# Formal Verification of the *rank* Function for Succinct Data Structures

Akira Tanaka[1] Reynald Affeldt[1] Jacques Garrigue[2]

[1] National Institute of Advanced Industrial Science and Technology

[2] Nagoya University

**Abstract** Succinct data structures are designed to use a minimal amount of computer memory. They are used for big data, so that the quality of big data analysis ultimately depends on the correct implementation of algorithms for succinct data structures. Yet, they are difficult to verify because they are usually written in low-level languages to achieve the best performance for bit-wise manipulations. In this paper, we report on an on-going effort to provide formal verification for succinct data structures. More precisely, we discuss the construction with the Coq proof-assistant of a verified implementation in OCaml of a standard algorithm, namely Jacobson's *rank* function. The main issue is the conflicting requirements about the data structures used for formal verification and the ones used for program execution. List-like data structures of mathematical objects are better suited to formal reasoning, whereas efficient execution requires bit-level arrays. To enjoy the best of both worlds, we propose to use code extraction from Coq to OCaml but with an original OCaml implementation of bitstrings. With this approach, one can take advantage of Coq to formalize correctness, including important claims about storage requirements, and still get OCaml code whose performance is acceptable. To the best of our knowledge, this is the first application of formal verification to succinct data structures.

## 1 Towards Formal Verification for Succinct Data Structures

[1] Succinct data structures are data structures designed to use an amount of computer memory close to the information-theoretic lower bound (see [15] for an introduction). They are used in particular to process big data. Thanks to an important amount of research, succinct data structures are now equipped with algorithms that are often as efficient as their classical counterparts. In this paper, we are concerned with the most basic one: the *rank* function, which counts the number of 1 (or 0) in the prefixes of a bitstring. This function requires $o(n)$ storage for constant-time execution where $n$ is the length of the bitstring (see Sect. 2 for background information).

Our long-term goal is to provide formal verification of algorithms for succinct data structures. In particular, we aim at the construction of a realistic library of verified algorithms. Such a library could significantly improve the confidence in software implementation of big data analysis. However, software implementations of algorithms for succinct data structures are difficult to verify. Indeed, since these data structures are designed at the bit-level and since performance is a must-have, they are usually written in low-level languages such as C++ (e.g., [13]), whose formal verification is still out of reach today.

In this paper, we show how to develop a verified implementation of an algorithm for succinct data structures using the Coq proof-assistant. Coq provides us with the ability to reason about the correctness of the algorithm: its functional correctness but also some important properties about storage requirements. We can also derive an efficient implementation thanks to the extraction facility from Coq to the OCaml language and the imperative features of the latter. The main issue when dealing with algorithms

[1] Presented at the 18th JSSST Workshop on Programming and Programming Languages (`http://logic.cs.tsukuba.ac.jp/ppl2016`)

for succinct data structures in Coq is that, since Coq is a purely functional language, arrays are better represented as lists to perform formal verification. However, lists do not enjoy constant-time random-access, making it difficult to use the extraction facility of Coq to generate efficient OCaml algorithms. As a solution, we provide an OCaml library for bitstrings with constant-time random-access that matches the interface of Coq lists so that we can use real bitstrings in the extracted code. This approach augments the trusted base but in the form of a localized, reusable library of OCaml code whose formal verification can anyway be carried out at a later stage. We think that this is a reasonable price to pay compared to the benefits of carrying out formal verification in Coq.

**Paper Outline** In this paper, we demonstrate our approach by building a verified implementation of the *rank* function using the Coq proof-assistant. More precisely, we provide formal verification for the *rank* function (formal proof of functional correctness in Sect. 5.4 and formal proof for storage requirements in Sect. 5.5) and extraction to executable OCaml code (by providing in particular a new library for bitstrings with constant-time random-access in Sect. 4.3). We will be able to check that the time-complexity of the extracted code is as expected (i.e., execution is constant-time, see Sect. 6.2). In the process, we discuss thoroughly the choices we made, in particular, the modular approach we took when formalizing the *rank* function in the Coq proof-assistant (Sect. 5).

# 2 A Formal Account of The *rank* Function

## 2.1 Formal Definition of the *rank* Function

Given a bitstring $s$ and an index $i$ in $s$, $rank_s(i)$ counts the number of 1's up to $i$ (excluded). For example, in Fig. 1 (the first and second-level directories will be explained in Sect. 2.2), $s$ contains 26 1's, $rank_s(4) = 2$, $rank_s(36) = 17$, and $rank_s(58) = 26$.

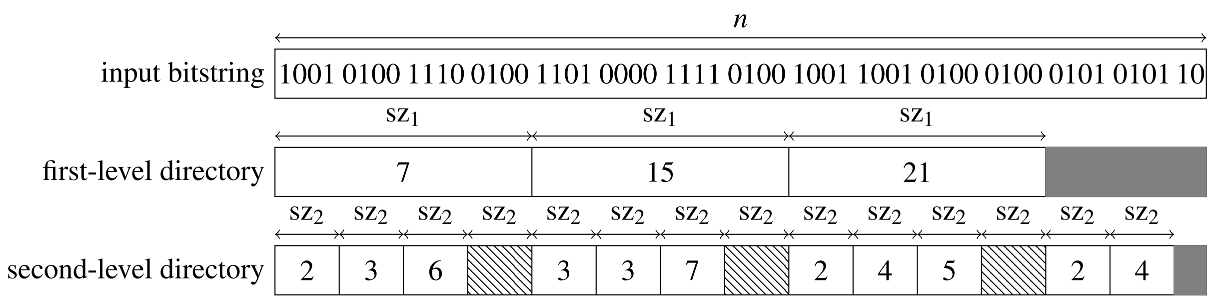


**Figure 1.** Illustration for the *rank* function ($sz_2 = 4$, $sz_1 = 4 \times sz_2$, $n = 58$). Example extended from [11].

The mathematically-inclined reader would formally specify the *rank* function as $rank_s(i) = |\{k \in [0,\ldots,i) \,|\, s[k] = b\}|$ where $b$ is the query bit (1 in the example above). Using the Mathematical Components [5] library of the Coq proof-assistant, such a specification can be formalized directly. For bits, one can use the Coq type for booleans `bool`. An input bitstring of length $n$ can be formalized as a tuple of $n$ booleans (type `n.-tuple bool`). An index $k$ is a natural number strictly less than $n$; this corresponds precisely to the type `'I_n`. Access to the $k$th element of a tuple $s$ is performed by the `tnth` function. This leads to the definitions of the following (finite) set of indices (defined by comprehension using the notation `[set x : T | P x ]`):

```
Definition Rank_set b i (s : n.-tuple bool) :=
  [set k : 'I_n | (k < i) && (tnth s k == b)].
```

The *rank* function is just the cardinal of the above set: `Definition Rank b i s := #| Rank_set b i s |`. A functional programmer would formally specify the *rank* function as list surgery and filtering. For example:

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

Both are provably equivalent but none of them provides an efficient implementation: `Rank` cannot even be executed, `rank` can be executed (both in Coq and as an extracted OCaml program) but computation would (hopefully) be linear-time.

