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、関数型言語である
- 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できる
  - これが Curry-Howard 対応となる
- 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

```
sample1 : (A : Set ) → Set
sample1 a = a

sample2 : (A : Set ) → (B : Set ) → Set
sample2 a b = b
```

Agda の基本 record

- 2つのものが同時に存在すること
- Record 型とはオブジェクトあるいは構造体のようなもの

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

記述する際の基本的な例を以下に挙げる

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

Agda の基本 data

- 一つでも存在すること
- どちらかが成り立つ型を Data 型 で書く

```
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
```

記述する際の基本的な例を以下に挙げる

```
datasample : bt → N
datasample leaf = zero
datasample (node key value left right) = suc zero
```

Agda の基本 短縮記法

- with を使用することでその変数のパターンマッチすることもできる

```
patternmatch-default : Env → N
patternmatch-default record { varn = zero ; vari = i } = i
patternmatch-default record { varn = suc n ; vari = i } = patternmatch-default record { varn = n ; vari = suc i }
```

```
patternmatch-extraction : env → N
patternmatch-extraction env with varn env
patternmatch-extraction zero = vari env
patternmatch-extraction suc n = patternmatch-extraction record { varn = n ; vari = suc (vari env) }
```

- ... | `を使用することで関数の定義部分を省略できる

```
patternmatch-extraction' : env → N
patternmatch-extraction' env with varn env
... | zero = vari env
... | suc n = patternmatch-default' record { varn = n ; vari = suc (vari env) }
```

Gears Agda の記法

- Agda では関数の再帰呼び出しが可能であるが、CbC で再起処理を実装しても値は帰って来ない
  - Agda で実装を行う際には再帰呼び出しを行わないようにする。

```
plus-com : {l : Level} {t : Set l} → Env → (next : Env → t) → (exit : Env → t) → t
plus-com env next exit with vary env
... | zero = exit (record { varx = varx env ; vary = vary env })
... | suc y = next (record { varx = suc (varx env) ; vary = y })
```

- `→ t` で Set に遷移させるようにし、この後に関数が続くことと、関数を tail call していることで Continuation を定義している

Gears Agda と Gears CbC の対応 x

- 証明のしやすいGears Agda で実装を行いながらContext単位で同時に検証も行う
- Gears Agda は検証向きの実装となり、CbCでは高速な実行向きとなる
- Gears Agda での検証がCbCによる高速な実行の信頼性となる

Normal level と Meta Level を用いた信頼性の向上

CbCのプログラムの実行部分は以下の2つから構成される

- Normal Level は軽量継続で関数型プログラミングとなる
  - atomicってなんだろう
  - ポインタの操作はここでは行わず Meta Level にて行う
- Meta Level では信頼性を保証するために必要な計算を行う
  - メモリやCPUなどの資源管理
  - context
  - 証明
  - 並列実行（他のプロセスとの干渉）
  - monad

S

Hoare Logic について

- Hoare Logic 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
- 「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
    - CbC の実行を継続するという性質に非常に相性が良い

pre/post condition を Gears Agda では Meta Input Data Gear として
プログラム中に遷移させていくことで、プログラムが終始仕様に沿った動きをしている
ことを検証する

Gears Agda による検証の例（実装）

- while program の検証を例に挙げる
  - 以下は実装部分のコードとなる

```
{-# 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
```

Gears Agda による検証の例（検証付き実装）

- 先ほど実装した while program に対して証明を付ける
- loop を接続する Meta Gear も用意する

```
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
```

- 入力として仕様の証明を受け取る
- 出力として次の関数に仕様の証明を渡す
- Hoare Conditionがプログラムの開始から停止するまで接続されていれば証明は完成
  - Meta Gear にて証明を値としてあたえているため

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

- Agdaが変数への再代入を許していないためそのままでは木構造を実装できない
  - 木構造を辿る際に現在いるノード以下の木構造をそのまま stack に格納する
  - replace / delete などの操作を行った後に stack を参照し Tree を再構築する
  - CbCへの変換の時に問題になりそうな予感
- ここの説明いらないかも？

Gears Agda による BinaryTree の実装 find node

```
find : {n m : Level} {A : Set n} {t : Set m} → (env : Env A )
  → (next : (env : Env A ) → t ) → (exit : (env : Env 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)
```

Gears Agda による BinaryTree の実装 replace node

```
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
```

Gears Agda による BinaryTree の実装 loop connecter

Gears Agda による Binary Tree の検証 Invariant

具体的な例を一つ挙げて、Invariantの説明を行う

- Binary Tree の性質である、左の子のkeyは親より小さく、
  右の子のkeyは親より大きいことを検証
- Stack に格納されているTreeはその次のStackの減少列になっているか
- Tree を正しく入れ替えられているか

Gears Agda による Binary Tree の検証 Invariant

- Tree Invariant

```
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))
```

Gears Agda による Binary Tree の検証 Invariant

- Stack Invariant

```
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)
```

Gears Agda による Binary Tree の検証 Invariant

- Replace Invariant

```
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)
```

Gears Agda による Binary Tree の検証

Binary Tree の実装に対して上述した3つのInvariantを
Meta Data Gear として渡しながら実行できるように記述する

```
findP : {n m : Level} {A : Set n} {t : Set m} → (env : Env A)
  → treeInvariant env ∧ stackInvariant env
  → (exit : (env : Env A) → treeInvariant env ∧ stackInvariant env → t ) → t
```

今後の研究方針

- 現在は Binary Tree の検証までしか行えていないが、
  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
- 今後は Gears Agda による実装と条件の追加をおこなう
- モデル検査

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率した
  - 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

英語版も欲しい

condition を テンプレみたいに作ってかきやすくする話