Proving tree algorithms for succinct data structures

Reynald Affeldt, Jacques Garrigue, Xuanrui Qi, Kazunari Tanaka

Succinct data structures give space efficient representations of large amounts of data without sacrificing performance. In order to do that they rely on cleverly designed data representations and algorithms. We present here the formalization in Coq/SSReflect of two different succinct tree algorithms. One is the Level-Order Unary Degree Sequence (aka LOUDS), which encodes the structure of a tree in breadth first order as a sequence of bits, where access operations can be defined in terms of Rank and Select, which work in constant time for static bit sequences. The other represents dynamic bit sequences as binary red-black trees, where Rank and Select present a low logarithmic overhead compared to their static versions, and with efficient Insert and Delete. The two can be stacked to provide a dynamic representation of dictionaries for instance. While both representations are well-known, we believe this to be their first formalization and a needed step towards provably-safe implementations of big data.

1 Introduction

Succinct data structures [9] represent combinatorial objects (such as bit vectors or trees) in a way that is space-efficient (using a number of bits close of the information theoretic lower bound) and time-efficient (i.e., not slower than classical algorithms). This topic is attracting all the more attention as we are now collecting and processing large amounts of data in various domains such as genomes or text mining. As a matter of fact, succinct data structures are now used in software products of data-centric companies such as Google (e.g., [7]).

The more complicated a data structure is, the harder it is to process it. A moment of thought is enough to understand that constant-time access to bit representations of trees requires ingenuity. Succinct data structures therefore make for intricate algorithms and their importance in practice make them perfect targets for formal verification (e.g., [13]).

In this paper, we tackle the formal verification of tree algorithms for succinct data structures. We first start by formalizing basic operations such as counting (rank) and searching (select) bits in arrays (see Sect. 3). This is an important step because the theory of these basic operations sustains most succinct data structures. Next, we formally define and verify a bit representation of trees called LOUDS (see Sect. 3). It is for example used in the Mozc Japanese input method by Google [7]. However, like most succinct data structures, this bit representation only deals with static data. Last, we further explore the advanced topic of dynamic bit vectors (see Sect. 4). The implementation of the latter requires to combine static bit vectors from succinct

[*] 簡潔データ構造における木構造アルゴリズムの形式証明について

This is an unrefereed paper. Copyrights belong to the Author(s).

レナルド・アフェルト, 産業技術総合研究所, National Institute of Advanced Industrial Science and Technology.

ジャック・ガリグ　田中一成, 名古屋大学多元数理科学研究科, Nagoya University Grad. School of Mathematics.

Xuanrui Qi, タフツ大学, Tufts University.

data structures with classical balanced trees; we show in particular how to use red-black trees for the purpose of formal verification.

2 Two Functions to Build Them All

The rank and select functions are the most basic blocks to form operations on succinct data structures: rank counts bits while select searches for their position. The rest of this paper (in particular Sect. 3.2 and Sect. 4) explains how they are used in practice to perform operations on trees. In this section, we just briefly explain their formalization and theory.

2.1 Counting Bits with rank

The `rank` function counts the number of elements `b` (most often bits) in the prefix of an array `s` (up to some index `i`). It can be conveniently formalized by means of standard list functions used in functional programming:

```
Definition rank b i s := count_mem b (take i s).
```

The mathematically-inclined reader can alternatively[†1] think of it as the cardinal of the number of indices of `b` bits in a tuple `B`:

```
Definition Rank (i : nat) (B : n.-tuple T) :=
#|[set k : [1,n] | (k <= i) && (tacc B k == b)]|.
```

(`[1,n]` is the type of integers between 1 and n; `tacc` accesses the tuple counting the indices from 1.)

Figure 1 provides concrete examples.

2.2 Finding Bits with select

Intuitively, compared with rank, select performs the converse operation: it returns the *minimum* index of a bit in an array. It is conveniently specified using the `ex_minn` construct of the SSReflect library [5]:

```
Variables (T : eqType) (b : T) (n : nat).
Lemma select_spec (i : nat) (B : n.-tuple T) :
  exists k, ((k <= n) && (Rank b k B == i)) ||
    (k == n.+1) && (count_mem b B < i).
Definition Select i (B : n.-tuple T) :=
  ex_minn (select_spec i B).
```

†1 This is actually the definition that appears in Wikipedia at the time of this writing.

With this definition, select returns the index of the sought bit *plus* 1 (counting indices from 0); selecting the 0th bit always returns 0; when no adequate bit is found, select returns the size of the array plus 1.

Figure 2 illustrates the select function and can be compared with Fig. 1.