## 2.2 Jacobson's *rank* Function

Jacobson's *rank* function [9] is a constant-time implementation of the *rank* function. It uses auxiliary data structures, in particular two arrays called the first and second-level directories that essentially contain pre-computed values of *rank* for substrings of the input bitstring $s$ of size $n$ (see Fig. 1). More precisely, each directory contains fixed-size integers, whose bit-size is large enough to represent the intended values, so that the bit-size for each directory depends on $n$.

Let $\mathrm{sz}_2$ be the size of the substrings used for the second-level directory. Hereafter, we refer to these substrings as the "small blocks". The size of the substrings used for the first-level directory is $\mathrm{sz}_1 = k \times \mathrm{sz}_2$ for some $k$. We refer to these substrings as the "big blocks". The first-level directory is precisely an array of $n/\mathrm{sz}_1$ integers such that the $i$th integer is $rank_s((i+1) \times \mathrm{sz}_1)$. The second-level directory is also an array of integers. It has $n/\mathrm{sz}_2$ entries and is such that the $i$th entry is the number of bits among the $(i\%k+1) \times \mathrm{sz}_2$ bits starting from the $((i/k) \times \mathrm{sz}_1)$th bit ($/$ is integer division and $\%$ is the remainder operation). One can observe that when $i\%k = k-1$, the $i$th entry of the second-level directory (the hatched rectangles in Fig. 1) can be computed from the first-level directory and therefore does not need to be remembered. It can be shown that these data structures require only $o(n)$ bits with integers of the appropriate size (not necessarily the word size of the underlying architecture).

Given an index $i$, Jacobson's *rank* function decomposes $i$ such that $rank_s(i)$ can be computed by adding the results of (1) one lookup into the first-level directory, (2) one lookup into the second-level directory, and (3) direct computation of *rank* for a substring shorter than $\mathrm{sz}_2$. For example, in Fig. 1, $rank_s(36) = rank_s(2 \times 16 + 1 \times 4 + 0)$ is computed as $15+2$ and $rank_s(58) = rank_s(3 \times 16 + 2 \times 4 + 2)$, as $21+4+1$. Since the computation of *rank* for a substring shorter than $\mathrm{sz}_2$ in (3) can also be tabulated or computed with a single instruction on some platforms, *rank*'s computation is constant-time.

## 2.3 Formal Definitions of the First and Second-level Directories

The first and second-level directories can be specified formally in a few lines of Coq code using standard list functions. Let us consider a bitstring `s` and a block size `sz`. The corresponding first-level directory has `size s/sz` entries and it is obtained by computing, for the `i`th block, `rank b (i * sz) s`:

```
Definition first_level_dir b sz s :=
  [seq rank b (i * sz) s | i <- iota 1 (size s %/ sz)].
```

(`iota a m` is $[a, a+1, \ldots, a+m-1]$, `[seq ... | ... <- ...]` is a notation for the map function, `%/` is integer division in Mathematical Components.)

As for the second-level directory, one can see it as the concatenation of first-level directories for the substrings $s[i \times \mathrm{sz}_1, \ldots, (i+1) \times \mathrm{sz}_1 - 1]$ where $i$ is an index in the first-level directory. We first introduce a function that computes for a bitstring `s` the consecutive blocks of size `sz1` (and ignores what is left):

```
Definition shape_dir sz1 s := reshape (nseq (size s %/ sz1) sz1) s.
```

(`nseq n a` is a lists of `n` `a`'s.) For each block computed by `shape_dir`, we remove the trailing `sz2` bits and let `firsts` be the resulting list of blocks (line 3). Let `last` be the trailing bits of the original bitstring `s` that were not part of any block (line 4). We obtain the second-level directory by computing the "first-level directories" for the blocks in the list `rcons firsts last` ("`last` put at the end of `firsts`", line 5):

```
1 Definition second_level_dir b sz2 k s :=
2   let sz1 := k * sz2 in
3   let firsts := [seq (take (sz1 - sz2) x) | x <- shape_dir sz1 s] in
4   let last := drop (size s %/ sz1 * sz1) s in
5   [seq first_level_dir b sz2 x | x <- rcons firsts last].
```

**Summary of Sect. 2** We regard the above Coq functions as a formal specification of Jacobson's algorithm. In this paper, we provide Coq functions that are more realistic in the sense that they can be extracted to executable OCaml code. Nevertheless, their correctness are proved w.r.t. to the functions explained in this section. For example, functional correctness is proved w.r.t. `rank` (see Sect. 5.4). We have also proved that these functions produce directories with the same contents as `first_level_dir` and `second_level_dir` (see [16] for details).

# 3 Our Approach: Extraction From a Generic *rank* Function

## 3.1 A Generic Rank Function in Coq

We start explaining the *rank* function that we will extract. In this section, we explain a generic version of the function; we provide a concrete instantiation in Sect. 5.3. The generic version essentially consists of two functions: one that constructs the directories and one that performs the lookup.

To simplify the presentation, we first explain a function that counts bits in a naive way[2]. `bcount b i l s` counts the number of bits `b` (0 or 1) inside the slice $[\mathtt{i},\ldots,\mathtt{i}+\mathtt{l})$ of the bitstring `s` (essentially a list of booleans—see Sect. 4.1):

```
Definition bcount b i l (s : bits) := count_mem b (take l (drop i s)).
```

In the code below, we use notations from the Mathematical Components [5] library: `.+1` is the successor function, `%/` and `%%` are the integer division and modulo operators, and `if x is xp.+1 then e1 else e2` means: if `x` is greater than 0 then return `e1` with `xp` bound to $\mathtt{x}-1$, else return `e2`.

**Construction of the Directories** The function `rank_init_iter` computes both directories in one pass (it returns a pair). It has been written with extraction in mind. In particular, it uses tail calls, and indexing instead of list pattern-matching.

`j` is a counter for small blocks (we start counting from `nn`, the total number of small blocks, i.e., $n/\mathrm{sz}_2$). `i` is a counter to count small blocks in one big block. `n1` contains the number of bits counted so far for the current big block. `n2` contains the number of bits counted so far for the current small block. `dir1` (resp. `dir2`) are abstract data types meant for the first-level (resp. second-level) directory (so that `push_dir1`, `finalize_dir1`, etc. are meant to be instantiated with concrete functions later).

The function `rank_init_iter` iterates over the number of small blocks. At each iteration, the number of bits in the current small block is stored in `m` (line 2) (`b` is the query bit, `sz2` is the size of small blocks, `input_bs` is the input bitstring). For each small block, `n2` is stored in the second-level directory (line 4). After a big block has been scanned, the number of bits counted so far for the current big block `n1 + n2` is stored in the first-level directory (line 8). The number of small blocks in one big block (`kp` plus 1) is used to control the iteration inside a big block (line 10).

