Formal Verification of the rank Algorithm for Succinct Data Structures

Akira Tanaka
Reynald Affeldt
Jacques Garrigue

2016-11-17

ICFEM2016

Motivation

More data with less memory

Why?

- For Big Data
  - Compact data representation reduces number of servers

How?

- Succinct Data Structures
  - designed to save memory
  - Succinct Spark is over 75x faster than native Apache Spark
    http://succinct.cs.berkeley.edu/wp/wordpress/
  - but at the price of complex, low-level algorithms

⇒ We need formal verification!

- To trust Big Data analysis

This Presentation

A realistic yet verified ***rank*** function

rank is the most important primitive of Succinct Data Structure

Contributions:

- Formal verification of ***rank*** using Coq
  - Functional correctness <u>and</u> storage requirements
- Automatic extraction from Coq of a realistic ***rank*** implementation
  - Main issue: limitations of naive Coq extraction
    - No array. Linear time access for list
    - Waste memory for list of booleans

Verified Properties

- Property 1: Functional correctness
  rank returns the expected value
- Property 2: Storage requirements
  The size of auxiliary data structure is the expected size

Coq Proof Assistant

- Proof assistant
- Programmer describes
  - program written in Gallina (ML-like language)
  - proposition on the program
  - proof for the proposition
- Coq checks the proof

Why We Use Coq

- Extraction: Programs written in Gallina can be extracted into OCaml, Haskell and Scheme
- Infinite states: Coq can check proofs on infinite states (unlike model checker)
- Static checking: Proof check has no runtime cost

Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Succinct Data Structures

A short history

Compact data representation but operations are still fast

- 1988 rank/select (bitstring), Jacobson
- 1989 LOUDS (tree), Jacobson
- 2000 FM-index (full text index), Ferragina, et al
- 2003 wavelet tree (fixed alphabet string), Grossi, et al
- 2003 compressed suffix array, Sadakane
- 2005 DFUDS (tree), Benoit, et al

rank Function

Informal specification

- "$rank_b$ i s" counts the number of "b" in the first "i" bits of "s" (which length is "n")

n = 23 bit

i = 17 bit

$rank_1$ 17 10000101101011101111101 = 9

Nine "1" bits

- Naive implementation needs O(i) time:

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

Jacobson's rank Algorithm

Overview

- Uses two auxiliary (precomputed) arrays

D1 = [0, 4, 10] # first-level directory

D2 = [0, 1, 2, 0, 1, 4, 0, 3] # second-level directory

- Split rank into 3 parts

17 = 9 + 6 + 2

$rank_1$ 17 10000101101011101111101 =

$rank_1$ 9 100001011 +

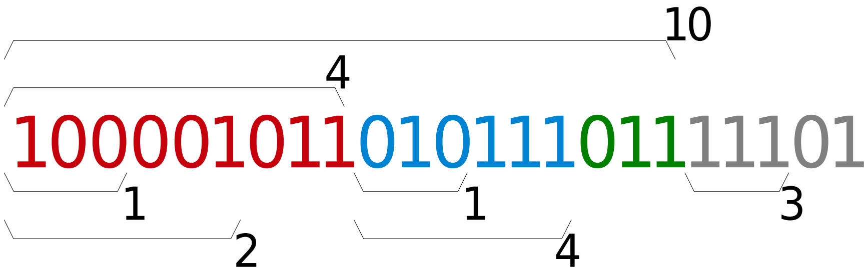
$rank_1$ 6 010111 +

$rank_1$ 2 011 =

D1[17/9] + D2[17/3] + $rank_1$ 2 011 = 9

Jacobson's rank Algorithm

Technical Details (D1, D2, etc.)