2.3 The Theory of rank and select

The rank and select functions are used in a variety of applications whose formal verification naturally calls for a shared library of lemmas. Our first work is to identify and isolate this theory. Its lemmas are not all difficult to prove. For instance, the fact that `Rank` cancels `Select` directly follows from the definitions:

```
Lemma SelectK n (s : n.-tuple T) (j : nat) :
  j <= count_mem b s ->
    Rank b (Select b j s) s = j.
```

However, as often with formalization, it requires a bit of work and try-and-error to find out the right definitions and the right lemmas to put in the theory of rank and select. For example, how appealing the definition of `Select` above may be, proving its equivalence with a functional version such as

```
Fixpoint select i (s : seq T) : nat :=
  if i is i.+1 then
    if s is a :: s' then
      (if a == b then select i s'
       else select i.+1 s').+1
    else 1
  else 0.
```

turns out to add much comfort to the development of related lemmas.

As a consequence, the resulting theory of rank and select sometimes looks technical and we therefore refer the reader to the source code [2] to better appreciate its current status. Here, we just provide for the sake of completeness the definition of two derived functions that are used later in this paper.

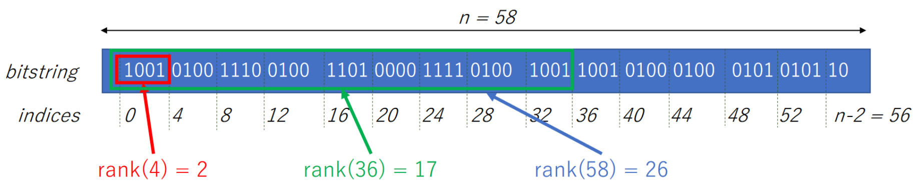

Figure 1 Illustration of the rank function

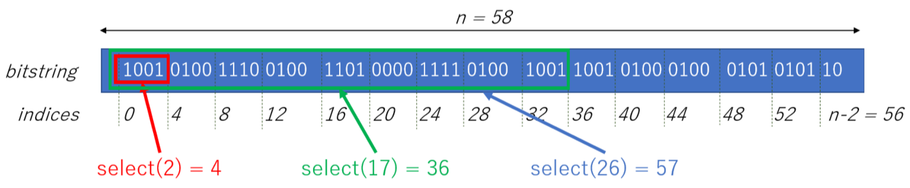

Figure 2 Illustration of the select function

2.3.1 The succ Function

In a bitstring, the succ function computes the position of the next 0-bit or 1-bit. It will find its use when dealing with LOUDS operations (see Sect. 3.2.2). More precisely, given a bitstring `s`, `succ b s y` returns the index of the next `b` following index `y`. This operation can be achieved by a combination of rank and select. First, a call to rank counts the number of `b`'s up-to index `y`; let `N` be this number. Second, a call to select searches for the `(N + 1)`th `b` (*p. 89 of [9]*):

```
Definition succ (b : T) (s : seq T) y :=
  select b (rank b y.-1 s).+1 s.
```

In particular, there is no `b` in the set $\{s_i | y \leq i < \texttt{succ b s y}\}$:

```
Lemma succP b n (s : n.-tuple T) (y : [1, n]) :
  b \notin
    \bigcup_(i : [1,n] | y <= i < succ b s y)
       [set tacc s i].
```

2.3.2 The pred Function

The pred function computes the position of the previous bit and will find its use in Sect. 3.2.3. It is similar to the succ function above so that we only provide its implementation for reference:

```
Definition pred (b : T) (s : seq T) y :=
  select b (rank b y s) s.
```

3 LOUDS Formalization

Operationally, a LOUDS (Level-Order Unary Degree Sequence) encoding consists in turning a tree into an array of bits via a breadth-first traversal. Figure 3 provides a concrete example. The resulting array is the ordered concatenation of the bit representation of each node. Each node is represented by a list of bits that contains as many 1-bits as there are children and that is terminated by a 0-bit.

The significance of the LOUDS encoding is that it preserves the branching structure of the tree without pointers: this makes therefore for a compact representation of trees in memory. Moreover, read-only operations can be implemented by means of rank and select, for which there exist constant-time implementations.

In this section, we explain how we formalize the LOUDS encoding (Sect. 3.1) and how we formally verify the correctness of operations on trees built out of rank and select (Sect. 3.2).

3.1 LOUDS Encoding Formalized in Coq

We define arbitrarily-branching trees by an inductive type:

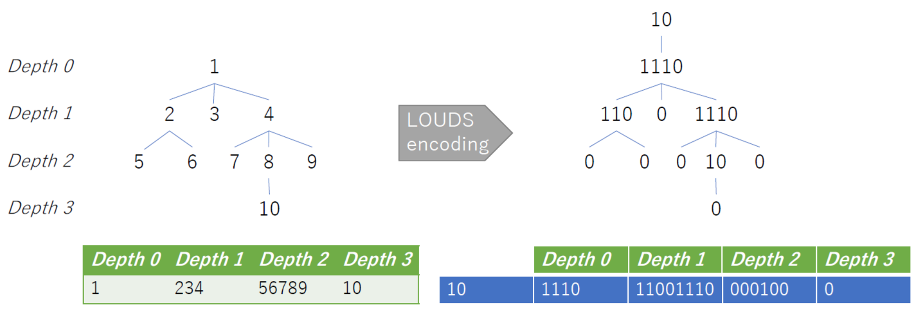

Figure 3 LOUDS encoding of a tree (without the labels)

```
Variable A : Type.
Inductive tree := Node : A -> seq tree -> tree.
```