Observe that the directories built by `rank_init_iter` are slightly different from the data structures explained in Sect. 2.2: they start with a 0 (stored at line 8 for the first-level directory and stored at line 9 for each group of small blocks) which is of course not necessary but this simplifies the lookup function.

```
1  Fixpoint rank_init_iter j i n1 n2 dir1 dir2 :=
2    let m := bcount b ((nn - j) * sz2) sz2 input_bs in
3    if i is ip.+1 then
4      let dir2' := push_dir2 dir2 n2 in
5      if j is jp.+1 then rank_init_iter jp ip n1 (n2 + m) dir1 dir2'
6      else (dir1, dir2')
7    else
8      let dir1' := push_dir1 dir1 (n1 + n2) in
9      let dir2' := push_dir2 dir2 0 in
10     if j is jp.+1 then rank_init_iter jp kp (n1 + n2) m dir1' dir2'
```

[2] The function `bcount` is not intended to be extracted as it is but replaced by a more efficient function. It could be tabulated as explained in Sect. 2.2, but in this paper, it will be replaced by a single gcc built-in operation (see Sect. 4.3).

```
    else (dir1', dir2').
Definition rank_init_iter0 := rank_init_iter nn 0 0 0 empty_dir1 empty_dir2.
Definition rank_init_gen :=
  let (dir1, dir2) := rank_init_iter0 in (finalize_dir1 dir1, finalize_dir2 dir2).
```

**Lookup** The function `rank_lookup_gen` is a generic implementation of the lookup function. It computes the rank for index `i`:

```
Definition rank_lookup_gen i :=
  let j2 := i %/ sz2 in (* index for the second-level directory *)
  let j3 := i %% sz2 in (* index inside a small block *)
  let j1 := j2 %/ k in  (* index for the first-level directory *)
  lookup_dir1 j1 dir1  +  lookup_dir2 j2 dir2  +  bcount b (j2 * sz2) j3 input_bs.
```

`j1` (resp. `j2`) is the index of the block in the first-level directory (resp. second-level directory). They are computed using the size of small blocks `sz2` and the ratio between the size of big and small blocks `k` (or in other words, the number of small blocks in a big block; so that the size of big blocks is `k * sz2`). `lookup_dir1` (resp. `lookup_dir2`) is meant to perform array lookup; it will be instantiated later.

### 3.2 Our Approach w.r.t. Extraction

In the code above, lookup in the directories is meant to be performed by the functions `lookup_dir1` and `lookup_dir2`. Constant-time execution for these functions is required for Jacobson's *rank* function to be efficient. If we implement these functions with `nth`-like access to standard `lists` (which is linear-time), Coq will not generate OCaml functions with the desired time complexity. At first, one may think of looking for an ingenious implementation scheme that may cause Coq to generate *in fine* efficient OCaml code. This approach seems to us too optimistic as a first step towards the goal of providing a verified library of functions for succinct data structures for the following two reasons:

- Coming up with new implementation schemes is likely to make more difficult the task of proving formally the functional correctness and the storage requirements of algorithms.
- The code extraction facility of Coq is not optimized in any way (by design, because it is part of the trusted base). In practice, it tends to generate inefficient code for convoluted formalizations. As a matter of fact, previous work shows that Coq requires significant engineering to handle imperative features and native data structures (e.g., [2]).

Instead, our approach consists in (1) making the best we can out of list-like data structures in Coq and (2) providing an efficient OCaml implementation of the list interface that we will substitute to Coq-generated functions.

## 4 An OCaml Bitstring Library for Coq Lists of Booleans

Direct extraction of Coq lists and list functions suffers two major problems w.r.t. succinct data structures: (1) memory usage is very inefficient (assuming 64-bit machine words, it would take 192 bits to represent one boolean), (2) random-access will be linear-time instead of the required constant-time complexity. We now explain an OCaml implementation for the interface of Coq's lists that solves above problems.

### 4.1 Bitstrings Implemented in Coq and their Extraction

We define bitstrings as an inductive type similar to Coq lists (with two constructors `bnil` and `bcons`):

```
Inductive bits : Type := bnil | bcons of bool & bits.
```

The type `bits` is isomorphic to the type of lists of booleans (functions `seq_of_bits` and `bits_of_seq` do the conversion, the former is registered as a coercion so that it does not need to be referred to explicitly). In consequence, many functions and lemmas for `bits` are easily derivable from Coq standard libraries:

```
Definition bsize (s : bits) := size s.
Definition bnth i (s : bits) := nth false s i.
Definition breverse (s : bits) := bits_of_seq (rev s).
Definition bappend (s1 s2 : bits) := bits_of_seq (s1 ++ s2).
Definition bcount b i l (s : bits) := count_mem b (take l (drop i s)).
```

However, code extracted from above functions does not achieve the desired complexity. The code extracted from `bsize`, `bnth`, `breverse`, and `bcount` would be linear-time because these functions scan `bits` like a list[3]. Regarding memory usage, the constructor `bcons` would allocate a memory block with one per argument (see Fig. 2, on the left, for an illustration). In addition, OCaml needs one more word for each block to manage memory. Assuming the machine word is 64 bits, `bcons` would therefore need 192 bits to represent a Coq `bool`, that was originally supposed to represent a single bit...

In the next section (Sect. 4.2), we provide OCaml datatypes and functions to replace the Coq type `bits`, its constructors `bnil` and `bcons`, pattern-matching of `bits`, and the functions `bsize`, `bnth`, etc.

**Overview of the Extraction of Coq Lists** Concretely, extraction from Coq is the matter of the command `Extraction` (see file `Extract.v` [16]). To replace inductive types and functions with custom OCaml code, we provide the hints such as:

```
1 Extract Inductive bits =>
2   "Pbits.bits" [ "Pbits.bnil" "Pbits.bcons" ] "Pbits.bmatch".
3 Extract Inlined Constant bsize => "Pbits.bsize".
4 Extract Inlined Constant bnth => "Pbits.bnth".
```

At line 1, we replace the Coq inductive type `bits` with the OCaml type `Pbits.bits` and its constructors with the constant `Pbits.bnil` and the function `Pbits.bcons` (of type `bool * Pbits.bits -> Pbits.bits`). Pattern-matching of the form

```
match bs with bnil => E1 | bcons b t => E2 end
```

is replaced with the function call

```
Pbits.bmatch (fun () -> E1) (fun (b, t) -> E2) bs
```

From line 3, the functions `bsize`, `bnth`, etc. from Sect. 4.1 are replaced by the functions `Pbits.bsize`, `Pbits.bnth` etc. to be explained in Sect. 4.3.

## 4.2 Bitstrings Implemented in OCaml

The main idea to achieve linear-time construction and constant-time random-access in OCaml is to implement bitstrings using a datatype that allows for random-access of bits. For this purpose, we use the type `bytes` introduced in OCaml 4.02.0. (Currently, `bytes` is the same as `string`; OCaml plans to change `string` to immutable.) The resulting OCaml type is as follows[4]:

```
type bits_buffer = { mutable numbits_used : int; s : bytes; }
type bits = Bdummy0 | Bdummy1 | Bref of int * int * bool * bits_buffer
```

Bitstrings are stored in a `bits_buffer` as a `bytes` together with the number of bits `numbits_used` used so far. Let us first explain the constructor for arbitrary-length bitstrings (`Bref`) and then explain how short bitstrings are implemented as immediate values (this will explain `Bdummy0` and `Bdummy1`).

**`bits` represented with `Bref`** The data structure `Bref(`*start*`,`*len*`,true,buf)` (depicted on the right of Fig. 2) represents the slice $[start, start+len)$ of the `bits_buffer` bitstring `buf`. The third argument of `Bref` specifies the ordering: when it is `false`, then the first bit is at index *start*, and when it is `true`, then the first bit is at index $start+len-1$ (the `bytes` order is least significant bit first).