s = 10000101101011101111101 n = 23

sz1 = k×sz2=9 k = 3 # sz1: big block size

sz2 = 3 # sz2: small block size

D1 = [0,4,10] # first-level directory

D2 = [0,1,2,0,1,4,0,3] # second-level directory

$rank_1$ i s =

D1[i / sz1] + # O(1) time

D2[i / sz2] + # O(1) time

$rank_1$ (i % sz2) 011 # O(sz2) time, naively

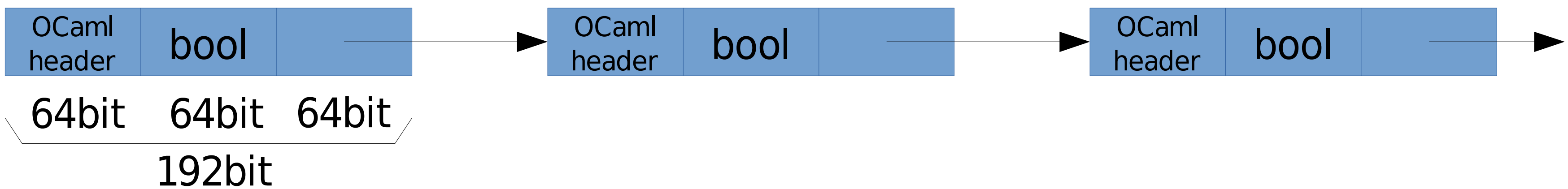
Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Coq Extraction Problem

Default bitstring representation

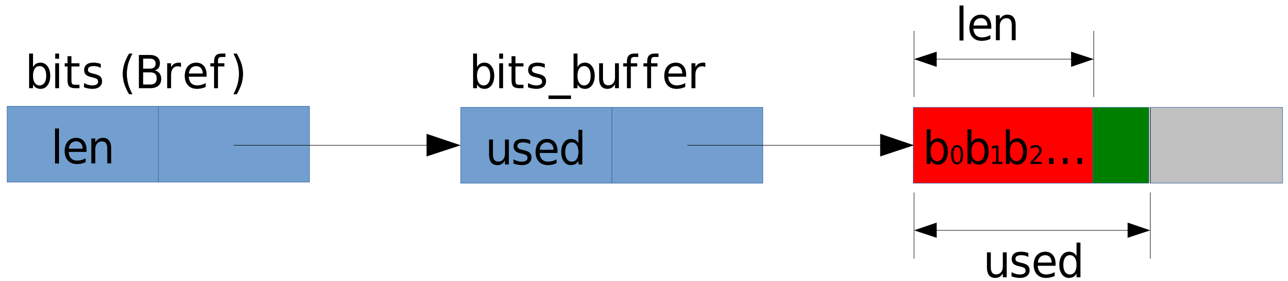
- (* Coq/Ssreflect *)
  Inductive bits : Type := bseq of seq bool. (* seq bool is list bool in Coq *)
  Extraction bits.
  (* OCaml: *)
  type bits = bool list (* usual OCaml list *)



- Problem 1: Linear time random access
  We need constant time random access for succinct data structures!
- Problem 2: Waste of memory space
  3 words / bit (192 times bigger than required on 64bit architecture)

A New OCaml Bitstring Library

- Constant time random access
- Dense representation (1 bit / bit)
- type bits_buffer =
  { mutable used : int; data : bytes; }
  type bits = Bref of int * bits_buffer



Coq List Functions and OCaml Array functions

Coq functions are replaced with OCaml functions at extraction

- bsize s
  count the length of "s"
  - Coq: scans a list, O(n)
  - OCaml: just returning "len" field, O(1)
- bappend s1 s2 (* bsize s1 = len1, bsize s2 = len2 *)
  append "s1" and "s2"
  - Coq: copy s1, O(len1)
  - OCaml: append s2 into s1 destructively if possible, O(len2)
    copy s1 and s2 otherwise, O(len1+len2)
- bcount b i l s
  count "b" bits in "l" bits from "i"'th bits in "s"
  - Coq: skip first "i" bits and scans "l" bits, O(i+l)
  - OCaml: random access and uses POPCNT instruction, O(l)

Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

rank Formalization in Coq

- Define rank_init_gen and rank_lookup_gen
  Algorithm parameters: sz1, etc.
  Array definition is abstracted
  - rank_init_gen: precompute auxiliary data
  - rank_lookup_gen: compute rank value
- Instantiate rank_init and rank_lookup
  Bound parameters
  - algorithm parameters: sz1, etc.
  - array definition

