Gears Agda による
Left Learning Red Black Tree の検証

Uechi Yuto, Shinji Kono 琉球大学

証明を用いてプログラムの信頼性の向上を目指す

- プログラムの信頼性を高めることは重要な課題である
  - 信頼性を高める手法として「モデル検査」や「定理証明」など
- 当研究室でCbCという言語を開発している
  - C言語からループ構造とサブルーチンを取り除き、継続を導入したC言語の下位言語となる
- 先行研究では Hoare Logic を用いて 簡単なプログラムの検証を行った
- 本研究では Gears Agda にて重要なアルゴリズムの一つである Red Black Tree を検証する

CbC について

- CbCとは当研究室で開発しているC言語の下位言語
  - 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
  - 継続呼び出しは引数付き goto 文で表現される。
  - 処理の単位を Code Gear, データの単位を Data Gear として記述するプログラミング言語
- CbC のプログラミングでは Data Gear を Code Gear で変更し、その変更を次の Code Gear に渡して処理を行う。

Agda の基本

- Agdaとは定理証明支援器であり、関数型言語
- Agdaでの関数は、最初に型について定義した後に、関数を定義する事で記述する
- 型の定義部分で、入力と出力の型を定義できる
  - input → output のようになる
- 関数の定義は = を用いて行う
  - 関数名の後、 = の前で受け取る引数を記述します

Agda の基本 record

オブジェクトあるいは構造体

```
record Env  : Set where
  field
    varn : N
    vari : N
open Env
```

型に対応する値の導入(intro)

```
record {varx = zero ; vary = suc zero}
```

record の値のアクセス(elim)

```
(env : Env) → varx env
```

Gears Agda の記法

Gears Agda では CbC と対応させるためにすべてLoopで記述する
loopは `→ t` の形式で表現する
再帰呼び出しは使わない(構文的には禁止していないので注意が必要)

```
{-# 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
```

- t を返すことで継続を表す(tは呼び出し時に任意に指定してもよい. testに使える)
- tail call により light weight continuation を定義している