The type `A` is for labels. With this defintion, a leaf is a node with an empty list of children. For example, the tree of Fig. 3 becomes:

```
Definition t : tree nat := Node 1
[:: Node 2 [:: Node 5 [::]; Node 6 [::]];
    Node 3 [::];
    Node 4 [:: Node 7 [::];
               Node 8 [:: Node 10 [::]];
               Node 9 [::]]].
```

For the sake of presentation, let us introduce the definition of a *forest* as a list of trees:

```
Definition forest := seq tree.
```

We formalize level-order traversal of a tree by recursion on its height. The recursion itself is defined by an intermediate function that applies more generally to a forest (parameter `l` below):

```
Fixpoint lo_traversal''
  (f : forest A -> seq B) n (l : forest A) :=
  if n is n'.+1 then
    f l ++
    lo_traversal'' f n' (children_of_forest l)
  else
    [::].
```

The parameter `n` is expected to be filled with the maximum height of the forest. The definition is furthermore parameterized by an arbitrary function `f` for generality. Level-order traversal is obtained by instantiating `lo_traversal''` appropriately:

```
Variable (f : forest A -> seq B).
Definition lo_traversal' n (l : forest A) :=
  lo_traversal'' (flatten \o map f \o
    @children_of_forest' _) n l.
Definition lo_traversal t :=
  lo_traversal' (height t) [:: t].
```

The functions (`height`, etc.) used for the instantiation are as expected (and not displayed to save space). Just observe that the definition is still parameterized by some function `f`.

Finally, the LOUDS encoding is obtained by (1) instantiating `lo_traversal` with an appropriate function (the so-called *node description* of a node), and (2) prepending an artificial root (this is a technical convenience, see *p. 212 of [9]*):

```
Definition node_description l :=
  rcons (nseq (size l) true) false.
Definition LOUDS (t : tree A) :=
  [:: true, false &
    lo_traversal node_description t].
```

For example, we can recover the encoding displayed in Fig. 3 with this definition of `LOUDS`:

```
Lemma LOUDS_t : LOUDS t = [:: true; false; true;
  true; true; false; true; true; false; false;
  true; true; true; false; false; false; false;
  true; false; false; false].
```

3.2 LOUDS Functions using rank and select

In this section, we explain how we formalize LOUDS functions and prove their correctness. These functions are essentially built out of rank and select. Their correctness statements typically establish a correspondence between operations on trees defined inductively and operations on their LOUDS encoding. We start by explaining how we represent positions in trees and then comment on the formal verification of LOUDS operations using representative examples.

3. 2. 1 Positions in Trees

For a tree defined inductively, the representation of the position of a node is textbook: using a *path*, i.e., a list that records the branches taken from the root to reach the node. For example, the position of the node 8 in Fig. 3 is `[:: 2; 1]`. Not all positions are valid, we sort out the valid ones by means of a predicate (`valid_position`, omitted for brevity).

In contrast, the position of nodes in the LOUDS encoding is not immediate. In our formalization, it is computed by the following function:

```
Definition LOUDS_position
  (t : tree A) (p : seq nat) :=
  (count_smaller t p +
   (count_smaller t (rcons p 0)).-1).+2.
```

`count_smaller` is an intermediate function that counts the number of nodes appearing before during the level-order traversal. Here, the first occurrence of `count_smaller` counts the number of nodes (or equivalently the number of 0-bits) before the position `p`. The second occurrence counts the number of nodes before the position of the first child (i.e., `rcons p 0`; whether this child exists or not does not matter here): this is almost the number of 1-bits before the position `p` (`.-1` accounts for the fact that we have counted the root node, which is nobody's child). `.+2` is for the artificial root prepended by the LOUDS encoding (see Sect. 3. 2. 1).

For example, in Fig. 3, the position of the node 8 is `[:: 2; 1]` in the inductively defined tree and `17` in the LOUDS encoding:

```
Definition p8 := [:: 2; 1].
Eval compute in LOUDS_position t p8.
  (* 17 *)
Eval compute in count_smaller t p8.
  (* 7 *)
Eval compute in count_smaller t (rcons p8 0).
  (* 9 *)
```

For illustration, we also display the intermediate results of evaluating the `count_smaller` function whose code can be found online [2].