Generic rank init. function

Construct D1 and D2

- Scan s from left to right, tail recursion
- O(n) time expected
- Array functions (emptyD1, etc.) are parameters

```
Fixpoint buildDir j i n1 n2 D1 D2 :=
  let m := bcount b ((nn - j) * sz2) sz2 s in
  if i is ip.+1 then
    let D2' := pushD2 D2 n2 in
    if j is jp.+1 then buildDir jp ip n1 (n2 + m) D1 D2'
    else (D1, D2')
  else
    let D1' := pushD1 D1 (n1 + n2) in
    let D2' := pushD2 D2 0 in
    if j is jp.+1 then buildDir jp kp (n1 + n2) m D1' D2'
    else (D1', D2').
Definition rank_init_gen := buildDir nn 0 0 0 emptyD1 emptyD2.
```

Generic rank lookup function

- No loop
- O(1) time expected
- Array functions (lookupD1, etc.) are parameters

```
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 *)
  lookupD1 j1 D1  +  lookupD2 j2 D2  +  bcount b (j2 * sz2) j3 input_s.
```

- MathComp notation:
  - x %/ y $\lfloor x / y \rfloor$
  - x %% y x mod y

Instantiate rank Functions

Specify parameters to rank_{lookup,init}_gen

- Algorithm parameters: sz1, sz2, etc.
- Array functions: lookupD1, etc.

```
Definition rank_lookup 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
    D1Arr (lookupD1 w1) D2Arr (lookupD2 w2)
    (directories aux) i.
```

Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Formal Verification

- Property 1: Functional correctness
  rank returns the expected value
- Property 2: Storage requirements
  D1 and D2 are of the expected size

Functional Correctness

- Implemented rank returns same value as the simple rank function

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

- Arrays work as expected
  Array lookup returns the pushed value
- rank_init and rank_lookup also works as expected

Parameters and Last rank

		sz1	sz2	last rank
1988	Jacobson	$(\log_2 n)(\log n)$	$\log_2 n$	linear scan, O($\log_2 n$)
1996	Clark	$(\log_2 n)^2$	$\log_2 n$	table lookup c times (1 < c), O(1)
1999	Benoit, et al	$(\log_2 n)^2$	$\frac{(\log_2 n)}{2}$	table lookup once, O(1)
2016	Ours	$(\log_2 n)^2$	$\log_2 n$	POPCNT, O(1)

We uses Clark's parameters
but avoid the table for last rank

Our Parameters

- sz1 = (bitlen n + 1)² $\sim (\log_2 n)^2$
- sz2 = bitlen n + 1 $\sim \log_2 n$

where bitlen x = $\lceil \log_2 (x+1) \rceil$

- w1 = bitlen ($\lfloor$n / sz2$\rfloor$×sz2) # D1 element size
- w2 = bitlen ((sz1/sz2-1)×sz2) # D2 element size
- D1 size: ($\lfloor$n/sz1$\rfloor$+1)×w1 [bit]
- D2 size: ($\lfloor$n/sz2$\rfloor$+1)×w2 [bit]
- Use POPCNT, no table to count one bits

Storage Requirements

- Directory size of implementation

```
Lemma rank_spaceD1 b s :
  size (directories (rank_init b s)).1 = let n := size s in let m := bitlen n in
  ((n %/ m.+1) %/ m.+1).+1 * (bitlen (n %/ m.+1 * m.+1)).-1.+1.
Lemma rank_spaceD2 b s :
  size (directories (rank_init b s)).2 = let n := size s in let m := bitlen n in
  (n %/ m.+1).+1 * (bitlen (m * m.+1)).-1.+1.
```

- This is same as Clark's paper

size of D1 + size of D2 $\sim \frac{n}{\log_2 n} + \frac{2n \log_2 \log_2 n}{\log_2 n} \in o(n)$

The storage requirement for auxiliary data structure is ignorable if n is large enough
I.e. This is a succinct data structure

Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Complexity of OCaml Bitstring Functions

