Proving tree algorithms for succinct data structures

Reynald Affeldt [ID]
National Institute of Advanced Industrial Science and Technology, Japan
reynald.affeldt@aist.go.jp

Jacques Garrigue [ID]
Graduate School of Mathematics, Nagoya University, Japan
garrigue@math.nagoya-u.ac.jp

Xuanrui Qi [ID]
Department of Computer Science, Tufts University, United States
xqi01@cs.tufts.edu

Kazunari Tanaka [ID]
Graduate School of Mathematics, Nagoya University, Japan
tanaka.kazunari@k.mbox.nagoya-u.ac.jp

— Abstract

Succinct data structures give space-efficient representations of large amounts of data without sacrificing performance. They rely on cleverly designed data representations and algorithms. We present here the formalization in Coq/SSReflect of two different tree-based succinct representations and their accompanying algorithms. One is the Level-Order Unary Degree Sequence, 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 balanced trees, where Rank and Select present a low logarithmic overhead compared to their static versions, and with efficient insertion and deletion. 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 [15] represent combinatorial objects (such as bit vectors or trees) in a way that is space-efficient (using a number of bits close to 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 [12].

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 [24].

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. 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 Level-Order Unary Degree Sequence (hereafter LOUDS). It is for example used in the Mozc Japanese input method [12]. The challenge there is that this representation is based on a level-order (i.e., breadth-first) traversal of the tree, which is difficult to describe in a structural way. Nonetheless, like most succinct data structures, this bit representation only deals with static data. Last, we further explore the advanced topic of dynamic bit vectors. The implementation of the latter requires to combine static bit vectors from succinct data structures with classical balanced trees. We show in particular how this can be formalized using a flavor of red-black trees where the data is in the leaves (rather than in the internal nodes, as in most functional implementations).

In both cases, our code can be seen as a verified functional specification of the algorithms involved. We were careful to use the right abstractions in definitions so that this specification could be easily translated to efficient code using arrays. For LOUDS we only rely on the rank and select functions; we have already provided an efficient implementation for rank [24]. For dynamic bit vectors, while the code we present here is functional, it closely matches the algorithms given in [15]. We did prove all the essential correctness properties, by showing the equivalence of each operation with its functional counterpart (functions on inductive trees for LOUDS, and on sequences of bits for dynamic bit vectors).

Independently of this verified functional specification, we identify two technical contributions, that arised while doing this formalization. One is the notion of level-order traversal up to a path in a tree, which solves the challenge of performing path-induction on a level-order traversal. Another is our experience report with using small-scale reflection to prove algorithms on inductive data, which we hope could provide insights to other researchers.

The rest of this paper is organised as follows. The next section introduces rank and select. Section 3 describes our formalization of LOUDS, including the notion of level-order traversal up to a path. Section 4 uses trees to represent bit vectors, defining not only rank and select, but also insertion and deletion. Section 5 reports on our experience. Section 6 compares with the litterature, and Section 7 concludes.

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 (i.e., up to some index `i`) of an array `s`. It can be conveniently formalized using standard list functions:

length $n = 58$

bitstring   1001 0100 1110 0100 1101 0000 1111 0100 1001 1001 0100 0100 0101 0101 10

bit indices   0   4   8   12   16   20   24   28   32   36   40   44   48   52   56

rank(4) = 2   rank(36) = 17   rank(58) = 26
select(2) = 4   select(17) = 36   select(26) = 57
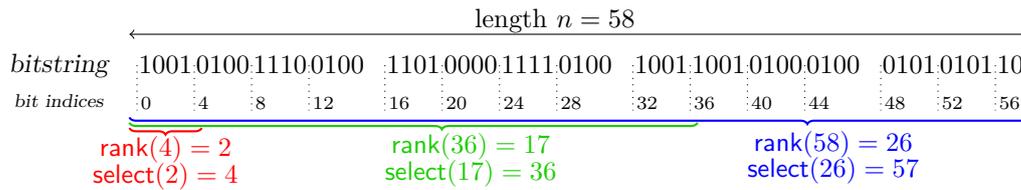
■ Figure 1 Examples of rank and select queries on a sample bitstring (bit indexed from 0 to 57)

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

Figure 1 provides several examples of rank queries. The mathematically-inclined reader can alternatively[1] think of rank 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)]|.
```

In this definition, n.-tuple T denotes[2] sequences of T of length n; [1,n] is the type of integers between 1 and n; and tacc accesses the tuple counting the indices from 1.

2.2   Finding bits with select

Intuitively, compared with rank, select performs the converse operation: it returns the index of the i-th occurrence of b, i.e., the *minimum* index whose rank is i. It is conveniently specified using the ex_minn construct of the SSREFLECT library [8]:

```
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).
```

With this definition, select returns the index of the sought bit *plus one* (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 one. The need for the 0 case explains why it makes sense to return indices starting from 1. Figure 1 provides several examples to illustrate the select function.

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

---

[1] This is actually the definition that appears in Wikipedia at the time of this writing.
[2] The notation .-tuple is a SSREFLECT idiom for a suffix operator. Similarly we use .+1 and .-1 for successor and predecessor.

```
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 [3] 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.

2.3.1   The succ and pred functions

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 in 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 is 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 $(N+1)^{\text{th}}$ b [15, p. 89]:

```
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].
```

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

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

3   LOUDS formalization

Operationally, a LOUDS encoding consists in turning a tree into an array of bits via a level-order traversal. Figure 2 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, making for a compact representation in memory. Moreover, read-only operations can be implemented using rank and select, which can be implemented in constant-time.

