Proving properties of the tree representation of dynamic bit sequences

Reynald Affeldt [1] Jacques Garrigue [2]
Xuanrui Qi [3] Kazunari Tanaka [2]

[1] 産業技術総合研究所

[2] 名古屋大学多元数理科学研究科

[3] Tufts University

November 22, 2018

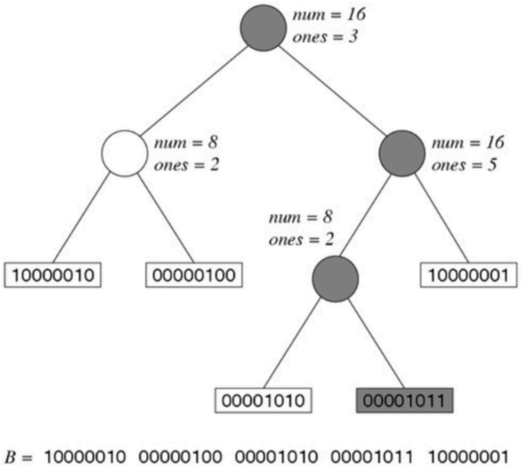
動的簡潔データ構造

- 簡潔データ構造の最適な表現は配列を使うので, 変更に不向きである
- しかし, データを動的に変えたい場合が多々ある
- 挿入・削除のコストを抑えるために, アクセスや rank・select の定数時間を諦めなければならない
- 表現に平衡木を使うと, 全ての操作が $O(\log n)$ で行える

[Navarro 2016, Chapter 12]

Proving properties of the tree representation of dynamic bit sequences

動的ビット列の表現



num は左部分木のビット数, *ones* は左部分木の 1 の数

実装方針

- 平衡木として red-black tree を使用
  - 複雑さの結果は平衡木の種類に寄らない
  - 純粋な関数型プログラミング言語で表現しやすい
  - 既に Coq で複数の形式化がある
  - ただし，データの配置が異なるので再実装した
- 型の使い方の異なる 2 つの実装を試みた
  1. 「通常」の型を使った一階述語による実装
  2. 依存型を使い，正しいデータしか書けない実装
- rank・select・insert・delete の実装の証明

通常型による実装

red-black tree をビット列に応用

```
Inductive color := Red | Black.
Inductive btree (D A : Type) : Type :=
| Bnode of color & btree D A & D & btree D A
| Bleaf of A.

Definition dtree := btree (nat * nat) (seq bool).
```

dflattenで木の意味を定義する

```
Fixpoint dflatten (B : dtree) :=
  match B with
  | Bnode _ l _ r => dflatten l ++ dflatten r
  | Bleaf s => s
  end.
```

内部データが守るべき不変量

```
Fixpoint wf_dtree (B : dtree) :=
  match B with
  | Bnode _ l (num, ones) r =>
    [&& num == size (dflatten l), ones == count_mem true (dflatten l),
        wf_dtree l & wf_dtree r]
  | Bleaf arr => (w ^ 2)./2 <= size arr < (w ^ 2).*2
  end.
```

基本操作

```
Fixpoint drank (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    if i < num then drank l i
               else ones + drank r (i - num)
  | Bleaf s =>
    rank true i s
  end.

Lemma dtree_ind (P : dtree -> Prop) :
  (forall c l r num ones,
   num = size (dflatten l) ->
   ones = count_mem true (dflatten l) ->
   wf_dtree l /\ wf_dtree r ->
   P l -> P r -> P (Bnode c l (num, ones) r)) ->
  (forall s, (w ^ 2)./2 <= size s < (w ^ 2).*2 -> P (Bleaf _ s)) ->
  forall B, wf_dtree B -> P B.

Lemma drankE (B : dtree) i :
  wf_dtree B -> drank B i = rank true i (dflatten B).
```

どちらも証明が数行で収まる

基本操作

```
Fixpoint dselect_1 (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    if i <= ones then dselect_1 l i
                 else num + dselect_1 r (i - ones)
  | Bleaf s => select true i s
  end.

Fixpoint dselect_0 (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    let zeroes := num - ones in
    if i <= zeroes then dselect_0 l i
                   else num + dselect_0 r (i - zeroes)
  | Bleaf s => select false i s
  end.

Lemma dselect_1E B i :
  wf_dtree B -> dselect_1 B i = select true i (dflatten B).

Lemma dselect_0E B i :
 wf_dtree B -> dselect_0 B i = select false i (dflatten B).
```