3. 2. 2 Number of Children using succ

In this section, we explain how to verify the LOUDS function that counts the number of children of a node. For a tree defined inductively, this operation can be achieved by first walking down the path to the node and then looking at the list of its children. The formalization is unsurprising:

```
Definition children_of_node (t : tree) :=
  let: Node _ l := t in l.
Fixpoint subtree (t : tree) (p : seq nat) :=
  match p with
  | nil => t
  | n :: p' =>
    subtree (nth t (children_of_node t) n) p'
  end.
Definition children (t : tree) (p : seq nat) :=
  size (children_of_node (subtree t p)).
```

To count the number of children of a node using a LOUDS encoding, one first has to notice that in this representation each node is terminated by a 0-bit. Given such a 0-bit (or equivalently the corresponding node), one can therefore find the number of children by computing the distance with the next 0-bit (see *p.214 of [9]*). Finding the next 0-bit is the purpose of the `succ` function (see Sect. 2. 3. 1):

```
Definition LOUDS_children (B : bitseq) v :=
  succ false B v.+1 - v.+1.
```

`LOUDS_children` is correct because, when applied to the `LOUDS_position` of a position `p`, it produces the same result as the function `children`:

```
Theorem LOUDS_childrenE
  (t : tree A) (p : seq nat) :
  let B := LOUDS t in
  valid_position t p ->
  children t p =
    LOUDS_children B (LOUDS_position t p).
```

3. 2. 3 Parent Node using rank and select

A path in a tree defined inductively gives direct ancestry information. In particular, the penultimate element denotes the parent. It takes more ingenuity to find the parent using a LOUDS representation and functions from Sect. 2 alone. The idea is to count the number of nodes and branches up-to the position in question. More precisely, given a LOUDS position `v`, Let `Nv` be the number of nodes up to `v` (`rank false v B` computes this number). Then, `select true Nv B` looks for the `Nv`th down-branch, which is the branch leading to the node of position `v`. Last, this branch belongs to a node whose position can be recovered using the `pred` function (from Sect. 2.3.2). This leads to the following definition (see *p.215 of [9]*):

```
Definition LOUDS_parent (B : bitseq) v :=
  let j := select true (rank false v B) B in
  pred false B j.
```

One can check the correctness of `LOUDS_parent` as follows. Consider a node reached by the path `rcons p x`. Its parent is the node reached by the path `p`. We can formally prove that the LOUDS position of `p` and the position computed by `LOUDS_parent` coincide:

```
Theorem LOUDS_parentE (t : tree A) p x :
  valid_position t (rcons p x) ->
  LOUDS_parent (LOUDS t)
    (LOUDS_position t (rcons p x)) =
  LOUDS_position t p.
```

The approach that we explained so far shows how to carry out the formal verification of the LOUDS operations that are listed in *Table 8.1 of [9]*. However, how useful they may be for many big-data applications, these operations assume static compact data structures. The next section (Sect. 4) explains how to extend our approach to deal with dynamic structures.

4 Dynamic vectors

4.1 Representing Dynamic Vectors

All of the work described above are done in the context of static, immutable bit vectors, but in some real-life applications bit vectors need to support dynamic operations—not just static queries—on them. In our formalization of such *dynamic* bit vectors, we implemented and verified three dynamic operations: inserting a bit into the bit vector, setting a bit (i.e., set the value of the bit at a certain position to 1), and clearing bit (set the value of a bit to 0).

Insertion into a linear array has time complexity $O(n)$, but we can improve this by using a balanced binary search tree to represent the bit array, which will enable us to handle insertions in $\mathbf{max}(O(w), O(\log n))$ time, with a trade-off of $O(n/w)$ bits of extra space, where w is a parameter controlling the width of each tree node [9].

In our formalization of the dynamic bit vector's insert algorithm, we used a red-black tree as our balanced tree structure, as not only multiple purely functional implementations [6, 11], but also several Coq formalizations [3, 4], of red-black trees already exist. While it was useful to refer to previous formalizations of red-black trees, due to the difference of stored contents, we had to reimplement them. Namely, the above formalizations of red-black trees deal with sets, and maintain the ordering invariant, while our trees represent vectors, and maintain both that the contents (as concatenation of the leaves) are unchanged, and that meta-data in inner nodes is correct.

Incidentally, we believe that our formalization of tree algorithms for dynamic bit vectors is also the first Coq formalization of red-black trees using the SSReflect proof language and library. Using SSReflect, our formalization becomes significantly clearer than the other formalizations, as SSReflect provides us with many useful functionalities, such as boolean reflection and intro-patterns, which allow us to write statements and proofs in a readable and user-friendly way: for example, inductive properties of trees are no longer represented by `Inductive`, but now by simple boolean propositions (i.e., functions returning `bool`), whose validity can be checked by simple case analysis.

We experimented with two different representations of red-black trees: one with simple polymorphic algebraic data types, similar to what one would use in a typical OCaml or Haskell implementation, as well as another with stronger type-level guarantees using dependent types. We refer to the former representation as the *simply-typed* variation, and the latter as the *richly-typed* variation.