We explain how we formalize the LOUDS encoding in Sect. 3.1 and how we formally verify the correctness of operations on trees built out of rank and select in Sect. 3.2.

3.1   LOUDS encoding formalized in Coq

We define arbitrarily-branching trees by an inductive type:

```
Variable A : Type.
Inductive tree := Node : A -> seq tree -> tree.
Definition forest := seq tree.
Definition children_of_node : tree -> forest := ...
Definition children_of_forest : forest -> forest := flatten \o map children_of_node.
```

(a) A sample tree

(b) Its LOUDS encoding
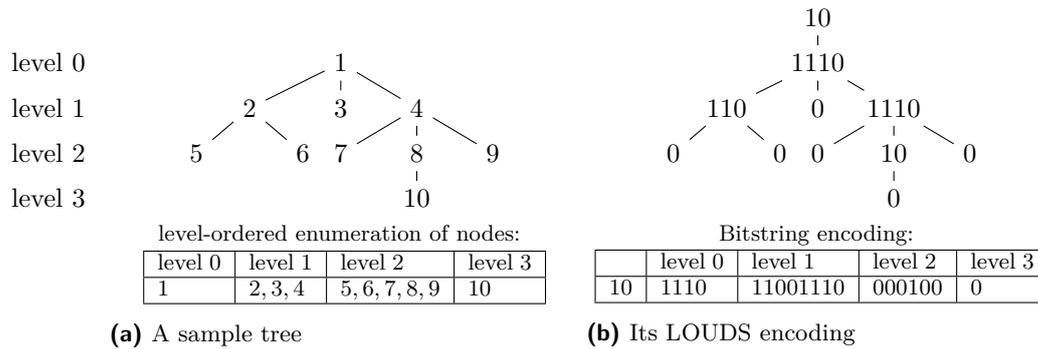
Figure 2 LOUDS encoding of a sample unlabeled tree

where `\o` is the function composition operator (i.e., ∘), and where the type `A` is the type of labels. We also introduce the abbreviation `forest` for a list of trees, and functions to obtain children. With this definition of trees, a leaf is a node with an empty list of children. For example, the tree of Fig. 2 becomes in CoQ:

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

3.1.1 Height-recursive level-order traversal

The intuitive definition of level-order traversal iterates on a forest, returning first the toplevel nodes of the forest, then their children (applying `children_of_forest`), etc. We parameterize the definition with an arbitrary function `f` for generality.

```
Variables (A B : Type) (f : tree A -> B).
Fixpoint lo_traversal' n (s : forest A) :=
  if n is n'.+1 then map f s ++ lo_traversal' n' (children_of_forest s) else [::].
Definition lo_traversal t := lo_traversal' (height t) [:: t].
```

The parameter `n` is filled here with the maximum height of the forest, meaning that we iterate just the right number of times for the forest to become empty.

Yet, this definition is not fully satisfactory. One reason is that it is not structural: we are not recursing on a tree, but iterating on a forest, using its height as recursion index. Another one is that, as we will see in Sect. 3.2, the name *level-order* is misleading. For many proofs, we are not interested in complete traversal of the tree, level by level, but rather by partial traversal along a path in the tree, where the forest we consider actually overlaps levels.

3.1.2 A structural level-order traversal

At first it may seem that the non-structurality is inherent to level-order traversal. There is no clear way to build the sequence corresponding to the traversal of a tree from those of its children. However, Gibbons and Jones [7, 10] showed that this can be achieved by splitting the output into a list of levels. One can combine two such structured traversals by *zipping* them, i.e., concatenating corresponding levels, and recover the usual traversal by

flattening the list. Since concatenation of lists forms a monoid, zipping of traversals also forms a monoid.

```
Variable (A : Type) (e : A) (M : Monoid.law e).
Fixpoint mzip (l r : seq A) : seq A := match l, r with
  | (l1::ls), (r1::rs) => (M l1 r1) :: mzip ls rs
  | nil, s | s, nil    => s
  end.

Lemma mzipA : associative mzip.
Lemma mzip1s s : mzip [::] s = s.
Lemma mzips1 s : mzip s [::] = s.
Canonical mzip_monoid := Monoid.Law mzipA mzip1s mzips1.
```

Here `Monoid.Law`, from the `bigop` module of SSREFLECT, denotes an operator together with its neutral element (here `[::]`) and the required monoidal equations, which are also satisfied by `mzip`.

We now define our traversal by instantiating `mzip` to the concatenation monoid. The resulting `mzip_cat` is a structure of type `Monoid.law [::]` that can be used as an operator of type `seq (seq B) -> seq (seq B) -> seq (seq B)` enjoying the properties of a monoid.

```
Variables (A : eqType) (B : Type) (f : tree A -> B).
Definition mzip_cat := mzip_monoid (cat_monoid B).

Fixpoint level_traversal t :=
  [:: f t] :: foldr (mzip_cat \o level_traversal) nil (children_of_node t).

Lemma level_traversalE t :
  level_traversal t =
  [:: f t] :: \big[mzip_cat/nil]_(i <- children_of_node t) level_traversal i.

Definition lo_traversal_st t := flatten (level_traversal t).
Theorem lo_traversal_stE t : lo_traversal_st t = lo_traversal f t.
```

To let COQ recognize the structural recursion, we have to use the recursor `foldr` in the definition of `level_traversal`. Yet, the intended equation is the one expressed by `level_traversalE`, i.e., first output the image of the node, and then combine the traversals of the children. Then `lo_traversal_st` can be proved equal to the previously defined `lo_traversal`. Deforestation can furthermore improve the efficiency of `level_traversal`[3].