[3] Let `s` be a bitstring of length $n$. `bsize s` is $O(n)$, `bnth i s` is $O(i)$, `breverse s` is $O(n)$, and `bcount b i l s` is $O(i+l)$. `bcount` requires an additional $O(i)$ because of the `drop` function (see Sect. 3.1).

[4] The OCaml definitions below belong to the module `Pbits`; the prefix `Pbits.` is omitted when no confusion is possible.


**Figure 2.** A Coq `bits` on the left and the corresponding OCaml `bits` on the right

The dynamics of `bits` represented with `Bref` is as follows. Initially, `numbits_used` is 0, which means that the bitstring is empty. When a bit is appended to the buffer, it is assigned to the `numbits_used`$^{th}$ bit in `s` and `numbits_used` is incremented. When the buffer is full (i.e., $8 \times |\texttt{s}| = $`numbits_used`), `s` is copied into a new `bytes` with a doubled length before the bit is appended.

The constructor `Bref` can represent any bitstring but it requires memory allocation for each value, even to represent an empty bitstring, a single boolean, etc. We can improve efficiency by avoiding memory allocation with immediate values. Note that there is no soundness problem in implementing bitstrings as immediate values because bitstrings `bits` are immutable in Coq.

**`bits` represented with immediate values** In summary, we use the unboxed integers of OCaml to represent short bitstrings. In OCaml, values are represented by $w$-bit integers, $w$ being the number of bits in a machine word (32 or 64). These integers represent either (1) a $(w-1)$-bit unboxed integer or (2) a pointer to a block allocated in the heap. OCaml datatypes use unboxed integers for constant constructors and pointers to blocks otherwise. Therefore, we can represent short bitstrings by unboxed integers. More precisely, we represent bitstrings of length $u \leq w-2$ as a $(w-1)$-bit integer using the following format: $\overbrace{0\dots0}^{w-u-2}1b_{u-1}\dots b_1 b_0 1$ (the trailing 1 is a tag bit to distinguish unboxed integers and pointers). To treat the latter integers as `bits` we use `Obj.magic`. The reason for adding the constructors `Bdummy0` and `Bdummy1` to the datatype `bits` is technical. Without them, OCaml optimizes pattern-matching (with `match`) by assuming there is no or only one constant constructor in `bits`; this causes a segmentation fault error because an unboxed integer is wrongly considered as a pointer.

## 4.3 OCaml Functions for Bitstrings

We now equip the OCaml type `bits` with the same functions as the interface of the Coq type `bits`, but so as to achieve the time-complexities required by Jacobson's *rank* function. Indeed, most OCaml functions that we propose as a replacement achieve the same tasks in constant-time instead of linear-time.

- `bsize` runs in constant-time because it just returns the second parameter of `Bref`.
- We implement `bnth` in constant-time easily by using OCaml functions for random-access to `bytes` (`Bytes.get`, `Bytes.set`).
- `breverse` runs in constant-time by returning a new `bits` where the third parameter of `Bref` is negated. `breverse` is important to implement functions which use tail calls to avoid stack overflow.
- `bappend s s'` runs in $O(\min(n,n'))$-time for array construction ($n$, $n'$ are the lengths of `s`, `s'`) by copying the shortest argument among `s` and `s'`. This may need reversing of the longest one, but, as far as we are concerned, this only happens at the beginning of array construction.
- `bcons` works in constant-time for array construction. More precisely, `bcons` works in constant-time if it is possible to append a bit in the `bits_buffer` (i.e., when the third argument of `Bref` is true and `start` + `len` = `numbits_used`). In this case, `bcons` appends the bit into the `bits_buffer` by a destructive update and returns a newly allocated `bits` which refers to a sub-range including the appended bit. (If the buffer is full, it is doubled but this still needs only constant-time with amortization.) If the destructive update is not possible, `bcons` copies the referenced sub-range of `bits_buffer` into a newly allocated `bits_buffer`. This copy needs linear-time and space w.r.t. the length of the sub-range. However, as far as the initialization phase of Jacobson's *rank* algorithm is concerned, the two arrays are constructed from left to right, so that `bcons` always runs in amor-

tized constant-time. We implement `bcons` using `bappend` in order to reduce the number of `bits` allocations, compared to iterative invocations of `bcons`. (Pushing a $w$-bit integer in an array using `bcons` would require $w$ invocations, each allocating one `bits`. Yet, all allocated `bits` but the last one would immediately be discarded.)

- `bcount b i l s` runs in $O(l)$-time in general (the Coq `bcount` requires an additional $O(i)$ because of the `drop` function, whereas in OCaml access to the `i`th bit is direct). In fact, we have implemented `bcount` to use specialized assembly instructions when possible. Concretely, `bcount` is implemented in C to use gcc's `__builtin_popcountl` [4], which counts the number of bits set in a `long` value. For example, gcc generates POPCNT instructions for Intel SSE4.2 [8], so that we can assume that `__builtin_popcountl` works in constant-time.
- `bitlen n` returns $\lceil \log_2(n+1) \rceil$. This function is implemented in C using gcc's `__builtin_clzl` [4], which counts the number of leading zero bits in a `long` value. gcc generates LZCNT instructions (introduced with Intel AVX2 [6]). `bits` functions uses `bitlen` to calculate the length of `bits` represented as an immediate value. (This function is also used in `rank_default_param` in Sect. 5.5.)

### 4.4 Manipulation of Fixed-size Integers

At the Coq level, our implementation of the *rank* algorithm also uses functions that are more coarse-grained than the functions we explained just above. They implement bitstrings manipulations performed on natural numbers. (See Sect. 5.2 for how these functions are used.)

- `bword u n` builds a short bitstring from the lower `u` bits of a natural number `n` in constant-time. In OCaml, a natural number is formatted as $b_{w-2} \dots b_1 b_0 1$, where $w$ is the number of bits in a machine word (the trailing 1 is a tag bit). In order to construct short bitstrings as immediate values following the format explained in Sect. 4.2, we use simple bit operations: mask and set the topmost bit. (If $u = w-1$, one `Bref` is allocated but Jacobson's *rank* algorithm uses smaller sizes.)
- `getword i w` looks for the $w$ bits (ordered with least significant bit first) starting from index $i$, regarding them as a natural number. In OCaml, this function is implemented by accessing data at the level of bytes (not bits) to reduce the overhead (number of bit operations and number of loops).
- `wbitrev w n` returns the lower `w` bits of `n` (of type `nat`) in reversed order. This function is implemented in OCaml using a table for bit reversal; it has 256 entries to reverse a byte at once, so that `wbitrev` reverses its argument with 4 (resp. 8) table lookups on a 32 (resp. 64) bits architecture.

## 5 A Modular Formalization of the *rank* Function

In this section, we instantiate the generic *rank* function from Sect. 3.1 by implementing in Coq the first and second-level directories and providing adequate `lookup_dir`, `push_dir`, etc. functions. For this purpose, we provide an interface for arrays (Sect. 5.1) and a concrete instance with bitstrings (Sect. 5.2) that we use to obtain a concrete implementation of Jacobson's *rank* function (Sect. 5.3). Finally, we prove that this function indeed computes *rank* (Sect. 5.4) and fulfills storage requirements (Sect. 5.5).