We represent our bit vector as a red-black tree, where each node holds a color and meta-data about the bit vector, and each leaf holds a *flat* (i.e., list-based) bit array. In the simply-typed version, we store two natural numbers in each node, namely the size and the rank of the left subtree:

```
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).
```

Intuitively, one would prove properties of tree algorithms using induction, but the inductive hypothesis that comes with our inductive definition, `btree_ind`, is not strong enough to prove many interesting properties of our tree. We could, however, note that the numbers encoded in each node are *in fact* the left child's size and rank, and we capture this property using a boolean proposition, `wf_dtree`:

```
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 _ => true
end.
```

With this well-formedness property, we could prove a more powerful inductive hypothesis, `dtree_ind`, which in turn allows us to prove more interesting properties of our algorithm:

```
Lemma dtree_ind (P : dtree -> Prop) :
  (forall c (l r : dtree) 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, P (Bleaf _ s)) ->
  forall B, wf_dtree B -> P B.
```

In the richly-typed version, we encode the size, rank and color of the tree, as well as the *black-depth* (or "black-height") of the tree [3] (i.e., the number of black nodes on the path from the root) into the type itself. Constructing a new node, thus, requires providing a "well-coloredness" proof of the node and its children:

```
Inductive color := Red | Black.

Definition color_ok parent child : bool :=
  match parent,child with
  | Red,Red => false
  | _,_ => true
  end.

Inductive param (A : Type) : Prop :=
  Param : A -> param A.

Definition inc_black d c :=
  match c, d with
  | Black, Param n => Param n.+1
  | _, _ => d
  end.

Inductive tree : nat -> nat -> param nat -> color
  -> Type :=
| Leaf : forall (arr : seq bool),
    tree (size arr) (count_one arr) (Param 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) (inc_black d c) c.
```

The type of the constructor `Node` enforces the invariants of red-black trees [11], which will be discussed in Sect. 4.3. Since the major properties of the tree have already been encoded in the type and the constructors of the dependent tree, the regular inductive hypothesis `tree_ind` suffices. Note that we wrap the black-depth of the tree in a type `param` of sort `Prop`, so that extraction erases this extra parameter, which is only used in the formalization and proofs, and is not necessary for executing the algorithms.

4.2 Verifying Basic Queries

The basic query operations, rank and select, could be easily defined via traversal of the tree, and we implement these queries as three Coq functions, `drank`, `dselect_1` and `dselect_0`, corresponding to the queries rank, select_1, and select_0, respectively.

The rank query (see Sect. 2. 1 for details), for example, is implemented as following in the simply-typed variation (the implementation for the richly-typed variation is very similar).

```
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.
```

The select_1 and select_0 queries are implemented similarly. To prove the correctness of our implementation, however, we need to define a "flatten" function that converts our tree representation of a bit vector to a flat representation of the vector:

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

With this `dflatten` function, we could prove that our functions `drank`, `dselect_1` and `dselect_0` actually compute the queries rank, select_1, and select_0, proceeding by induction. In the simply-typed version, we may use the `dtree_ind` lemma defined above. Note that our implementation is only correct on well-formed trees:

```
Lemma drankK (B : dtree) i : wf_dtree B ->
  drank B i = rank true i (dflatten B).
Proof.
move=> wf; move: B wf i.
apply: dtree_ind => // c l r num ones -> -> _
IHl IHr i /=.
(* the rest of the proof omitted *)
Qed.
```

For the richly-typed version, however, the signature of our data constructor ensures that all trees are well-formed, and thus regular induction by `elim` will suffice:

```
Lemma drankK nums ones d c (B: tree nums ones d c)
  i : drank B i = rank true i (dflatten B).
Proof.
elim: B i => //= lnum o1 s2 o2 d0 cl cr c0 i i0 l
                 IHl r IHr x.
by rewrite rank_cat -dflatten_size IHl -IHr
  -dflatten_rank.
Qed.
```

Other queries could be verified similarly.

4. 3 Implementing and Verifying Insertion

4. 3. 1 The Simply-Typed Variation

Insertion is significantly harder to implement than the static queries. While each of the static queries only takes around a dozen lines to implement and verify, the Coq implementation and proofs for insertion is on the order of 300-400 lines in both versions.

For the simply-typed variation, we translate the algorithm given by [9] directly into Coq.

```
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.
```

where `balanceL` and `balanceR` balance an unbalanced red-black tree [3,11], fixing imbalances occuring on the left and on the right, respectively. See [11] for the conventional method of balancing purely functional red-black trees. `dinsert` is a simple wrapper over `dins` that completes the insertion by painting the root black.

Verifying `dinsert` requires verifying three different properties: `dinsert` must preserve the data, must return a balanced red-black tree, and must also maintain the well-formedness property.