3.1.3   LOUDS encoding

Finally, the LOUDS encoding is obtained by instantiating `lo_traversal_st` with an appropriate function (called the *node description* of a node), and flattening once more:

```
Definition node_description s := rcons (nseq (size s) true) false.
Definition children_description t := node_description (children_of_node t).
Definition LOUDS t := flatten (lo_traversal_st children_description t).
```

Here, `rcons s x` adds `x` to the end of the sequence `s`, while `nseq n x` creates a sequence consisting of `n` copies of `x`. Note that we chose here not to add the usual "10" prefix [15, p. 212] shown in Fig. 2, as it appeared to just complicate definitions. It can be easily recovered by adding an extra root node, as "10" is the representation of a node with 1 child.

---

[3] See `level_traversal_cat` in [3, `tree_traversal.v`].
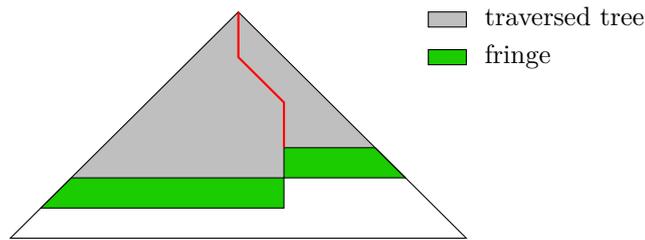
Figure 3 Level-order traversal of a tree up to a path

For example, we can recover the encoding displayed in Fig. 2 with this definition of `LOUDS`:

```
Lemma LOUDS_t : LOUDS (Node 0 [:: t]) =
  [:: true; false; true; true; true; false;
      true; true; false; false; true; true; true; false;
      false; false; false; true; false; false; false].
```

We can also prove some properties of this representation, such as its size:

```
Lemma size_LOUDS t : size (LOUDS t) = 2 * number_of_nodes t - 1.
```

This is an easy induction, remarking that `size \o flatten \o flatten` is a morphism between `mzip_cat` and `+`.

3.2   LOUDS functions using rank and select

In this section, we formalize LOUDS functions and prove their correctness. These functions are essentially built out of `rank` and `select`. Their correctness statements 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, we represent the position of a node as usual: 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. 2a is `[:: 2; 1]`. Not all positions are valid; we sort out the valid ones by means of the predicate `valid_position` (definition omitted for brevity).

In contrast, the position of nodes in the LOUDS encoding is not immediate. We define it as the length of the generated LOUDS up to the corresponding path. To do that, we first need to define a notion of level-order traversal up to a path, which collects all the nodes preceding the one referred by that path (which need not be valid):

```
Definition split {T} n (s : seq T) := (take n s, drop n s).
Variables (A : eqType) (B : Type) (f : tree A -> B).
Fixpoint lo_traversal_lt (s : forest A) (p : seq nat) : seq B := match p, s with
  | nil, _ | _, nil => nil
  | n :: p', t :: s' =>
    let (fs, ls) := split n (children_of_node t) in
    map f (s ++ fs) ++ lo_traversal_lt (ls ++ children_of_forest (s' ++ fs)) p'
  end.
```

This new traversal appears to be the key to clean proofs of LOUDS properties. In a previous attempt using the height-recursive level-order traversal of Sect. 3.1.1, proofs were unwieldy (one needed to manually set up inductions) and lemmas did not arise naturally. We expect this new traversal to have applications to other uses of level-order traversal.

This definition may seem scary, but it closely corresponds to the imperative version of level-order traversal, which relies on a queue: to get the next node, take it from the front of the queue, and add its children to the back of the queue. We define our traversal so that the node we have reached is the one at the front of the queue `s`. To move to its nth child (indices starting from 0), we first output all the nodes in the queue, and its children up to the previous one, and proceed with a new queue containing the remaining children (starting from the nth) and the children of the other nodes we have just output. Figure 3 shows how the traversal progresses. The point is that as soon as the queue spans all the fringe of the traversed tree, it is able to generate the remainder of the traversal. We can verify that `lo_traversal_lt` indeed qualifies as a level-order traversal by proving that its output converges to the full level-order traversal when the length of `p` reaches the height of the tree:

```
Theorem lo_traversal_ltE (t : tree A) (p : seq nat) :
  size p >= height t -> lo_traversal_lt [:: t] p = lo_traversal_st f t.
```

We also introduce a function that computes the fringe of the traversal up to `p`, i.e., the forest generating the remainder of the traversal.

```
Fixpoint lo_fringe (s : forest A) (p : seq nat) : forest A := ...
Lemma lo_traversal_lt_cat s p1 p2 :
  lo_traversal_lt s (p1 ++ p2) =
  lo_traversal_lt s p1 ++ lo_traversal_lt (lo_fringe s p1) p2.
```

We omit the definition but the lemma states exactly this property. It decomposes the traversal generated by a path, allowing induction from either end of the list representing the position.