Gears Agda と Gears CbC の対応

Gears Agda

- 証明向きの記述
- Hoare Condition を含む

Gears CbC

- 実行向きの記述
- メモリ管理, 並列実行を含む

Context

- processに相当する
- Code Gear 単位で並列実行

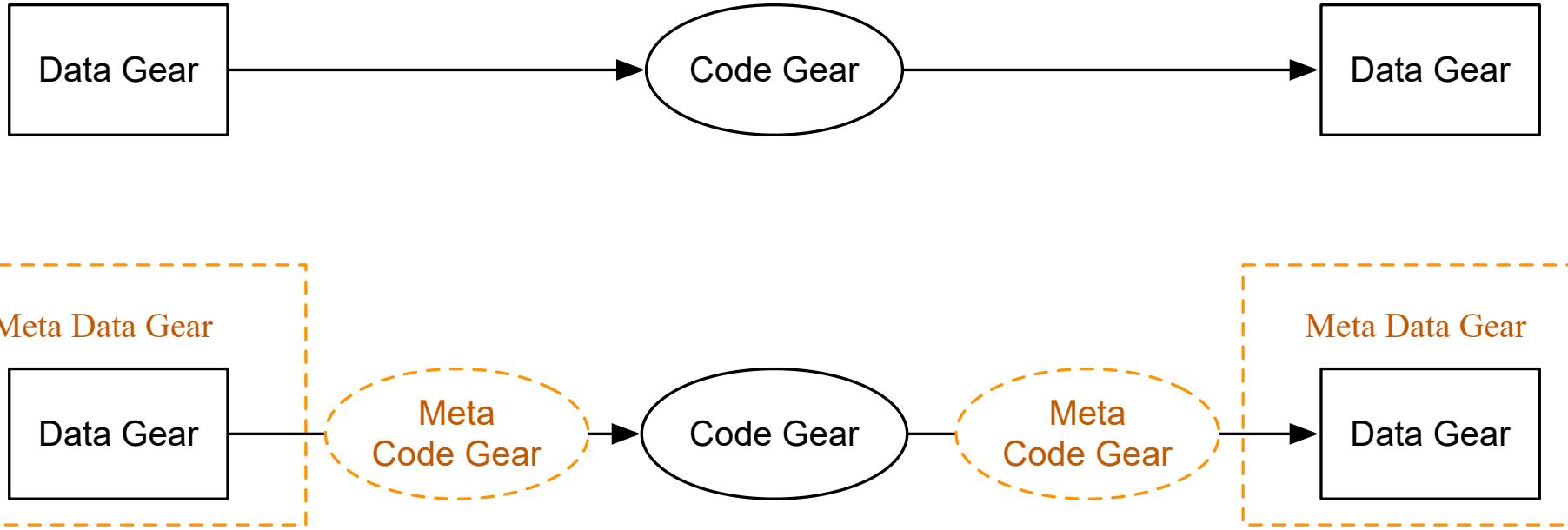
Normal level と Meta Level を用いた信頼性の向上

Normal Level

- 軽量継続
- Code Gear 単位で関数型プログラミングとなる
- atomic(Code Gear 自体の実行は割り込まれない)
- ポインタの操作は含まれない

Meta Level

- メモリやCPUなどの資源管理, ポインタ操作
- Hoare Condition と証明
- Contextによる並列実行
- monadに相当するData構造


Data Gear
Code Gear
Data Gear
Meta Data Gear
Data Gear
Meta
Code Gear
Code Gear
Meta
Code Gear
Meta Data Gear
Data Gear

Gears Agda と Hoare Logic

C.A.R Hoare, R.W Floyd が考案

- Pre Condition → Command → Post Condition

Gears Agda による Command の例

```
{-# 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
```

- `{-# TERMINATING #-}`
- with 文
- 型と値

Gears Agda の Pre Condition Post Condition

```
whileLoopSeg : {l : Level} {t : Set l} → {c10 :  N } → (env : Env) → (varn env + vari env ≡ c10)
   → (next : (e1 : Env )→ varn e1 + vari e1 ≡ c10 → varn e1 < varn env → t)
   → (exit : (e1 : Env )→ vari e1 ≡ c10 → t) → t
```

- Terminatingを避けるためにloopを分割
- `varn env + vari env ≡ c10` が Pre Condition
- `varn e1 + vari e1 ≡ c10` が Post Condition
- `varn e1 < varn env → t` が停止を保証する減少列

これは命題なので証明を与えてPre ConditionからPost Conditionを導出する必要がある
証明は値として次のCodeGearに渡される

Loop の接続

分割したloopを接続するMeta Code Gearを作成する

```
TerminatingLoopS : {l : Level} {t : Set l} ( Index : Set )
  → {Invraiant : Index → Set } → ( reduce : Index → N)
  → (loop : (r : Index) → Invraiant r
    → (next : (r1 : Index) → Invraiant r1 → reduce r1 < reduce r → t ) → t)
  → (r : Index) → (p : Invraiant r) → t
```

- IndexはLoop変数
- Invariantはloop変数の不変条件
- loopに接続するCode Gearを与える
- reduceは停止性を保証する減少列

これを一般的に証明することができている

実際のloopの接続

証明したい性質を以下のように記述する

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

loopをTerminatingLoopSで接続して仕様記述に繋げる

```
proofGearsTermS : {c10 :  N} → ⊤
proofGearsTermS {c10} = whileTest' {_} {_}  {c10} (λ n p →  conversion1 n p (λ n1 p1 →
    TerminatingLoopS Env (λ env → varn env)(λ n2 p2 loop → whileLoopSeg {_} {_} {c10} n2 p2 loop
    (λ ne pe → whileTestSpec1 c10 ne pe)) n1 p1 ))
```

- conversion1はPre Condition をPost Conditionに変換する
- whileLoopSeg
- ⊤

test との違い

- test では実数を与えた際の出力が仕様に沿ったものであるかを検証する
  - コーナーケースで仕様に沿った動きをしていない場合を考慮できない
- 今回の定理証明を用いた証明では実数を必要としない
  - そのため、入力の型の範囲全てに対して仕様を満たしているか検証できる

Gears Agda による BinaryTree の実装

CbCと並行した実装をするため、Stackを明示した実装をする

```
find : {n m : Level} {A : Set n} {t : Set m} → (key : ℕ) → (tree : bt A ) → List (bt A)
  → (next : bt A → List (bt A) → t ) → (exit : bt A → List (bt A) → t ) → t
find key leaf st _ exit = exit leaf st
find key (node key₁ v1 tree tree₁) st next exit with <-cmp key key₁
find key n st _ exit | tri≈ ¬a b ¬c = exit n st
find key n@(node key₁ v1 tree tree₁) st next _ | tri< a ¬b ¬c = next tree (n :: st)
find key n@(node key₁ v1 tree tree₁) st next _ | tri> ¬a ¬b c = next tree₁ (n :: st)
```