### 5.1 An Interface for Arrays of Integers

We first define an interface for arrays of integers (precisely, fixed-size integers, implemented using lists of bits) to be used as an abstraction for the first and second-level directories of the *rank* function. Abstraction is achieved via the section mechanism of Coq (commands `Section`, `End` and `Variable`, `Hypothesis`). We do not use functors because they prevent inlining beyond module boundary.

The array interface is parameterized by a natural number `n` that represents the number of bits in a fixed-size integer (precisely, the number of bits minus one, so that we avoid dealing with the uninteresting corner case of empty sequence of integers of size 0). Indeed, Jacobson's *rank* algorithm uses two arrays of fixed-size integers (first and second-level directories) whose bit-size are different in general. Here follows the first part of the array interface:

```
Variable V : nat -> Set.
Variable Arr0 : nat -> Set.
Variable Arr : nat -> Set.
Variable empty : forall n, Arr0 n.
Variable push : forall n, Arr0 n -> V n -> Arr0 n.
Variable finalize : forall n, Arr0 n -> Arr n.
Variable lookup : forall n, nat -> Arr n -> V n.
```

The type `V` represents the values stored in the array (parameterized by a size). We distinguish between arrays in construction (type `Arr0`) and lookup (type `Arr`) phases. Construction starts from an `empty` array. Values are added by `push`. The array transit from construction to lookup phase by `finalize`. The array values can be indexed (starting form 0) using `lookup`.

The code above does not say anything about the conditions under which the functions `push`, `lookup`, etc. work as expected. For example, we can choose `V n` to represent integers of size `n` and implement them using the type `nat`. But then, we need to know when a natural number `nat` can indeed be represented as a integer of size `n`. Otherwise, it may well happen that one cannot retrieve a `push`ed value using `lookup`. Therefore, we extend the above code with the expected properties for the functions `push`, `lookup`, etc. For this properties to hold for concrete implementations, we sort out valid values from the others using the following predicate: `Variable ValidV : forall n, V n -> Prop`. For example, as long as values are `ValidV`, we know that `lookup` retrieves `push`ed values. The properties we are referring to are unsurprising but because of the abstraction approach we have chosen their formalization is a bit involved, so that we refer the reader to [16] for details.

## 5.2 Arrays of Integers Instantiated with Bits

We now implement the array interface of Sect. 5.1 for the first-level directory (the implementation for the second-level directory is similar). The word array is implemented using the type `bits`:

```
Definition Dir1Arr0 := bits.
Definition Dir1Arr := bits.
Definition empty_dir1 := bnil.
```

Word array functions are implemented using the functions explained in Sect. 4.3 and 4.4. In the following, we think of the fixed-size integers as ordered with least significant bit first, and similarly for the natural numbers (they are seen as lists of bits). Let `w1` be the number of bits in a fixed-size integer.

The function `push_dir1` is implemented using the functions `wbitrev` and `wcons`. `wbitrev w1 n` returns a natural number formed by the lower `w1` bits of `n` in reversed order. `wcons w1 m s` prepends the lower `w1` bits of `m` to `s` (using bappend and bword):

```
Definition push_dir1 w1 s n := wcons w1 (wbitrev w1 n) s.
```

The function `finalize_dir1` is just a list reversal. It cancels the reversal caused by `wbitrev`, so that the bits in a fixed-size integer are still ordered with least significant bit first:

```
Definition finalize_dir1 s := breverse s.
```

`lookup_dir1 w1 i s` looks for the `i`th fixed-size integer (as a `nat`) in `s`. It is implemented by `wnth` (itself implemented using `getword`):

```
Definition lookup_dir1 w1 i s := wnth w1 i s.
```

## 5.3 An Extractable Instance of the *rank* Algorithm

We instantiate the functions from Sect. 3.1 (`rank_lookup_gen` and `rank_init_gen`) with the array of bits from Sect. 5.2. Beforehand, we introduce two datatypes: `Record Param` carries the parameters of Jacobson's algorithm, `Record Aux` essentially carries the results of the execution of the initialization phase:

```
1 Record Param : Set := mkParam {
2  kp_of : nat ; (* sz1 / sz2 - 1 *)
3  sz2p_of : nat ; (* sz2 - 1 *)
4  nn_of : nat ; (* nb. of small blocks *)
5  w1_of : nat ;
6  w2_of : nat }.
```

```
7 Record Aux : Set := mkAux {
8  query_bit: bool;
9  input_bits: bits;
10 parameter: Param;
11 directories: Dir1Arr * Dir2Arr }.
```

Jacobson's algorithm is parameterized by the number of small blocks (minus 1) in a big block (line 2), the number of bits (minus 1) in a small block (line 3), the number of small blocks (line 4), and the bit-size of fixed-size integers for each directory (lines 5–6). The instantiation of `rank_init_gen` returns the query bit (line 8), the input bitstring (line 9), the parameters of Jacobson's algorithm (line 10), the first and second-level directories themselves (line 11).

The instantiation of `rank_init_gen` is a matter of passing the appropriate `param`eters and the functions `Dir1Arr0`, `Dir2Arr0`, etc. that we explained in Sect. 5.2:

```
Definition rank_init b s : Aux :=
  let param := rank_init_param (bsize s) in
  let w1 := w1_of param in let w2 := w2_of param in
  mkAux b s param
    (rank_init_gen b s param
      Dir1Arr0 Dir1Arr empty_dir1 (push_dir1 w1) finalize_dir1
      Dir2Arr0 Dir2Arr empty_dir2 (push_dir2 w2) finalize_dir2).
```

Similarly, `rank_lookup_gen` is instantiated with the `param`eters resulting from the execution of `rank_init` together with the functions `Dir1Arr`, `Dir2Arr`, etc. from Sect. 5.2:

```
Definition rank_lookup (aux : Aux) i :=
  let b := query_bit aux in
  let param := parameter aux in
  let w1 := w1_of param in let w2 := w2_of param in
  rank_lookup_gen b (input_bits aux) param
    Dir1Arr (lookup_dir1 w1) Dir2Arr (lookup_dir2 w2)
    (directories aux) i.
```

### 5.4 Functional Correctness of Jacobson's Algorithm in Coq

The functional correctness of Jacobson's algorithm is stated using the generic *rank* function (`rank_lookup_gen`, Sect. 3.1) with its formal specification (`rank`, Sect. 2.1). Regarding the directories, we just assume that they are arrays in the sense of Sect. 5.1. We abbreviate as `...` the many parameters (`push_dir1`, `lookup_dir1`, etc.) corresponding to the array interface:

```
Lemma rank_lookup_gen_ok_to_spec : forall p dirpair,
  p <= size input_bs ->
  dirpair = rank_init_gen b input_bs param ... ->
  rank_lookup_gen b input_bs param ... dirpair p = rank b p input_bs.
```

Since this proof is carried out at the level of the array interface, it does not deal with the low-level details of directories implementation (e.g., the bitstring reversal caused by `wbitrev` and `finalize_dir1` explained in Sect. 5.2). Thanks to the abstraction, we expect proofs done at the level of this interface to be reusable for other algorithms for succinct data structures that use directories such as the *select* function.