Using the path-indexed traversal function, we can directly obtain the index of a node in the level-order traversal of a tree:

```
Definition lo_index (s : forest A) (p : seq nat) := size (lo_traversal_lt id s p).
```

The expression `lo_index [:: t] p` counts the number of nodes in the traversal of `t` before the position `p`. Similarly, we give an alternative definition of the LOUDS encoding, and use it to map a position in the tree to a position in its encoding (i.e., the index of the first bit of the representation of a node):

```
Definition LOUDS_lt s p := flatten (lo_traversal_lt children_description s p).
Definition LOUDS_position s p := size (LOUDS_lt s p).
```

Here the position in the whole tree is obtained as `LOUDS_position [:: t] p`, but we can also compute relative positions by using `LOUDS_position s p` where `s` is a generating forest whose front node is the one we start from. Note that both `lo_index` and `LOUDS_position` return indices starting from 0.

For example, in Fig. 2, 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 [:: Node 0 [:: t]] (0 :: p8). (* 17 *)
```

Finally, here is one of the essential lemmas for proofs on LOUDS, which relates `lo_index` and `LOUDS_position` using `select`:

```
Lemma LOUDS_position_select s p p' : valid_position (head dummy s) p ->
  LOUDS_position s p = select false (lo_index s p) (LOUDS_lt s (p ++ p')).
```

Namely if the index of `p` is n, then its position in the LOUDS encoding is the index of its nth 0-bit (recall that `select` counts indices starting from 1). Here `p'` allows us to complete `p` to a path of sufficient length, so that `LOUDS_lt` converges to `LOUDS`.

3.2.2   Number of children using succ

As a first example, let use formalize 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.

```
Fixpoint subtree (t : tree) (p : seq nat) :=
  if p is n :: p' then subtree (nth t (children_of_node t) n) p' else t.
Definition children t p := 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 each node is terminated by a 0-bit. Given such a 0-bit (or equivalently the corresponding node), one can find the number of children by computing the distance with the next 0-bit [15, p. 214]. Finding this bit is the purpose of the succ function of Sect. 2.3.1:

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

The `.+1` offset comes from the fact succ computes on indices starting from 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 p' : seq nat) :
  let B := LOUDS_lt [:: t] (p ++ 0 :: p') in
  valid_position t p -> LOUDS_children B (LOUDS_position [:: t] p) = children t p.
```

3.2.3   Parent and child node using rank and select

A path in a tree defined inductively gives direct ancestry information. In particular, removing the last index denotes the parent, and adding an extra index denotes the corresponding child. It takes more ingenuity to find parent and child 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 [15, p. 215]. 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. 2.3.1). Reciprocally, one computes the ith child by using `rank true` and `select false`. This leads to the following definitions:

```
Definition LOUDS_parent (B : bitseq) (v : nat) : nat :=
  let j := select true (rank false v B) B in pred false B j.
Definition LOUDS_child (B : bitseq) (v i : nat) : nat :=
  select false (rank true (v + i) B).+1 B.
```

One can check the correctness of `LOUDS_parent` and `LOUDS_child` as follows. Consider a node reached by the path `rcons p i`. Its parent is the node reached by the path `p`, and conversely it is the ith child of this node. We can formally prove that the LOUDS position of `p` (respectively `rcons p i`) and the position computed by `LOUDS_parent` (respectively `LOUDS_child`) coincide:
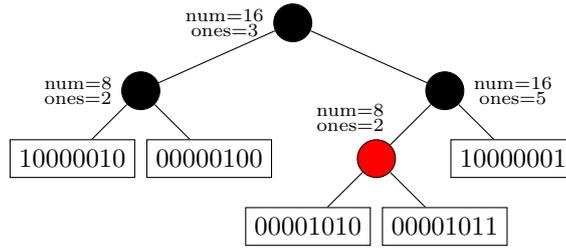
■ **Figure 4** Example of tree representation of a dynamic bit vector

```
Variables (t : tree A) (p p' : seq nat) (i : nat).
Hypothesis HV : valid_position t (rcons p i).
Let B := LOUDS_lt [:: t] (rcons p i ++ p').
Theorem LOUDS_parentE :
  LOUDS_parent B (LOUDS_position [:: t] (rcons p i)) = LOUDS_position [:: t] p.
Theorem LOUDS_childE :
  LOUDS_child B (LOUDS_position [:: t] p) i = LOUDS_position [:: t] (rcons p i).
```

The approach that we explained so far shows how to carry out the formal verification of the LOUDS operations that are listed in [15, Table 8.1]. However, how useful they may be for many big-data applications, these operations assume static compact data structures. The next section explains how to extend our approach to deal with dynamic structures.

4   Dynamic bit vectors

In some applications bit vectors need to support dynamic operations—not just static queries. We formalize such *dynamic bit vectors*, and implement and verify "dynamic operations" on them: inserting a bit into a bit vector, and deleting a bit from one.

In Sect. 4.1, we explain the data structure that allows for an efficient implementation of dynamic operations. In Sect. 4.2, we formalize the rank and select queries. Sections 4.3 and 4.4 are dedicated to the formalization of the more difficult insertion and deletion.

4.1   Representing dynamic bit vectors

The choice of representation for dynamic bit vectors is motivated by complexity considerations. 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 enables us to handle insertions in at most $O(w)$ 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 and should no more than the size of a native machine word in bits[4] [15]: i.e., for a typical 64-bit machine, we would set w to 32 or 64.

On a side note, balanced binary trees are certainly not the most compact data structure that could be used here. In fact, various data structures with better complexity have been designed [16, 20], however those structures are complicated and are unlikely to offer practical improvements over the structure presented here [15]. As a result, we choose to work only with balanced binary trees, which are much easier to reason about.

---

[4] The complexity bounds referred to in this section are dependent on the model of computation used. Here, we assume that we are working with a sequential RAM machine, where we have $O(w) = O(\log n)$ as we can only address at most 2^w bits of memory.

In our formalization of the dynamic bit vector's algorithms, we use a red-black tree as our balanced tree structure. Each node holds a color and meta-data about the bit vector, and each leaf holds a *flat* (i.e., list-based) bit array. Following Navarro [15], we store two natural numbers in each node: the size and the rank of the left subtree (recorded as "num" and "ones" in Fig. 4).