The first part is relatively simple: about 30 lines, including lemmas for `balanceL`, `balanceR` and `dins`.

```
Lemma dinsertK (B : dtree) b i w :
  wf_dtree B -> dflatten (dinsert B b i w) =
  insert1 (dflatten B) b i.
```

The more interesting (and also important) part, however, is proving that `dinsert` never breaks the red-black tree invariant; more importantly, we want to eliminate cases where the "height balance" at a node is broken. It is easy to model the property that no red node has a red child; the "height balance" property is modeled using the black-depth. We can thus model the red-black tree invariant with a recursive function, `is_redblack`:

```
Fixpoint is_redblack (B : dtree) ctxt bh :=
  match B with
  | Bleaf _ => bh == 0
  | Bnode c l _ r =>
    match c, ctxt with
    | Red, Red => false
    | Red, Black => is_redblack l Red bh
                    && is_redblack r Red bh
    | Black, _ => (bh > 0)
                  && is_redblack l Black (bh.-1)
                  && is_redblack r Black (bh.-1)
    end
  end.
```

where `ctxt` is the "color context", or the color of the parent's node, and `bh` is the black-depth of the node.

Splitting a leaf will introduce a red node, which may break this invariant if its parent was already red. While the balancing functions will fix that, they may have to push up the red-red conflict to avoid raising the black depth. To inductively verify `dins`, we introduce a weaker invariant, saying that, when the original parent was black, sub-trees returned by `dins` would be red-black if their root node were painted black [3], as modeled by the `nearly_redblack` property:

```
Definition nearly_redblack (B : dtree) bh :=
  match B with
  | Bnode Red l _ r => is_redblack l Black bh
                       && is_redblack r Black bh
  | _ => is_redblack B Black bh
  end.
```

Once again, we may prove properties of `dins` relatively straightforwardly, proceeding by case analysis, but the parts involving `balanceL` and `balanceR` require special attention.

```
Lemma balanceL_Black_nearly_is_redblack l r n :
  nearly_redblack l n -> is_redblack r Black n ->
  is_redblack (balanceL Black l r) Black n.+1.
Proof.
case: l => [[[[] lll llD llr|llA] lD [] lrl lrD
           lrr|lrA]|ll lD lr]|lA] /=;
  repeat decompose_rewrite => //;
  by rewrite !is_redblack_Red_Black -?(eqP H1).
Qed.

Lemma balanceR_Black_nearly_is_redblack l r n :
  nearly_redblack r n -> is_redblack l Black n ->
  is_redblack (balanceR Black l r) Black n.+1.

Lemma dins_is_redblack (B : dtree) b i w n :
  (is_redblack B Black n -> nearly_redblack
    (dins B b i w) n) /\
  (is_redblack B Red n -> is_redblack
    (dins B b i w) Black n).
```

The case analysis in `balanceL_Black_nearly_is_redblack` generates 11 distinct cases, each of them having complex hypotheses and goals. Specifically, hypotheses and goals will be stated in terms of boolean conjunctions, thus we defined a tactic that simplifies these goals by destructing all boolean conjunctions in the premises of the goal into separate hypotheses, and attempting to rewrite the goal using each of them:

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

Note that `try rewrite (eqP H)` is essential, as many of the generated hypotheses will be in the form of SSReflect boolean equalities, and it is necessary to reflect them into the corresponding Coq equality using `eqP` before the call to `rewrite`. Thanks to `decompose_rewrite`, the proof of `balanceL_Black_nearly_is_redblack`, and that of similar propositions, could be kept simple.

Finally, we may prove that `dinsert` preserves the red-black tree invariant by noting that `dinsert` repaints the root such that “nearly red-black” trees would become fully red-black after the repainting. Also, note that `dinsert` will increase the black-depth of the tree by 1 if the root node is red after the call to `dins`; however, as we are not interested in the specific value of the tree's black-depth, we use an existential qualifier to “abstract over” the value of the black-depth.

```
Lemma dinsert_is_redblack (B : dtree) b i w n :
  is_redblack B Red n -> exists n',
    is_redblack (dinsert B b i w) Red n'.
Proof.
exists (if (dins B b i w) is Bnode Red _ _ _
        then n + 1 else n).
(* the rest omitted *)
Qed.
```

Of course, this is not the end of our proof of correctness for `dinsert`, as we still need to verify that `dinsert` preserves the well-formedness of trees. This is essential, as queries on non-well-formed trees will not give the expected result! Since the proof procedure is broadly similar to what we described above for red-black tree invariants (again using `decompose_rewrite`), we will not get into proof details.

```
Lemma dinsert_wf (B : dtree) b i w :
  wf_dtree B -> wf_dtree (dinsert B b i w).
```

4.3.2 The Richly-Typed Variation: An Experiment in Dependently-Typed Programming in Coq

Our richly-typed representation of dynamic bit vectors enforces some very strong correctness predicates using dependent types, but this also makes programming `dinsert` much more complex. Hopefully, Coq's `Program` framework [12] should assist us in writing dependently-typed programs in a clear and intuitive fashion. However, `Program` is not always stable, and can sometimes reject perfectly reasonable programs, or even fail due to internal bugs. In order to improve the readability of our code, we use Gallina (often with the aid of `Program`) whenever possible, but occasionally our only practical option was to build programs using Ltac.