### 5.5 Storage Complexity of Auxiliary Data Structures

The required storage depends on the parameters of Jacobson's algorithm explained in Sect. 5.3. They should be chosen appropriately to achieve $o(n)$ space complexity. We use the following parameters (Coq formalization on the right). They are taken from [3, Sect 2.2.1] (we add 1 to $\mathrm{sz}_2$ and $k$ to make them strictly positive for all $n \geq 0$).

$$
\begin{aligned}
k &= \lceil \log_2(n+1) \rceil + 1 \\
\mathrm{sz}_2 &= \lceil \log_2(n+1) \rceil + 1 \\
\mathrm{sz}_1 &= k \times \mathrm{sz}_2 = (\lceil \log_2(n+1) \rceil + 1)^2 \\
w_1 &= \lceil \log_2(\lfloor n/\mathrm{sz}_2 \rfloor \times \mathrm{sz}_2 + 1) \rceil \\
w_2 &= \lceil \log_2((k-1) \times \mathrm{sz}_2 + 1) \rceil
\end{aligned}
$$

```
Definition rank_default_param n :=
  let kp := bitlen n in (* k - 1 *)
  let sz2p := bitlen n in (* sz2 - 1 *)
  let sz2 := sz2p.+1 in
  let nn := n %/ sz2 in
  let w1 := bitlen (n %/ sz2 * sz2) in
  let w2 := bitlen (kp * sz2) in
  mkParam kp sz2p nn w1 w2.
```

Using these parameters, we proved the asymptotic storage requirement for the auxiliary data structures. The space complexity is $o(n)$, more precisely $\frac{n}{\log_2 n} + \frac{2n \log_2 \log_2 n}{\log_2 n}$, similarly to [3, Theorem 2.1]. For the sake of illustration, let us show how we prove in Coq that the contribution of the first-level directory to space complexity is indeed $\frac{n}{\log_2 n}$. First, we fix *rank*'s parameters using the following declaration:

```
Definition rank_init_param n := rank_param_nonzero_word_size (rank_default_param n).
```

`rank_default_param` has been explained just above. `rank_param_nonzero_word_size` is just a technicality to take care of the uninteresting case where the length of input bitstring is zero[5]. The contribution of the first-level directory to space complexity is the length of the bitstring that represents it, i.e., `size (directories (rank_init b s)).1` (`.1` stands for the first projection of a pair). In Coq, we proved the following lemma about this length:

```
Lemma rank_aux_space_dir1_exact b (s : bits) :
  size (directories (rank_init b s)).1 = let n := size s in
    ((n %/ (bitlen n).+1) %/ (bitlen n).+1).+1 *
      (bitlen (n %/ (bitlen n).+1 * (bitlen n).+1)).-1.+1.
```

(`.-1` is notation for the predecessor function.) In other words, since `bitlen` $n \sim \log_2 n$, we have shown that the length of the first-level directory is asymptotically equal to $\frac{n}{\log_2 n}$.

# 6 Final Extraction and Benchmark

In this section, we extract the *rank* function from Sect. 5.3 using the OCaml library for bitstrings from Sect. 4.3 and benchmark the extracted function to check that its execution is constant-time.

## 6.1 Extraction of the Verified *rank* Function

Because we used abstractions in Coq, we must be careful about inlining at extraction-time to obtain OCaml code as efficient as possible. (Recall that extraction of library functions on bitstrings has been discussed independently in Sect. 4.1.) In particular, we need to ensure that the function parameters we have introduced for modularity using Coq's `Section`s are inlined. Concretely, we inline most function calls (except `bcount` and `wnth`) using the following Coq command:

```
Extraction Inline empty_dir1 push_dir1 finalize_dir1 lookup_dir1 ... .
```

As a result, `rank_lookup` looks like an hand-written program, prefix notations aside (see appendix A for the extracted `rank_lookup` and `rank_init` functions). As for the function `rank_init_iter` in `rank_init`, we obtain a tail-recursive OCaml function, as we program in Coq, so that it should use constant-size stack independently of the input bitstring.

Since we obtain almost hand-written code, we can expect `ocamlopt` to provide us with all the usual optimizations. There are however specific issues due to Coq idiosyncrasies. For example, the pervasive usage of the successor function `S` for natural numbers is extracted to a call to the OCaml function `Pervasives.succ` that `ocamlopt` luckily turns into an integer increment. (One can check which inlining `ocamlopt` has performed by using `ocamlopt -dclambda`.) In contrast, anonymous function

[5] In this case, `w1` and `w2` become 0 and our word array cannot distinguish an empty array and non-empty array.

calls produced by extraction may be responsible for inefficiencies. For example, the mapping from Coq `nat` to OCaml `int` is defined as follows (file `ExtrOcamlNatInt.v` from the Coq standard library) :

```
Extract Inductive nat => int [ "0" "Pervasives.succ" ]
 "(fun fO fS n -> if n=0 then fO () else fS (n-1))".
```

It is responsible for calls of the form `(fun fO fS n -> ...) (fun _ -> E1) (fun jp -> E2)` (see `rank_init` in appendix A) that `ocamlopt` unfortunately cannot $\beta$-reduce.

### 6.2 Benchmarking of the Verified *rank* Function

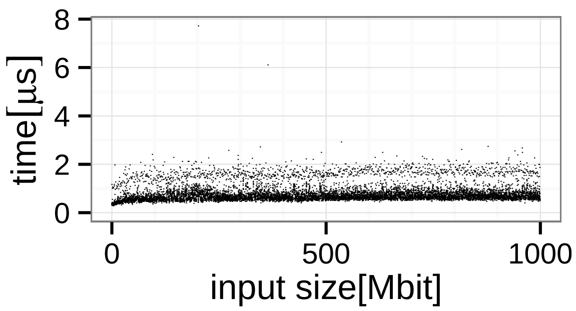


**Figure 3.** Performance of rank lookup

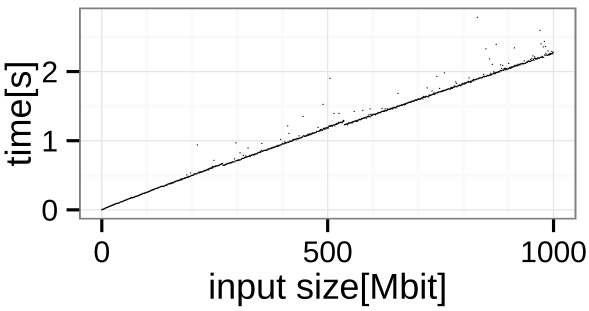


**Figure 4.** Performance of rank initialization

Fig. 3 shows the performance of a single lookup invocation for the *rank* function by measuring the time taken by `rank_lookup aux i` for inputs up to 1000Mbit (recall that the input string `s` is part of `aux`). We make measurements for 1000 values of the input size $n$. For each $n$, we make 10 measures for `i` between 0 and $n$. The measurement order is randomized ($n$ and `i` are picked randomly).