```
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).
```

Our first step is to formalize the structural invariant of our tree representation of bit vectors, which is required to prove the correctness of queries and updates on it. It states that the numbers encoded in each node are the left child's size and rank, and that leaves contain a number of bits between `low` and `high`.

```
Variables low high : nat.  (* instantiated as w^2 / 2 and w^2 * 2 *)
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                => low <= size arr < high
  end.
```

Here, the function `dflatten` defines the semantics of our tree representation of a bit vector (`dtree`) by converting it to a flat representation of that vector:

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

4.2 Verifying basic queries

The basic query operations can be easily defined via traversal of the tree. We implement the queries rank, select$_1$, and select$_0$ as the COQ functions `drank`, `dselect_1`, and `dselect_0`. For example, `drank` is implemented as follows, using the (static) `rank` function from Sect. 2.1:

```
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.
```

We prove that our function `drank` indeed computes the query rank using a custom induction principle `dtree_ind`, corresponding to the predicate `wf_dtree`:

```
Lemma drankE (B : dtree) i : wf_dtree B -> drank B i = rank true i (dflatten B).
Proof. move=> wf; move: B wf i. apply: dtree_ind. (* ... *) Qed.
```

Note that our implementation is only correct on well-formed trees.

The formalization and verification of the select queries proceed along the same lines.

4.3   Implementing and verifying insertion

Insertion is significantly harder to implement than static queries. We need to maintain the invariant on the size of the leaves, which means that we have to split a leaf if it becomes too big, and in that case we may need to rebalance the tree, to maintain the red-black invariant, updating the meta-data on the way.

We translate the algorithm given by Navarro [15] directly into Coq. Here, `high` is the maximum number of bits a leaf can contain before it needs to be split up:

```
Definition dins_leaf s b i :=
  let s' := insert1 s b i in (* insert element b in sequence s at position i *)
  if size s + 1 == high then
    let n := size s' %/ 2 in let sl := take n s' in let sr := drop n s' in
    Bnode Red (Bleaf _ sl) (n, count_mem true sl) (Bleaf _ sr)
  else Bleaf _ s'.

Fixpoint dins (B : dtree) b i : dtree := match B with
  | Bleaf s => dins_leaf s b i
  | Bnode c l d r =>
      if i < d.1 then balanceL c (dins l b i) r (d.1.+1, d.2 + b)
                 else balanceR c l (dins r b (i - d.1)) d
  end.

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

`dins` recurses on the tree, searching for the leaf where the insertion must be done, calling then `dins_leaf`, which inserts a bit in the leaf, eventually splitting it if required. On its way back, `dins` calls balancing functions `balanceL` and `balanceR` to maintain the red-black invariant. We omit the code of the balancing functions (see [3]). Like the standard version, they fix imbalances possibly occurring on the left and on the right, respectively, but they must also adjust the meta-data in the nodes. `dinsert` is a simple wrapper over `dins` that completes the insertion by painting the root black. The real definitions are more abstract [3]; we chose to instantiate them in this paper for readability.

Verifying `dinsert` requires verifying three different properties: `dinsert` must (a) preserve the data, (b) maintain the structural invariants of the tree, and (c) return a balanced red-black tree. Properties (a) and (b) are related, in that the latter is required by the former.

```
Notation wf_dtree_l := (wf_dtree low high).
Definition wf_dtree' t := if t is Bleaf s then size s < high else wf_dtree_l t.
Lemma wf_dtree_dtree' t : wf_dtree_l t -> wf_dtree' t.
Lemma wf_dtree'_dtree t : wf_dtree' t -> wf_dtree 0 high t.

Lemma dinsertE (B : dtree) b i :
  wf_dtree' B -> dflatten (dinsert B b i) = insert1 (dflatten B) b i.
Lemma dinsert_wf (B : dtree) b i : wf_dtree' B -> wf_dtree' (dinsert B b i).
```

A subtle point here is that we may start from a tree formed of a single small leaf, i.e., a leaf smaller than `low`. To handle this situation we introduce `wf_dtree'`, which does not enforce the lower bound on this single leaf. This new predicate is entailed by the original invariant (it removes one check), but interestingly it also entails it if we set the lower bound to 0. Since
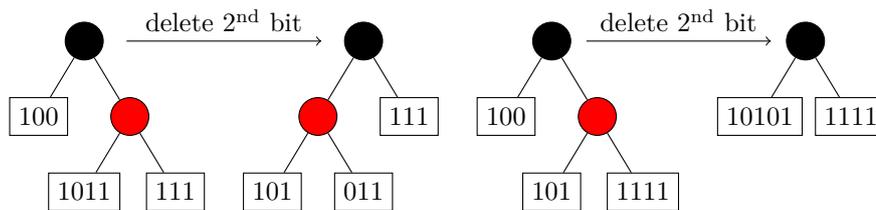
Figure 5 Base cases of delete (lower bound = 3)

the queries of Sect. 4.2 were proved with abstract lower and upper bounds, their proofs are readily usable through this weakening. However, we need to use `wf_dtree'` when we prove properties of `dinsert`, as it modifies the tree.