Since `dinsert` can break the invariants of `tree` in its intermediate steps (red nodes can have red children), we will need to introduce an “intermediate tree” representation that corresponds to `nearly_redblack`:

```
Inductive near_tree : nat -> nat -> param nat
  -> color -> Type :=
| Bad : forall {s1 o1 s2 o2 s3 o3 d},
    tree s1 o1 d Black ->
    tree s2 o2 d Black ->
    tree s3 o3 d Black ->
    near_tree (s1 + s2 + s3) (o1 + o2 + o3)
              d Red
| Good: forall {s o d c} p,
    tree s o d c ->
    near_tree s o d p.
```

We may observe that `near_tree` is a tree representation which allows at most one red node to have at most one red parent. As a result, this representation lets us implement `balanceL` and `balanceR` rather naturally. First, we define some helper functions required to implement them.

```
Definition fix_color {nl ml d c}
                     (l : near_tree nl ml d c) :=
  match l with
  | Bad _ _ _ _ _ _ _ _ _ _ => Red
  | Good _ _ _ _ _ _ => Black
  end.

Definition dflattenn {n m d c}
                     (B : near_tree n m d c) :=
  match B with
  | Bad _ _ _ _ _ _ _ x y z =>
    dflatten x ++ dflatten y ++ dflatten z
  | Good _ _ _ _ _ t => dflatten t
  end.
```

As `balanceL`'s type enforces complex invariants, it would be tedious to implement this function directly in Gallina. Unfortunately, `Program` was of no help here, all our attempts at using it ultimately failing with various kinds of errors. Thus, we ended up implementing `balanceL` using the Ltac proof language, as it support pattern matching over dependent algebraic data types better than Gallina.

```
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 l as [s1 o1 s2 o2 s3 o3 d' x y z
              | s o d' c' cc l'].
(* l is bad *)
-case: p => //= cpl cpr.
 rewrite -(addnA (s1 + s2)) -(addnA (o1 + o2)).
 exists (Good Black (rnode (bnode x y)
                          (bnode z r))).
 by rewrite /= !catA.
(* l is good *)
-case: p => /= cpl cpr; last
           by exists (Good Black (bnode l' r)).
 case Hc': c' in cpl.
 (* bad pattern (c' and p are red) *)
 +destruct l' as
     [|? ? ? ? ? cl' cr' c' ? ? l'1 l'2] => //.
  subst c'; destruct cl', cr', cr => //.
  exists (Bad l'1 l'2 r).
  by rewrite /= !catA.
 (* otherwise *)
 +subst c'; destruct cr => //.
  by exists (Good Red (rnode l' r)).
Defined.
```

Note that the types of balanceL include not only the well-formedness of the resulting tree, but also correctness as stated by balanceLK (using a subset type); thus, our implementation of balanceL is correct by construction. balanceR is defined symmetrically.

With balanceL and balanceR implemented, we may transcribe the insert algorithm into Coq and obtain a formalization of dinsert' (equivalent to dins in the simply-typed formalization), with the aid of Program:

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

  match B with
  | Leaf 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
         Good c (rnode (Leaf sl) (Leaf sr))
    else Good c (Leaf s')
  | Node s1 o1 s2 o2 d cl cr _ okl okr l r =>
    if i < s1
    then proj1_sig (balanceL c
                     (dinsert' l b i w)
                     r  _ okr)
    else proj1_sig (balanceR c l
                     (dinsert' r  b
                        (i - s1) w)
                     okl _)
  end.
```

where size_of_tree B denotes the number of leaves contained in B.

Finally, we paint the root node black using a helper function, real_tree, and obtain a well-formed red-black tree that represents the bit vector after insertion:

```
Definition real_tree {nl ml d c}
  (t : near_tree nl ml d c) : tree nl
    ml (inc_black d (inv (fix_color t)))
    (black_of_bad t) :=
  match t with
  | Bad _ _ _ _ _ _ _ _ x y z =>
    bnode (rnode x y) z
  | Good _ _ _ _ _ t' => t'
  end.

Definition dinsert {n m d c}
  (B : tree n m d c) (b : bool)
  (i : nat) (w : nat) :=
  real_tree (proj1_sig (dinsert' B b i w)).

Lemma real_treeK {nl ol d c}
  (t : near_tree nl ol d c) :
  dflatten (real_tree t) = dflattenn t.