挿入

```
Fixpoint dins (B : dtree) b i w : dtree :=
  match B with
  | Bleaf s =>
    let s' := insert1 s b i in
    if size s + 1 == 2 * (w ^ 2)
    then let n  := (size s') %/ 2 in
         let sl := take n s' in
         let sr := drop n s' in
         Bnode Red (Bleaf _ sl)
               (size sl, rank true (size sl) sl)
               (Bleaf _ sr)
    else Bleaf _ s'
  | Bnode c l (num, ones) r =>
    if i < num then balanceL c (dins l b i w) r
               else balanceR c l (dins r b (i - num) w)
  end.

Definition dinsert (B : dtree) b i w : dtree :=
  match dins B b i w with
  | Bleaf s => Bleaf _ s
  | Bnode _ l d r => Bnode Black l d r
  end.
```

平衡を保つ

- 平衡化の場合の多さが red-black tree の証明の難点
- balanceL に対する場合わけが 11 個のゴールを生成する
- SSReflect らしく，最低限の自動化で対応する

```
Ltac decompose_rewrite :=
  let H := fresh "H" in
  case/andP || (move=>H; rewrite ?H ?(eqP H)).

Lemma balanceL_wf c (l r : dtree) :
  wf_dtree l -> wf_dtree r -> wf_dtree (balanceL c l r).
Proof.
case: c => /= wfl wfr. by rewrite wfl wfr ?(dsizeE,donesE,eqxx).
case: l wfl =>
  [[[[] lll [lln llo] llr|llA] [ln lo] [[] lrl [lrn lro] lrr|lrA]
   |ll [ln lo] lr]|lA] /=;
  rewrite wfr; repeat decompose_rewrite;
  by rewrite ?(dsizeE,donesE,size_cat,count_cat,eqxx).
Qed.
```

依存型による定義

データ構造で不変量を保証する

- 動的ビット列として
- red-black tree として

```
Definition is_black c := if c is Black then true else false.
Definition color_ok parent child :=
    is_black parent || is_black child.

Inductive tree : nat -> nat -> nat -> color -> Type :=
  | Leaf : forall (arr : seq bool),
      (w ^ 2)./ 2 <= size arr < (w ^ 2).*2 ->
      tree (size arr) (count_one arr) 0 Black
  | Node : forall {s1 o1 s2 o2 d cl cr c},
      color_ok c cl -> color_ok c cr ->
      tree s1 o1 d cl -> tree s2 o2 d cr ->
      tree (s1 + s2) (o1 + o2) (d + is_black c) c.
```

依存型での操作

- 基本操作は定義も証明もほとんど変わらず
- dtree_indは要らない
- dinsは Program 環境で定義できた

```
Program Fixpoint dinsert' {n m d c} (B : tree n m d c) (b : bool) i
      {measure (size_of_tree B)} : { B' : near_tree n.+1 (m + b) d c
                 | dflattenn B' = insert1 (dflatten B) b i } := ...
```

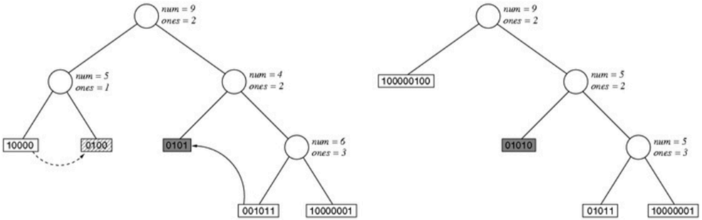
20 個の Obligation が生成され, 全ての性質の証明は 90 行

- balanceL とbalanceRの定義
  - 回避できないバグにより Program 環境が使えなかった
  - 17 行で tactic による定義を完成させた

```
Definition balanceL {nl ml d cl cr nr mr} (p : color)
    (l : near_tree nl ml d cl) (r : tree nr mr d cr) :
    color_ok p (fix_color l) -> color_ok p cr ->
    {tr : near_tree (nl + nr) (ml + mr) (inc_black d p) p
    | dflattenn tr = dflattenn l ++ dflatten r}.
  destruct r as [s1 o1 s2 o2 s3 o3 d' x y z | s o d' c' cc r'].
  + case: p => //= cpl cpr.
  (* さらに 11 行で定義と証明が完成 *)
Defined.
```

削除

- 他の操作に比べて複雑
- 依存型を用いて, 不変量を探した
- あたらしい balance が必要



葉の値の大きさは 5 以上.