Proving (a) and (b) involves no theoretical difficulty. We explain in Sect. 5 some techniques to write short proofs: about 100 lines in total for both properties, including lemmas for `balanceL` and `balanceR`, which involve large case analyses.

Property (c) about `dinsert` never breaking the red-black tree invariant is notoriously more challenging. 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 that takes as arguments the "color context" `ctxt` (the color of the parent's node) and the black-depth of the node `bh`:

```
Fixpoint is_redblack (B : dtree) (ctxt : color) (bh : nat) := 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.
```

To show that `dinsert` preserves the red-black tree property, we define and prove a number of weaker structural lemmas that are basically equivalent to stating that a tree returned by `dins` is structurally valid if the root is painted black. We do not describe the proof in detail because the technique is well-known [18] and has been formalized in multiple sources (see Sect. 6). Using these weaker lemmas, we can prove the following structural validity lemma:

```
Lemma dinsert_is_redblack (B : dtree) b i n :
  is_redblack B Red n -> exists n', is_redblack (dinsert B b i) Red n'.
```

4.4 Deletion: searching for invariants

Deletion in dynamic bit vectors is difficult for two reasons. One is that, in order to maintain the upper and lower bounds on the size of leaves, which is required to attain simultaneously space and time efficiency, deleting a bit in a leaf may require some rearrangement of the surrounding nodes. Figure 5 shows the result of deleting a bit in a leaf of the tree, when this leaf has already the smallest allowed size. This can be resolved by borrowing a bit from a sibling (left case), or merging two siblings (right case), but depending on the configurations of nodes, this may require to first rotate the tree.

The other is that deletion in a functional red-black tree is a complex operation [11], and that finding how to adapt the invariants of the litterature to our specific case proved to be

non-trivial. Therefore, we took a twofold approach. First, we searched for invariants in a concrete tree structure with invariants encoded using dependent types. Then, we removed dependent types and implemented delete and proved its correctness (more details in Sect. 5).

Contrary to insertion, knowing the color of the modified child is not sufficient to rebalance its parent correctly after deletion, and recompute its meta-data. We need to propagate two more pieces of information: whether the black-height decreased (d_down below), and the meta-data corresponding to the deleted bit (d_del). We encapsulate these in a "tree state":

```
Record deleted_dtree: Type := MkD { d_tree :> dtree; d_down: bool; d_del: nat*nat }.
```

Note that deleted_dtree is automatically coerced to dtree.

Now, we can define delete in the natural way, but we need to take care about balance operations and invariants on the size of leaves. Specifically, the balance operations must be reimplemented as balanceL' and balanceR', which need to satisfy the following invariants, i.e., the resulting "balanced" tree is *deleted-red-black* (i.e., a red-black tree, either with the same black height, or with a black root and decreased black height), given that the unproblematic subtree is red-black, while the unbalanced one is deleted-red-black.

```
Definition balanceL' (c:color)(l:deleted_dtree)(d:nat*nat)(r:dtree):deleted_dtree :=
Definition balanceR' (c:color)(l:dtree)(d:nat*nat)(r:deleted_dtree):deleted_dtree :=

Definition is_deleted_redblack tr (c : color) (bh : nat) :=
  if d_down tr then is_redblack tr Red bh.-1 else is_redblack tr c bh.

Lemma balanceL'_Black_deleted_is_redblack l r n c :
  0 < n -> is_deleted_redblack l Black n.-1 -> is_redblack r Black n.-1 ->
  is_deleted_redblack (balanceL' Black l r) c n.
Lemma balanceL'_Red_deleted_is_redblack l r n :
  is_deleted_redblack l Red n -> is_redblack r Red n ->
  is_deleted_redblack (balanceL' Red l r) Black n.
(* similar statements with respect to balanceR' *)
```

Regarding leaves, we need special processing in the base cases of delete, as illustrated in Fig. 5. delete might have to "borrow" a bit from a sibling of a target leaf or combine target siblings (possibly after a rotation), to preserve the size invariants. Afterwards, delete will recursively rebalance the whole dtree.

Thus we implement delete (as ddel), and prove its correctness as follows:

```
Fixpoint ddel (B : dtree) (i : nat) : deleted_dtree := ...

Lemma ddeleteE B i : wf_dtree' B -> dflatten (ddel B i) = delete (dflatten B) i.
Lemma ddelete_wf (B : dtree) n i :
  is_redblack B Black n -> i < dsize B -> wf_dtree' B -> wf_dtree' (ddel B i).
Lemma ddelete_is_redblack B i n :
  is_redblack B Red n -> exists n', is_redblack (ddel B i) Red n'.
```

These statements are variants of the properties (a), (b) and (c) of Sect. 4.3. The proofs are complicated by the huge number of cases, handled using the proof techniques discussed in the next section.

5   Using small-scale reflection with inductive data

The small-scale reflection approach is known to be beneficial for mathematical proofs [14]. However, while SSREFLECT tactics are now widely used in the COQ community, it is not

always clear how to write proofs of programs using inductive data structures in an idiomatic style, in particular in presence of deep case analysis.

In the first part of the paper, concerning level-order traversal, the question is not so acute, as the induction principle we need for LOUDS is not structural on the shape of trees, but rather on paths, represented as lists, which are already well supported by the SSREFLECT library. Thus the question was the more traditional one of which definitions to use, so that we can obtain natural lemmas. This proved to be a time consuming process, which led to gradually build a library of lemmas, resulting in proofs that match the intuition, using almost only case analysis and rewriting.