Lemma dinsertK {n m d c} (B : tree n m d c)
  b i w : dflatten (dinsert B b i w) =
  insert1 (dflatten B) b i.
```

5 Related Work

One can find a Coq formalization of a constant-time, $o(n)$-space rank function in [13], which is actually our starting point. The main focus of [13] is the extraction of efficient OCaml code for rank.

This topic is further discussed in [14] where efficient C code is extracted for the same rank function. In comparison, our work focuses on extending the toolset for succinct data structures with more functions (select, succ, etc.) and dynamic structures. Yet, we do plan to enable code extraction for the functions we have been verifying.

Dynamic bit vectors are represented using balanced binary search trees, and any type of height-balanced binary search tree could in principle be used [9]. There are multiple purely-functional balanced binary search tree implementations, such as purely functional AVL trees [8] and Adams trees [1], but purely functional red-black trees are by far the most common, most widely studied, and easiest to implement. Moreover, they have already been formalized using interactive theorem provers such as Coq [3, 4], Agda, and Isabelle [10].

We found Appel's formalization [3] easiest to work with, thus we formulate our invariants and propositions in a way broadly similar to his [3], but our style of proof is drastically different. Specifically, Appel's formalization prefers using "domain-specific" tactics and proof automation by Ltac hacking [3], but we generally prefer adopting the more uniform SSReflect approach of relying on rewriting and reduction to solve goals. Appel's proof style results in slightly shorter proofs, but our proofs seem easier to read and to step-through, and are more robust—Ltac is often opaque and difficult to debug.

6 Conclusion

In this paper, we have reported on an on-going effort to formalize succinct data structures. We started with a foundational theory of the rank and select functions for counting and searching bits in arrays. Using this theory, we have shown how one can formalize a standard compact representation of trees (the LOUDS representation) and prove the correctness of its basic operations. Last, we have almost completed the formalization of dynamic vectors: an advanced topic in succinct data structures.

Our work is a first step towards the construction of a formal theory of succinct data structures. We already overcame several technical difficulties when dealing with LOUDS trees: it took much care to deal with non-structural recursions and to sort out the off-by-one conditions when specifying basic operations. Similarly, the formalization of dynamic vectors was not the matter of bringing an existing formalization of balanced trees so as to extend operations on static data structures. Instead, it led to investigate in detail the literature on the formalization of red-black trees.

We have not yet formalized deletion for our bit vectors, due to the algorithm's complexity. The high-level algorithm for delete is already complex, and it becomes even more complex when translated into Coq [9]. Thus, we intend to investigate a modular way to formalize delete, such that we can solve the difficulties one by one.

Acknowledgements

We acknowledge the support of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" and thank the project participants, in particular Akira Tanaka for his comments on code extraction.

References

[1] Adams, S.: Functional Pearls: Efficient sets–a balancing act, *Journal of Functional Programming*, Vol. 3, No. 4(1993), pp. 553–561.

[2] Affeldt, R., Garrigue, J., Qi, X., and Tanaka, K.: A Coq formalization of succinct data structures, `https://github.com/affeldt-aist/succinct`, 2018.

[3] Appel, A. W.: Efficient Verified Red-Black Trees, September 2011. `http://www.cs.princeton.edu/~appel/papers/redblack.pdf`.

[4] Chlipala, A.: *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, The MIT Press, 2013.

[5] Gonthier, G., Mahboubi, A., and Tassi, E.: A Small Scale Reflection Extension for the Coq sys-

tem, Technical report, INRIA, 2008. Version 17 (Nov 2016).

[6] Kahrs, S.: Red-black trees with types, *Journal of Functional Programming*, Vol. 11, No. 4(2001), pp. 425–432.

[7] Kudo, T., Hanaoka, T., Mukai, J., Tabata, Y., and Komatsu, H.: Efficient dictionary and language model compression for input method editors, *Proceedings of the Workshop on Advances in Text Input Methods (WTIM 2011)*, 2011, pp. 19–25.

[8] Myers, E. W.: Efficient Applicative Data Types, *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84), Salt Lake City, Utah, USA, January 15–18, 1984*, ACM, 1984, pp. 66–75.

[9] Navarro, G.: *Compact Data Structures: A Practical Approach*, Cambridge University Press, 1st edition, 2016.

[10] Nipkow, T.: Automatic Functional Correctness Proofs for Functional Search Trees, *Interactive Theorem Proving (ITP 2016)*, Blanchette, J. and Merz, S.(eds.), Lecture Notes in Computer Science, Vol. 9807, 2016, pp. 307–322.

[11] Okasaki, C.: *Purely Functional Data Structures*, Cambridge University Press, 1998.

[12] Sozeau, M.: Subset Coercions in Coq, *Proceedings of the 2006 International Conference on Types for Proofs and Programs (TYPES'06), Nottingham, UK, April 18–21, 2006*, Altenkirch, T. and McBride, C.(eds.), Lecture Notes in Computer Science, Vol. 4502, Springer-Verlag, 2007, pp. 237–252.

[13] Tanaka, A., Affeldt, R., and Garrigue, J.: Formal Verification of the rank Algorithm for Succinct Data Structures, *18th International Conference on Formal Engineering Methods (ICFEM 2016), Tokyo, Japan, November 14–18, 2016*, Lecture Notes in Computer Science, Vol. 10009, Springer, Nov 2016, pp. 243–260.

[14] Tanaka, A., Affeldt, R., and Garrigue, J.: Safe Low-level Code Generation in Coq using Monomorphization and Monadification, *Journal of Information Processing*, Vol. 26(2018), pp. 54–72.