置き換えるnodeまでTreeを辿り、Stackに逆順にTreeを積んでいく

Gears Agda による BinaryTree の実装 replace node

置き換えたnodeをStackを解消しながらTreeを再構成する

```
replace : {n m : Level} {A : Set n} {t : Set m}
  → (key : N) → (value : A) → bt A → List (bt A)
  → (next : N → A → bt A → List (bt A) → t )
  → (exit : bt A → t) → t
replace key value repl [] next exit = exit repl    -- can't happen
replace key value repl (leaf :: []) next exit = exit repl
replace key value repl (node key₁ value₁ left right :: []) next exit with <-cmp key key₁
... | tri< a ¬b ¬c = exit (node key₁ value₁ repl right )
... | tri≈ ¬a b ¬c = exit (node key₁ value  left right )
... | tri> ¬a ¬b c = exit (node key₁ value₁ left repl )
replace key value repl (leaf :: st) next exit = next key value repl st
replace key value repl (node key₁ value₁ left right :: st) next exit with <-cmp key key₁
... | tri< a ¬b ¬c = next key value (node key₁ value₁ repl right ) st
... | tri≈ ¬a b ¬c = next key value (node key₁ value  left right ) st
... | tri> ¬a ¬b c = next key value (node key₁ value₁ left repl ) st
```

Binary Tree の3種類の Invariant

Tree Invariant

- Binary Tree が Key の順序に沿って構成されていることを表すData構造

Stack Invariant

- Stackが木の昇順に構成されていることを表す

Replace Tree

- 2つの木の1つのnodeが入れ替わっていることを示す

Tree Invariant

- Invariant というよりも可能な Binary Tree 全体の集合を表す型
- 制約付きのBinary Tree

```
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 key₁ : N} → {value value₁ : A} → {t₁ t₂ : bt A} → (key < key₁)
      → treeInvariant (node key₁ value₁ t₁ t₂)
      → treeInvariant (node key value leaf (node key₁ value₁ t₁ t₂))
    t-left  : {key key₁ : N} → {value value₁ : A} → {t₁ t₂ : bt A} → (key < key₁)
      → treeInvariant (node key value t₁ t₂)
      → treeInvariant (node key₁ value₁ (node key value t₁ t₂) leaf )
    t-node  : {key key₁ key₂ : N} → {value value₁ value₂ : A}
      → {t₁ t₂ t₃ t₄ : bt A} → (key < key₁) → (key₁ < key₂)
      → treeInvariant (node key value t₁ t₂)
      → treeInvariant (node key₂ value₂ t₃ t₄)
      → treeInvariant (node key₁ value₁ (node key value t₁ t₂) (node key₂ value₂ t₃ t₄))
```

Stack Invariant

- StackにはTreeを辿った履歴が残っている
- 辿り方はKeyの値に依存する
- 実際のStackよりも豊富な情報を持っている

```
data stackInvariant {n : Level} {A : Set n} (key : N) : (top orig : bt A)
  → (stack : List (bt A)) → Set n where
    s-single : {tree0 : bt A} → stackInvariant key tree0 tree0 (tree0 :: [])
    s-right : {tree tree0 tree₁ : bt A} → {key₁ : N } → {v1 : A } → {st : List (bt A)}
      → key₁ < key  → stackInvariant key (node key₁ v1 tree₁ tree) tree0 st
      → stackInvariant key tree tree0 (tree :: st)
    s-left :  {tree₁ tree0 tree : bt A} → {key₁ : N } → {v1 : A } → {st : List (bt A)}
      → key  < key₁ → stackInvariant key (node key₁ v1 tree₁ tree) tree0 st
      → stackInvariant key tree₁ tree0 (tree₁ :: st)
```

Replace Tree

- 木の特定のnodeが正しく置き換えられているか

```
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 : {value₁ : A} → {t t₁ : bt A}
      → replacedTree key value (node key value₁ t t₁) (node key value t t₁)
    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)
```

find の Hoare Condition

findPでは treeInvariant をつかって stackInvariant を生成する

停止性を証明する木の深さの不等式も証明する

```
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
```