However, the second part, about dynamic bit vectors, uses heavily structural induction on binary trees, and required developing some proof techniques to streamline the proofs.

A basic idea of small-scale reflection is to use recursive Boolean predicates (i.e., recursive computable functions) rather than inductive propositions. We have already presented two examples: `wf_dtree` and `is_redblack`. Properly designed, they allow one to prune case analysis by reducing to `false` on impossible cases. On the other hand, they do not decompose naturally in inductive proofs, which led us first to apply a standard technique: define a specialized induction principle for trees satisfying `wf_dtree` (`dtree_ind` in Sect. 4.2). Using it, the correctness of static queries and non-structural modification operations (i.e., setting and clearing of bits) were easy to prove, as the case analysis was trivial.

Properties of `dinsert`, `ddel`, and their auxiliary functions are trickier to prove, as they require complex case analyses and delicate re-balancing of branches. Nevertheless, we essentially applied the same principle of solving goals through direct case analysis. With this approach, the correctness lemmas (which state that our operations are semantically correct) were largely automated, consistent with prior research [17]. The structural lemmas were harder to prove, mainly due to the sheer number of cases involved and the complexity of invariants. Our proofs proceed by first applying case analysis to the tree up to the required depth, and then decomposing all assumptions to repeatedly rewrite the goal using them until it is solved. This proof pattern is captured by the following tactic:

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

It is reminiscent of the `intuition` tactic, a generic tactic for intuitionistic logic which breaks both hypotheses and goals into pieces; here we rather rely on rewriting inside Boolean conjunctions to solve goals piecewise. For `dinsert`, this approach instantly finishes most of our proofs, especially those about red-black tree invariants; the few cases that require manual treatment being usually handled in one single `rewrite`. This is true for most auxiliary functions of `ddel` too, with one caveat: where `dinsert` has us generate a dozen cases, `ddel` requires hundreds. To cope with this, we had first to decompose the case analysis in steps, solving most cases on the way, which means losing some simplicity to speed up proof search. The proof is still mostly automatic: apply `decompose_rewrite`, and throw in relevant lemmas. When possible, it appears that using `apply` instead of `rewrite` speeds up by a factor of 2 or more, which matters when the lemma takes more than 1 minute to prove. We have only 3 such time-consuming case analyses, one for each invariant. Among the 12 lemmas involved in proving the invariants, only the inductive proof of well-formedness for `ddel` seems to show the limit of this approach, as it required specific handling for each case of the function definition.

For comparison, Table 1 provides the size of code and proof required for each `Section` of our proof script. This does not include lemmas about the list-based reference implementation. Note that we count all Boolean predicates used to model properties as proofs.

Contents (`Section` in `dynamic_redblack.v`)	Lines of code	Lines of proof
Definitions (`btree`, `dtree`)	19	18
Queries (`dtree`)	38	58
Insertion (`insert`, `dinsert`)	65	208
Set/clear a bit (`set_clear`)	25	120
Deletion (`delete`, `ddelete`)	98	215

■ **Table 1** Implementation of dynamic bit vectors (see Table 2 for the whole implementation)

File in [3]	Section in this paper
`rank_select.v`	Sections 2.1, 2.2, 2.3
`pred_succ.v`	Sect. 2.3.1
`tree_traversal.v`	Sections 3.1.1, 3.1.2
`louds.v`	Sections 3.1.3, 3.2
`dynamic_redblack.v`	Sect. 4

■ **Table 2** Formalization overview [3] (see Table 1 for the details about dynamic bit vectors)

The proofs of set and clear, which we did not describe here, may seem relatively verbose. We prove the same properties (a,b,c) as in Sect. 4.3, but the number of lines hides a disparity between proofs of (a) correctness and (c) red-blackness, which are almost immediate, as the structure of the tree is unchanged, and (b) invariants of the meta-data, for which switching a bit requires to propagate the difference back to the root, with extra local invariants.

Last, we mention our experience with alternative approaches. In parallel with our developement using small-scale reflection, we attempted to formalize dynamic bit vectors using dependent types, where all invariants are encoded in the type of the data itself. While this guarantees that we never forget an invariant, difficulties with the `Program` [23] environment led us to write some functions using tactics [3, `dynamic_dependent_tactic.v`]. As written in Sect. 4.4, this direct connection between code and proof actually helped us discover some tricky invariants. However, the resulting code does not lend itself to further analysis, hence our choice here to stick to a more conventional separation between code and proof. We did eventually succeed in re-implementing the dependently-typed version using the `Program` environment, but at the price of very verbose definitions [3, `dynamic_dependent_program.v`].

Table 2 gives an at-a-glance overview of our entire Coq development, with a list of files and their corresponding sections in this paper.

6 Related work

Coq has been used to formalize a constant-time, $o(n)$-space rank function that was furthermore extracted to efficient OCaml code [24] and C code [25]. This work focuses on the rank query for static bit arrays while our work extends the toolset for succinct data structures with more queries (select, succ, etc.) and dynamic structures.

The functions `level_traversal` and `lo_traversal_st` of Sect. 3.1.2 match functions given in squiggle notation in related work by Jones and Gibbons [10]. In this work, the `mzip` function of Sect. 3.1.2 also appears and is called "long zip with plussle". To the best of our knowledge, the function `lo_traversal_lt` is original to our work.

Larchey-Wendling and Matthes recently studied the certification and extraction of breadth-first traversals [13]. They too define `lo_traversal_st`, but then prove it equivalent to a queue

based algorithm, which they extract to efficient OCaml code. Their goal is orthogonal to ours, as for succinct data structures what matters is not the efficiency of the traversal, but the correctness of the parent/child navigation functions, which by definition require a constant number of queries.