Execution seems constant-time (0.80µs on average) w.r.t. the input size. One can observe that execution seems a bit faster for small inputs. We believe that this is the effect of memory cache. One can also observe that the result is noisy. We believe that this is because of memory cache with access patterns and some instructions, such as IDIV (integer division), that use a variable number of clock cycles [7].

Fig. 4 shows the performance of initialization for the *rank* function by measuring the time taken by `rank_init` for inputs up to 1000Mbit. We make measurements for 1000 values of the input size. As expected, the result seems linear. There are several small gaps, for input size 537Mbit for example. This happens because the parameters for Jacobson's *rank* algorithm are changed at this point: $sz_2$ and `k` are changed from 30bit to 31bit, `w1` is changed from 29bit to 30bit. As a result, the size of the first-level directory decreases from 17.3Mbit to 16.8Mbit and the second-level directory, from 179Mbits to 173Mbits, leading to a shorter initialization time.

**Benchmark Environment** The operating system is Debian GNU/Linux 8.2 (Jessie) amd64 and the CPU is the Intel Core i7-4510U CPU (2.00GHz, Haswell). The time is measured using the `clock_gettime` function with the `CLOCK_PROCESS_CPUTIME_ID` resolution set to 1ns. The *rank* implementation is extracted by Coq 8.5beta2 and compiled to a native binary with `ocamlopt` version 4.02.3. C programs are compiled with gcc 4.9.2 with options `-O -march=native` (`-march=native` is used to enable POPCNT and LZCNT of recent Intel processors).

**About OCaml's Garbage Collector** `Gc.full_major` and `Gc.compact` are invoked before each measurement to mitigate the garbage collection effect. Garbage collection does not occur during lookup measurements (`major_collections` and `minor_collections` of `Gc.stat` are unchanged). During initialization measurements, the garbage collection has a small impact. Indeed, in Fig. 4, major garbage collections happen at most 232 times during one initialization measurement. Moreover, using another experiment with `gprof`, we checked that the time spent by garbage collection during the `rank_init` benchmark with `Gc.full_major` and `Gc.compact` disabled accounts for less than 5%.

## 7 Discussion and Perspectives

**About Complexity** For the time being, we limited ourselves to benchmarking the extracted code for time-complexity. It would be more convincing to perform formal verification using a monadic approach (e.g., [12]). We have addressed the issue of space-complexity in Sect. 5.5. In general, one may also wonder about the space-complexity of intermediate data structures. In this paper, we obviously did not build any but this could also be addressed by counting the number of cons cells using a monad.

**About Extraction of Natural Numbers** In this paper, there is no problem when we extract Coq `nat` to OCaml `int`, despite the fact that `nat` has no upper-bound. OCaml `int`s are $(w-1)$-bit signed integers that can represent positive integers less than $2^{w-2}$ ($w$ is the number of bits in a machine word) [10]. However, the maximum number of bits in an OCaml `bytes` is $2^{w-10}w$ bits because one `bytes` is less than $2^{w-13}w$ bytes [10]. Since $2^{w-10}w < 2^{w-2}$ for $w = 32$ and $w = 64$, an `int` can represent the number of bits in a `bytes`. For this reason, `nat` arguments of functions such as `bnth` or intermediate values in the *rank* algorithm do not overflow when turned into `int`. This can be ensured during formal verification by using a type for fixed-size integers (such as `int : nat -> Type` in [1]) instead of natural numbers.

**About Alignment** The extracted code can be further optimized by insisting on having the size (`w1`, `w2` in this paper) of the integers in the directories to be a multiple of 8. Indeed, depending on the size of the input, reading the entries of the directories may require bit operations such as masking and shifting. This overhead can be eliminated if `w1`, `w2` are multiples of 8, simply by modifying `rank_default_param`.

**About the Correctness of OCaml Code** There are some ways to improve confidence in the correctness of our OCaml library for bitstrings. Formal verification may be used to guarantee the time-complexity properties. For example, to achieve linear-time construction of arrays with `bappend` (Sect. 4.3), there must be no sharing (as in a tree) between the lists used in the Coq code (in other words, cons cells should be cons'd at most once). The approach that we are currently exploring to ensure this property is to augment the *rank* function with an appropriate monad.

In contrast, formal verification of the OCaml code need not be addressed in priority. We have already implemented a test suite for the OCaml bitstring library using OUnit [14]. Concretely, we regularly test functions for `bits` by comparison with list functions using random bitstrings. We have also tested the extracted *rank* function by comparison with an extracted `bcount` function on random bitstrings and never found any bug. Since we plan to reuse this library for other functions, it will endure even more testing. Moreover, formal verification of OCaml seem very difficult as of today because we are relying on unspecified features regarding optimization, `Obj.magic`, and C.

**About Performance of the Extracted Implementation** At this stage, it is difficult to make a comparison with rich (but not verified) libraries for succinct data structures. Yet, we have good reasons to believe that extracted OCaml code can be fast enough for practical purpose. For example, we have observed that the SDSL [13] *rank* function for $H_0$-compressed vectors executes in about $0.1 \sim 1.8\mu$s depending on algorithm's parameters while our *rank* function executed in $0.8\mu$s (see Sect. 6.2). To be fair, it is likely that our *rank* function consumes more memory since Jacobson's algorithm does not compress its input.

## 8 Conclusion

We discussed the verification of an OCaml implementation of the *rank* function for succinct data structures. We carried out formal verification in the Coq proof-assistant, from which the implementation was automatically extracted. We assessed not only functional correctness but also storage requirements, thus ensuring that data structures are indeed succinct. To obtain efficient code, we developed a new OCaml library for bitstrings whose interface match the Coq lists used in formal verification. Fig. 5 summarizes our experiment. To the best of our knowledge, this is the first application of formal verification to succinct data structures.