Hoare Condition の拡張

testなどでは仮定が入ることがある

Hoare Conditionを拡張可能な部分があるrecordで定義する

Cが拡張される部分で, これはDataの継続に相当する

Code Gear に継続があるように Data Gearにも継続がある

```
record findPR {n : Level} {A : Set n} (key : ℕ) (tree : bt A ) (stack : List (bt A))
  (C : ℕ → bt A → List (bt A) → Set n) : Set n where
   field
     tree0 : bt A
     ti0 : treeInvariant tree0
     ti : treeInvariant tree
     si : stackInvariant key tree tree0 stack
     ci : C key tree stack     -- data continuation
```

拡張部分の記述と推論

拡張部分はrecord findPC で定義する

拡張部分の推論はrecord findExt で定義する

```
record findPC {n : Level} {A : Set n} (value : A) (key1 : ℕ) (tree : bt A ) (stack : List (bt A)) : Set n where
   field
     tree1 : bt A
     ci : replacedTree key1 value tree1 tree

record findExt {n : Level} {A : Set n} (key : ℕ) (C : ℕ → bt A → List (bt A) → Set n) : Set (Level.suc n) where
   field
      c1 : {key₁ : ℕ} {tree tree₁ : bt A } {st : List (bt A)} {v1 : A}
        → findPR key (node key₁ v1 tree tree₁) st C → key < key₁  → C key tree (tree :: st)
      c2 : {key₁ : ℕ} {tree tree₁ : bt A } {st : List (bt A)} {v1 : A}
        → findPR key (node key₁ v1 tree tree₁) st C → key > key₁  → C key tree₁ (tree₁ :: st)
```

replade Node の Invariant

repraceTree は置き換えるべきnodeが実行時に決まるので関数を挟んだInvariantになる

```
child-replaced :  {n : Level} {A : Set n} (key : ℕ)   (tree : bt A) → bt A
child-replaced key leaf = leaf
child-replaced key (node key₁ value left right) with <-cmp key key₁
... | tri< a ¬b ¬c = left
... | tri≈ ¬a b ¬c = node key₁ value left right
... | tri> ¬a ¬b c = right

record replacePR {n : Level} {A : Set n} (key : ℕ) (value : A) (tree repl : bt A )
  (stack : List (bt A)) (C : bt A → bt A → List (bt A) → Set n) : Set n where
   field
     tree0 : bt A
     ti : treeInvariant tree0
     si : stackInvariant key tree tree0 stack
     ri : replacedTree key value (child-replaced key tree ) repl
     ci : C tree repl stack     -- data continuation
```

最終的な証明コード

insertしたkeyをfindすると元の値が取れてくることを証明する

insertTreeSpec0が仕様

conteinsTreeがtestコードかつ証明になっている

証明の詳細はコードのHoara condition の証明に入っている

```
insertTreeSpec0 : {n : Level} {A : Set n} → (tree : bt A) → (value : A)
  → top-value tree ≡ just value → ⊤
insertTreeSpec0 _ _ _ = tt

containsTree : {n : Level} {A : Set n} → (tree tree1 : bt A) → (key : ℕ) → (value : A)
  → treeInvariant tree1 → replacedTree key value tree1 tree  → ⊤
containsTree {n}  {A}  tree tree1 key value P RT =
   TerminatingLoopS (bt A ∧ List (bt A) )
     {λ p → findPR key (proj1 p) (proj2 p) (findPC value ) } (λ p → bt-depth (proj1 p))
              ⟪ tree , tree ∷ []  ⟫ ?
       $ λ p P loop → findPPC1 key value (proj1 p) (proj2 p) P (λ t s P1 lt → loop ⟪ t , s ⟫ P1 lt )
       $ λ t1 s1 P2 found? → insertTreeSpec0 t1 value (lemma6 t1 s1 found? P2)
```

replace Tree の性質

Replace Tree 自体は Tree に特定の値が入っていることを示すData構造になっている
これをHoare条件として持ち歩くことにより、Binary Treeの詳細に立ち入らず何が入っているかを指定できる

これにより木を用いるプログラムでの証明を記述できる
insert Tree や Find Node の停止性も証明されている

今後の研究方針

- Deleteの実装
- Red Black Tree の実装
  - 今回定義した条件はそのまま Red Black Tree の検証に使用できるはず
- Contextを用いた並列実行時のプログラムの正しさの証明
  - synchronized queue
  - concurrent tree
- xv.6への適用
- モデル検査

読めない間は待っている

tree

まとめ

- Gears Agda にて Binary Tree を検証することができた
  - Gears Agda における Termination を使用しない実装の仕方を確立した
  - Hoare Logic による検証もできるようになった
  - 今後は Red Black Tree の検証をすすめる
- モデル検査をしたい