One may use any kind of balanced binary tree to represent dynamic bit vectors [15]. There are many purely-functional balanced binary search trees, such as AVL trees [2] and weight-balanced trees [1], but purely functional red-black trees [11, 18] are most widely studied and preferred by us. As a matter of fact, they have already been formalized in Coq [4, 5, 6], Agda [19], and Isabelle [17].

We had to re-implement red-black trees due to the difference of stored contents. Above Coq formalizations are intended to represent sets, and maintain the ordering invariant. 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 (see Sect. 4.1). Still, we found many hints in related work. For example, in Sect. 4.3 about insertion, the balancing functions use Okasaki's well-known purely functional balance algorithm [18], and we formulate our invariants and propositions similarly to above Coq formalizations.

There are now many proofs of programs that use SSReflect, but we could not find much discussion trying to synthesize the new techniques put at work. Sergey et al. used SSReflect for teaching [21, 22], observing benefits for clarity and maintainability, but also giving examples of custom tactics needed to prove programs. Gonthier et al. [9] have shown how, in some cases, one can avoid relying on ad hoc tactics through an advanced technique involving overloading of lemmas. The techniques we describe in Sect. 5, while more rudimentary, are simple and efficient, yet we have not seen them described elsewhere.

7   Conclusion

We reported on an effort to formalize succinct data structures. We started with a foundational theory of the rank and select functions for counting and searching bits in immutable arrays. Using this theory, we formalized a standard compact representation of trees (LOUDS) and proved the correctness of its basic operations. Last, we formalized dynamic bit 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 while dealing with LOUDS trees: it took much care to find suitable recursive traversals and to sort out the off-by-one conditions when specifying basic operations. Similarly, the formalization of dynamic vectors could not be reduced to the matter of extending conservatively an existing formalization of balanced trees: we needed to re-implement them to accommodate specific invariants.

As for future work, we plan to enable code extraction for the functions we have been verifying, and prove their complexity, so as to complete previous work [24] and ultimately achieve a formally verified implementation of succinct data structures. We have already shown that the LOUDS representation of a tree with n nodes uses just $2n$ bits of data. For the LOUDS operations, constant time complexity is a direct consequence of their being implemented using a constant number of rank and select operations. For dynamic bit vectors, we will first need to properly define a framework for space and time complexity.

───── **References** ─────────────────────────────────

1   Stephen Adams. Functional pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, Oct 1993.

2   G. M. Adel'son-Vel'skiǐ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

3   Reynald Affeldt, Jacques Garrigue, Xuanrui Qi, and Kazunari Tanaka. A Coq formalization of succinct data structures. `https://github.com/affeldt-aist/succinct`, 2018.

4   Andrew W. Appel. Efficient verified red-black trees. Available at `http://www.cs.princeton.edu/~appel/papers/redblack.pdf` (code included in Coq standard library, file `MSetRBT.v`, with extra modifications by Pierre Letouzey), Sep 2011.

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

6   Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *Proceedings of the 13th European Symposium on Programming (ESOP 2004), Barcelona, Spain, March 29–April 2, 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2004. Source code about red-black trees available at `https://github.com/coq-contribs/fsets/SetRBT.v`.

7   Jeremy Gibbons. *Algebras for Tree Algorithms*. PhD thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.

8   Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008. Version 17 (Nov 2016).

9   Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23(4):357–401, 2013.

10   Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report 71, Department of Computer Science, University of Auckland, 1993.

11   Stefan Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11(4):425–432, Jul 2001.

12   Taku Kudo, Toshiyuki Hanaoka, Jun Mukai, Yusuke Tabata, and Hiroyuki Komatsu. Efficient dictionary and language model compression for input method editors. In *Proceedings of the Workshop on Advances in Text Input Methods (WTIM 2011)*, pages 19–25, 2011.

13   Dominique Larchey-Wendling and Ralph Matthes. Certification of breadth-first algorithms by extraction. In *Proceedings of the 13th International Conference on Mathematics of Program Construction*, 2019.

14   Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at: `https://math-comp.github.io/mcb/`, 2016. With contributions by Yves Bertot and Georges Gonthier.

15   Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

16   Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3), 2014.

17   Tobias Nipkow. Automatic functional correctness proofs for functional search trees. In *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2016.

18   Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

19   Julien Oster. An Agda implementation of deletion in left-leaning red-black trees. Available at `https://www.reinference.net/llrb-delete-julien-oster.pdf`, Mar 2011.

20   Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 426–437. Springer, 2001.

21   Ilya Sergey. Programs and proofs: Mechanizing mathematics with dependent types (lecture notes). Available at `https://ilyasergey.net/pnp/`.

22   Ilya Sergey and Aleksandar Nanevski. Introducing functional programmers to interactive theorem proving and program verification (teaching experience report). Available at `https://ilyasergey.net/papers/teaching-ssr.pdf`, 2015.

23 Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants.* PhD thesis, Université de Paris-Sud, 2008.

24 Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. Formal verification of the rank algorithm for succinct data structures. In *18th International Conference on Formal Engineering Methods (ICFEM 2016), Tokyo, Japan, November 14–18, 2016*, volume 10009 of *Lecture Notes in Computer Science*, pages 243–260. Springer, Nov 2016.

25 Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. Safe low-level code generation in Coq using monomorphization and monadification. *Journal of Information Processing*, 26:54–72, 2018.