依存型の削除

```
Inductive near_tree' : nat -> nat -> nat -> color -> Type :=
| Stay : forall {s o d c} p,
    color_ok c (inv p) ->
    tree s o d c -> near_tree' s o d p
| Down : forall {s o d},
    tree s o d Black -> near_tree' s o d.+1 Black.
Definition balanceL2 {s1 s2 o1 o2 d cl cr} (p : color)
           (dl : near_tree' s1 o1 d cl) (r : tree s2 o2 d cr) :
  color_ok p cl -> color_ok p cr ->
{B' : near_tree' (s1 + s2) (o1 + o2) (inc_black d p) p |
 dflattenn' B' = dflattenn' dl ++ dflatten r}.
(* 47 行の証明 *)
Defined.
Definition ddelete (d: nat) (c: color)
           (num ones : nat) (i : nat)
           (B : tree num ones (inc_black d c) c) :
{ B' : near_tree' (num - (i < num))
       (ones - (daccess B i)) (inc_black d c) c |
 dflattenn' B' = delete (dflatten B) i }.
(* 105 行の証明 *)
Defined.
```

通常型の削除

- 型チェックは通ったので正しいはずだが...
- このままだと大変すぎるので, 型を落とす
- ただし,Extraction はうまく働かなかったので手動で

通常型の削除

```
Inductive deleted_dtree: Type :=
| Stay : dtree -> deleted_dtree
| Down : dtree -> deleted_dtree.
Definition balanceL' col (l : deleted_dtree) (r : dtree) : deleted_dtree :=
match l with
| Stay l => Stay (rbnode col l r)
| Down l =>
match col,r with
| _, Bnode Black (Bnode Red rll _ rlr) _ rr =>
  Stay (rbnode col (bnode l rll) (bnode rlr rr))
| Red, Bnode Black (Bleaf _ as rl) _ rr
| Red, Bnode Black (Bnode Black _ _ _ as rl) _ rr =>
  Stay (bnode (rnode l rl) rr)
| Black,Bnode Red (Bnode Black (Bnode Black _ _ _ as rll) _ rlr) _ rr
| Black,Bnode Red (Bnode Black (Bleaf _ as rll) _ rlr) _ rr =>
  Stay (bnode (bnode (rnode l rll) rlr) rr)
| Black,Bnode Red (Bnode Black (Bnode Red rlll _ rllr) _ rlr) _ rr =>
  Stay (bnode (bnode l rlll) (rnode (bnode rllr rlr) rr))
| Black,Bnode Black (Bleaf _ as rl) _ rr
| Black,Bnode Black (Bnode Black _ _ _ as rl) _ rr =>
  Down (bnode (rnode l rl) rr)
| _,_ => Stay (rbnode col l r)
end
end.
```

通常型の削除

```
Function ddel (B : dtree) (i : nat) { measure height_of_dtree B } :
         deleted_dtree :=
match B  with
| Bnode c (Bleaf l) (s,o) (Bleaf r) => delete_leaves c l r i
| Bnode Black (Bnode Black ll (ls,lo) lr) (s,_) (Bnode Red rl (rs,ro) rr) =>
  let l := Bnode Black ll (ls,lo) lr in
  let r := Bnode Red rl (rs,ro) rr in
  if i < s
  then balanceL' Black (ddel (rnode l rl) i) rr
  else balanceR' Black l (ddel r (i - s))
| Bnode Black (Bnode Red ll (ls,lo) lr) (s,_) (Bnode Black rl (rs,ro) rr) =>
  let l := Bnode Red ll (ls,lo) lr in
  let r := Bnode Black rl (rs,ro) rr in
  if i < s
  then balanceL' Black (ddel l i) r
  else balanceR' Black ll (ddel (rnode lr r) (i - ls))
| Bnode c l (s,_) r =>
  if i < s
  then balanceL' c (ddel l i) r
  else  balanceR' c l (ddel r (i - s))
| Bleaf x => Stay (leaf (delete x i))
end.
(* 4 行の証明 *)
Defined.
```

動的ビット列のまとめと課題

- 通常型による証明
    - うまく SSReflect が利用でき，すっきりした証明
    - 特に，平衡化の証明の場合分けが従来研究より直感的
    - ただ，細々とした補題が多い
- 依存型による証明
    - 欲しい性質が型で表現され，証明の細分化を防ぐ
    - Program 環境のバグにより，読みにくい定義になる
    - 証明の修正が難しい
- 今後の課題
    - 削除も定義・証明できたが，改良の余地あり
    - 複雑さに関する証明も行いたい
      妥当な複雑さの定義が必要

動的ビット列のまとめと課題

- 通常型による証明
  - うまく SSReflect が利用でき，すっきりした証明
  - 特に，平衡化の証明の場合分けが従来研究より直感的
  - ただ，細々とした補題が多い
- 依存型による証明
  - 欲しい性質が型で表現され，証明の細分化を防ぐ
  - Program 環境のバグにより，読みにくい定義になる
  - 証明の修正が難しい
- 今後の課題
  - 削除も定義・証明できたが，改良の余地あり
  - 複雑さに関する証明も行いたい
    妥当な複雑さの定義が必要

https://github.com/affeldt-aist/succinct