はさまざまな実装がある。例えば B-Tree あるいは Skip LIST などがありえる。この場合は赤黒木に帰着、あるいは変換できれば良い。しかし、それが簡単な操作とは限らない。

変換の正しさは木の操作と同時に記述する必要がある。

11 Concurrency

赤黒木は、OS やデータベースあるいはファイルシステムでの様々な場所で使われることになり、その操作は並列に行われる。GearsAgda では、その証明も取り扱う。赤黒木のトランザクションは、木のルートの置き換えになる。

まず、GearsAgda での並行実行を定義する。GearsAgda での並行実行の単位は、codeGear であり、それは、メタレベルでは、単なる番号に対応したコードとして扱われる。このコードは、プロセス構造体とメモリ空間に相当する Context から実行の詳細を取り出して、より細かいレベルでの計算を行う。

Context には、dataGear 全部の他に、詳細とメタレベルの変換を行う codeGear の stub、次に実行する codeGear の番号が入っている。また、実行が失敗した時の例外処理を担当する codeGear の番号も入っている。

```
record Context  : Set  where
  field
    next :     Code
    fail :     Code

    c_Phil-API :    Phil-API
    c_AtomicNat-API : AtomicNat-API

    mbase :    ℕ
    memory :   bt Data
    error :    Data
    fail_next :     Code
```

codeGear の番号は **code_table** で管理される。

```
code_table :  {n : Level} {t : Set n} → Code
  → Context → ( Code → Context → t) → t
```

OS(あるいは分散計算を含む並行実行全体) は、単純に List Context で表される。

```
step : {n : Level} {t : Set n} → List Context → (List Context → t) → t
```

一つの codeGear の実行は List Context の変更となる。

これで可能な scheduling を網羅すれば並行実行を定義できることになる。

この時に、全体が満たすべき性質は、scheduler の invariant として記述することになる。それは、当然、時相論理的な仕様記述になる。

仕様は一般的には、フェアはスケジュールに対して、並行実行の時相論理的な記述となる。例えば、赤黒木の読み込みと書き込みの競合状態がない、あるいは、Live lock しないなどである。

これらの性質の記述は、まだ、十分に考察されてないが、ωオートマトンなどで意味を規定することになると思われる。

12　実際に実行するには

GearsAgda の記述は、並行実行を含めて CbC に変換可能であり、そのまま実行できる。フェアな scheduler であれば並行実行に関する証明も可能になる予定である。

Agda からの変換は、Haskell / JavaScript に大しては既に存在する。CbC に変換する場合は、メモリ管理の問題をなんとかする必要がある。メモリオーバフローに関しては、Context に状況が格納されており、オブジェクトの生成をメタ計算レベルで追跡することができる。

13　証明のスケーラビリティ

このように GearsAgda を用いた並行実行を含む検証は可能だが、それは実用的なスケーラビリティを持つのかが問題になる。証明が大きくなる、あるいは検証に時間がかかるのでは困る。

実際、Agda の証明が複雑になると、メモリを数十 G 食ったり、数分証明のチェックに時間がかかることがある。なので、確実にスケールするとは言い難い。

証明自体は、個々の証明は小さく簡単なことが多い。ただし、並行実行を含む invariant がどのような大きさになるかはまだ不明である。Agda による集合論の実装が既にあるので、集合を使った記述により invariant の大きさを小さくすることができるかも知れない。

また、Agda そのものに並行分散の機能をいれることが必要になると思われる。この場合は、Agda を GearsAgda で記述することによりより細かいレベルでの並行実行あるいは、メモリ管理、証明の優先順位などの処理が可能になると期待される。

個々の証明が AI によって容易になることは予想される。確率的な AI による生成であっても、invariant の正しさが担保できるのであれば証明のチェックはたやすい。また、GearsAgda は、AI によるコード生成と相性がよいと思われる。

Invariant 自体は、ライブラリとして実装されるべきだと思われるが、Agda のライブラリでもさまざまなものが既に実装されている。

14　比較

OS の検証自体の研究はいろいろあり、メモリ管理に関する専用の論理や、モデル検査などが研究されている。

Lamport の Temporal logic of action も有効である。

実際の証明は、Haskell に似た言語に変換される場合が多い。

GearsAgda は、CbC に直接対応した変換を持ち、並行実行の単位を持つところが他の手法と異なる。

また、code table を構成する時に、証明やモデル検査を OS が行えるならば、OS とそのアプリの信頼性を向上させることができると考えられる。

15　まとめ

GearsAgda を用いた、並行実行を含む赤黒木の検証方法について考察した。まだ、実装できてない部分、あるいは、並行実行の時相論理による仕様記述などの問題が残っている。

References

[1] Hoare logic in agda2. Accessed: 2018/12/17(Mon).

[2] The Coq Proof Assistance. https://coq.inria.fr.

[3] Andrzej Filinski. Monads in action. In POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 483–494, New York, NY, USA, 2010. ACM.

[4] Free Software Foundation, Inc. GCC, the GNU Compiler Collection, March 2008.

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

[6] Yoshiki Kinoshita. Agda language. Technical Report PS-2008-014, 独立行政法人産業技術総合研究所, 2008.

[7] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. ACM Trans. Comput. Syst., 32(1):2:1–2:70, February 2014.

[8] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[9] E. Moggi. Computational lambda-calculus and monads. In Proceedings of the Fourth Annual Symposium on Logic in Computer Science, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[10] Shinji KONO. CbC, March 2020.

[11] 伊波立樹, 東恩納琢偉, and 河野 真治. Code gear、data gear に基づく os のプロトタイプ. In **情報処理学会システムソフトウェアとオペレーティング・システム研究会** (OS), May 2016.

[12] 外間 政尊 and 河野 真治. Gears os の hoare logic をベースにした検証手法. In **電子情報通信学会ソフトウェアサイエンス研究会**, Jan 2019.