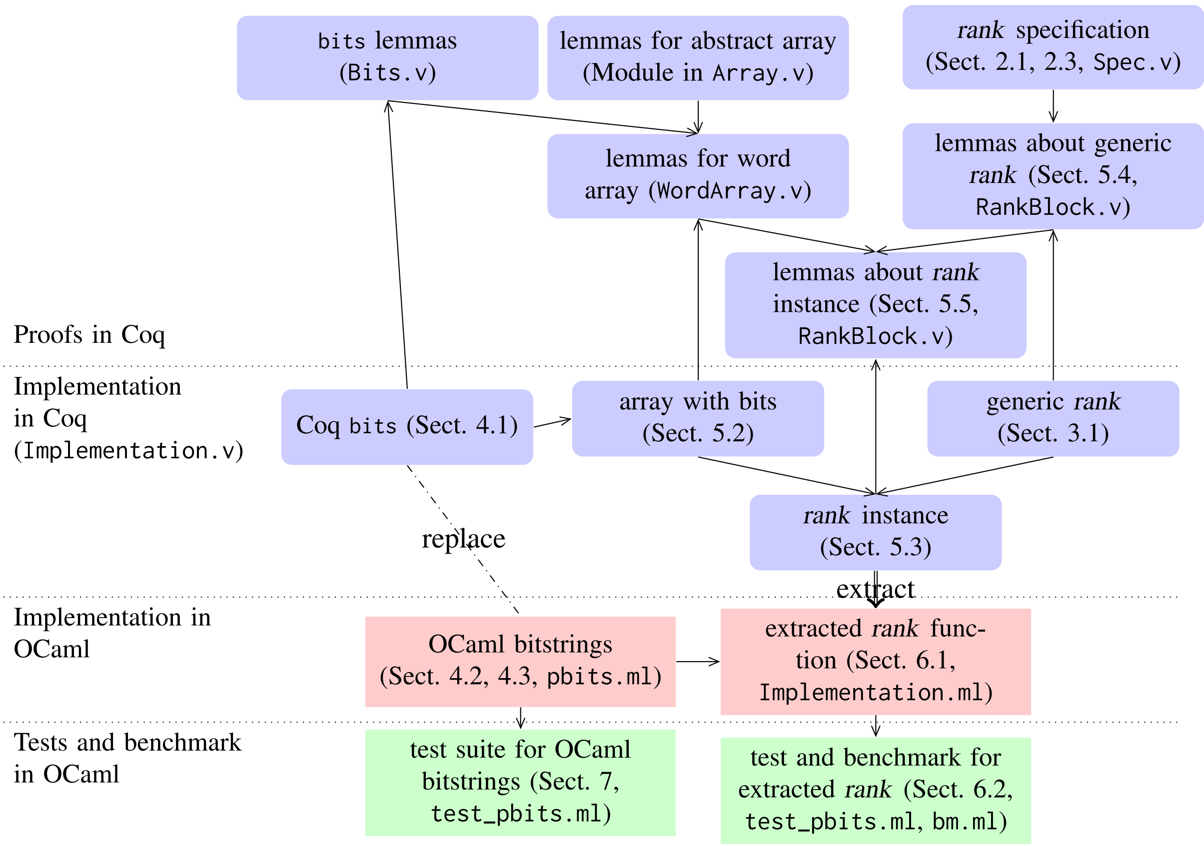


**Figure 5.** Dependency graph for the verification of Jacobson's algorithm. Arrows $A \leftarrow B$ read as "A depends on B". Relevant parts implementation files are indicated for browsing [16].

We believe that the libraries developed for the purpose of our experiment are reusable: the OCaml library for bitstrings of course, the array interface for directories (that are used by other functions for succinct data structures), lemmas developed for the purpose of formal specification of *rank*. We also discussed a number of issues regarding extraction from Coq to OCaml: the interplay between inlining at extraction-time and by the OCaml compiler, the soundness of code replacement at extraction-time, etc. Based on the results of this paper, we are now tackling formal verification of *rank*'s counterpart function *select* and plan to address more advanced algorithms.

**Acknowledgments** The authors are grateful to the anonymous reviewers of PPL2016 for their helpful comments. This work is partially supported by a JSPS Grant-in-Aid for Scientific Research (Project Number: 15K12013).

# References

[1] R. Affeldt, N. Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly. In: ASIAN 2006. LNCS, vol. 4435, pp. 346–360. Springer, 2008.

[2] M. Armand, B. Grégoire, A. Spiwack, L. Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In: ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, 2010.

[3] D. Clark. Compact Pat Trees. Doctoral Dissertation. University of Waterloo, 1996.

[4] Free Software Foundation. GCC 4.9.2 Manual. `http://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc`. 2014.

[5] G. Gonthier, A. Mahboubi, E. Tassi. A Small Scale Reflection Extension for the Coq system. Version 16. Technical report RR-6455. INRIA, 2015.

[6] Intel Advanced Vector Extensions Programming Reference. Jun. 2011.

[7] Intel 64 and IA-32 Architectures Optimization Reference Manual. Sep. 2015.

[8] Intel SSE4 Programming Reference. Apr. 2007.

[9] G. Jacobson. Succinct static data structures. Doctoral Dissertation. Carnegie Mellon University, 1988.

[10] R. W.M. Jones. A beginners guide to OCaml internals. `https://rwmj.wordpress.com/2009/08/04/ocaml-internals`. 2009.

[11] D. K. Kim, J. C. Na, J. E. Kim, K. Park. Efficient Implementation of Rank and Select Functions for Succinct Representation. In: WEA 2005. LNCS, vol. 3503, pp. 315–327. Springer, 2005.

[12] T. Nipkow. Amortized Complexity Verified. In: ITP 2015. LNCS, vol. 9236, pp. 310–324. Springer, 2015.

[13] SDSL: Succinct Data Structure Library. `https://github.com/simongog/sdsl-lite`.

[14] OUnit: Unit test framework for OCaml. `http://ounit.forge.ocamlcore.org/`.

[15] D. Okanohara. The world of fast character string analysis. (In Japanese.) Iwanami Shoten, 2012.

[16] A. Tanaka, R. Affeldt, J. Garrigue. Formal Verification of the rank Function for Succinct Data Structures. `https://staff.aist.go.jp/tanaka-akira/succinct/index.html`.

# A Core Part of the Extracted OCaml Code

```
let rank_lookup aux0 i =
  let b = aux0.query_bit in
  let param0 = aux0.parameter in
  let w1 = param0.w1_of in
  let w2 = param0.w2_of in
  let dirpair = aux0.directories in
  let j2 = (/) i (Pervasives.succ param0.sz2p_of) in
  let j3 = (mod) i (Pervasives.succ param0.sz2p_of) in
  let j1 = (/) j2 (Pervasives.succ param0.kp_of) in
  (+) ((+) (wnth w1 j1 (fst dirpair)) (wnth w2 j2 (snd dirpair)))
    (Pbits.bcount b (( * ) j2 (Pervasives.succ param0.sz2p_of)) j3
      aux0.input_bits)

let rank_init b s =
  let param0 = rank_init_param (Pbits.bsize s) in
  let w1 = param0.w1_of in
  let w2 = param0.w2_of in
  { query_bit = b; input_bits = s; parameter = param0; directories =
  (let (dir1, dir2) =
     let rec rank_init_iter j i n1 n2 dir1 dir2 =
       let m =
         Pbits.bcount b
           (( * ) ((-) param0.nn_of j) (Pervasives.succ param0.sz2p_of))
           (Pervasives.succ param0.sz2p_of) s
       in
       ((fun fO fS n -> if n=0 then fO () else fS (n-1))
          (fun _ ->
          let dir1' = let n = (+) n1 n2 in wcons w1 (Pbits.wbitrev w1 n) dir1
          in
          let dir2' = let n = 0 in wcons w2 (Pbits.wbitrev w2 n) dir2 in
          ((fun fO fS n -> if n=0 then fO () else fS (n-1))
             (fun _ -> (dir1',
             dir2'))
             (fun jp ->
             rank_init_iter jp param0.kp_of ((+) n1 n2) m dir1' dir2')
             j))
          (fun ip ->
          let dir2' = wcons w2 (Pbits.wbitrev w2 n2) dir2 in
          ((fun fO fS n -> if n=0 then fO () else fS (n-1))
             (fun _ -> (dir1,
             dir2'))
```

```
          (fun jp ->
          rank_init_iter jp ip n1 ((+) n2 m) dir1 dir2')
          j))
       i)
  in rank_init_iter param0.nn_of 0 0 0 Pbits.bnil Pbits.bnil
in
((Pbits.breverse dir1), (Pbits.breverse dir2))) }
```