Library Overview

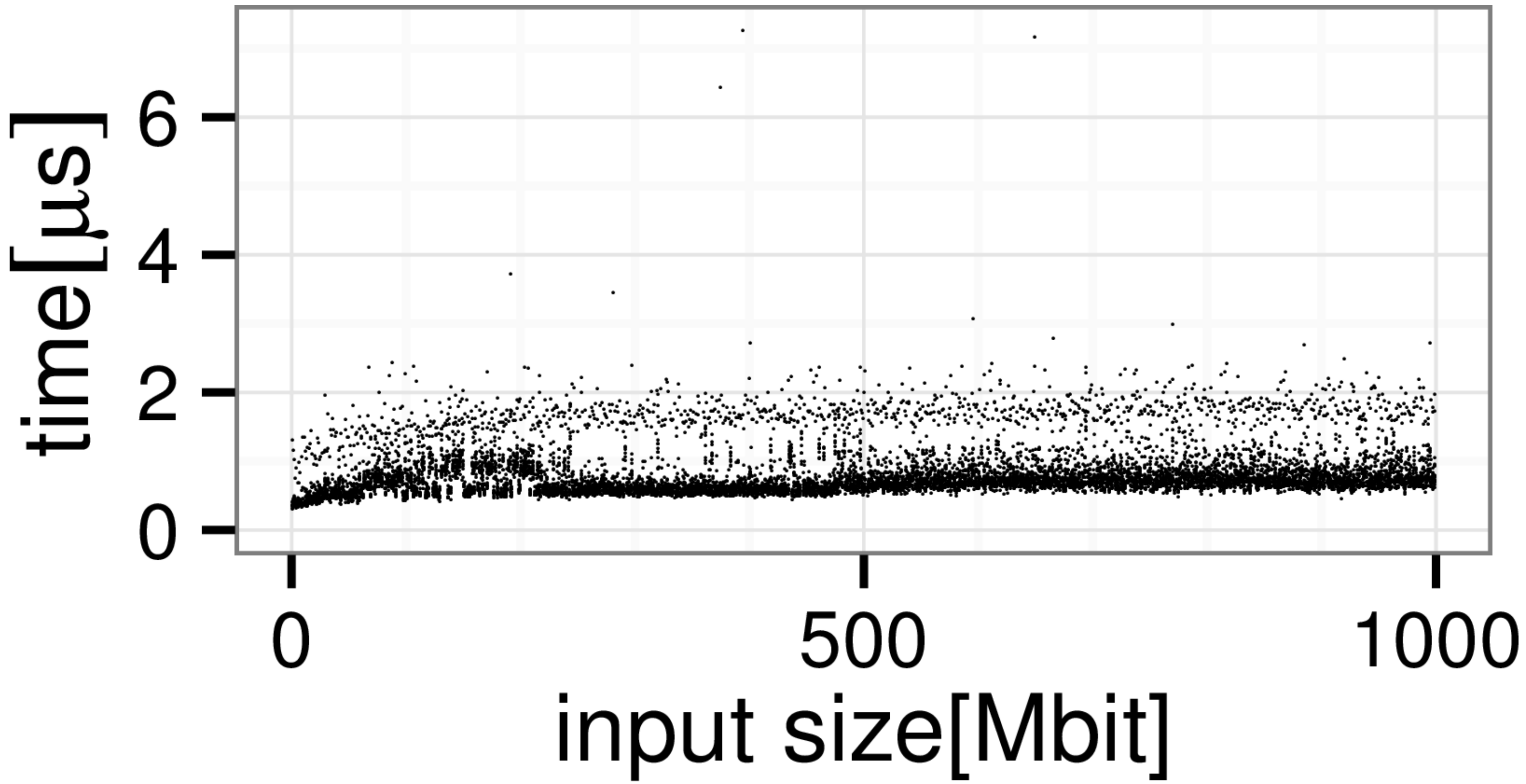
- Array construction in linear time
  - let s = bappend bnil s1 in
    let s = bappend s s2 in …
    s
  - Always len1 = used1 and bappend is O(len2), this works in O(total len) time
  - bits_buffer is doubled when bits_buffer is full
    Amortized copy cost doesn't increase complexity
- Random access in constant time
  - random access in a bytes by bcount

Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

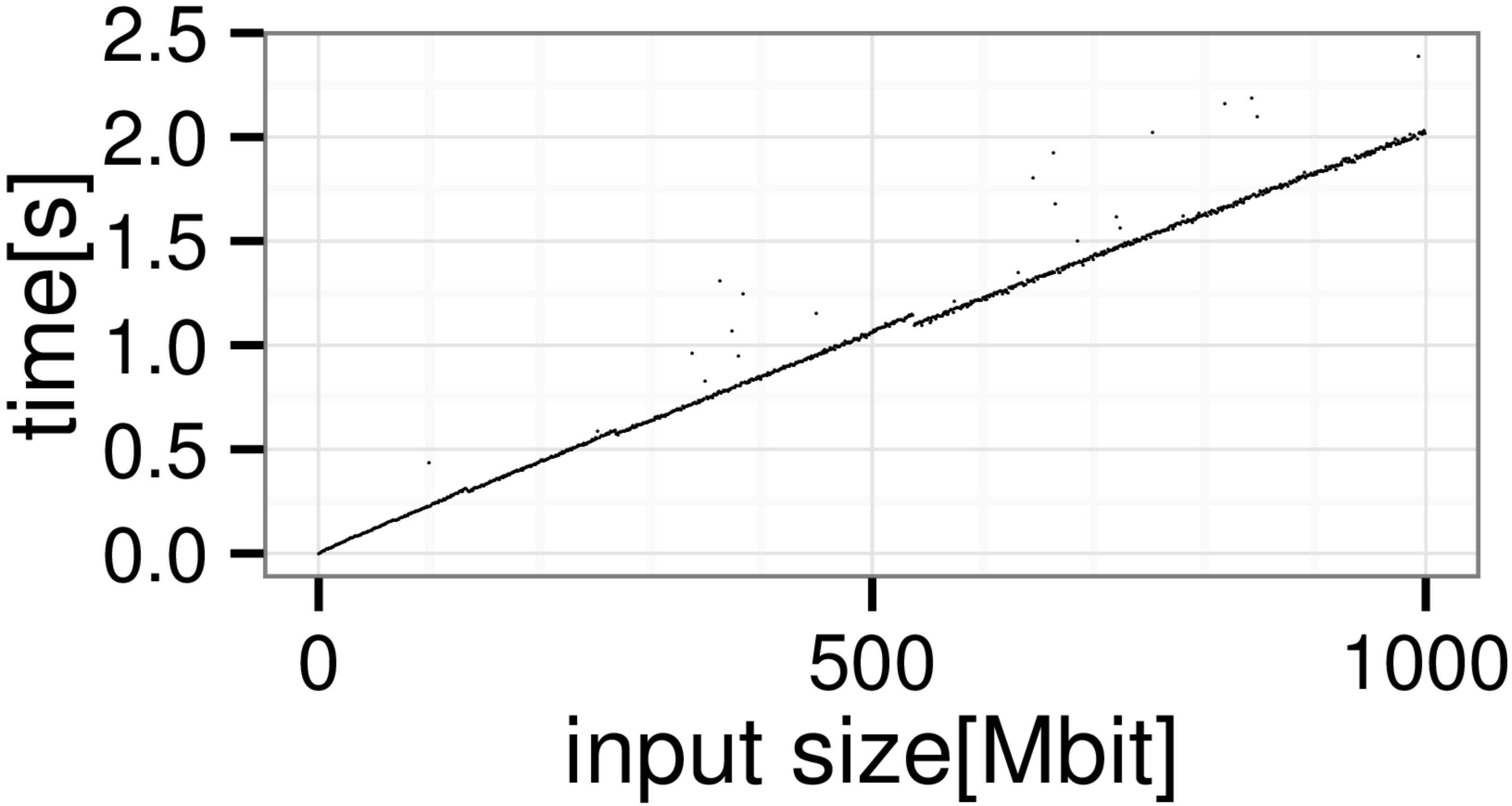
rank_lookup Benchmark

- Lookup seems O(1). Average 0.83[μs]
- Memory cache effect for small input



rank_init Benchmark

- Initialization seems O(n)
- sz2 increment causes small gaps



Outline

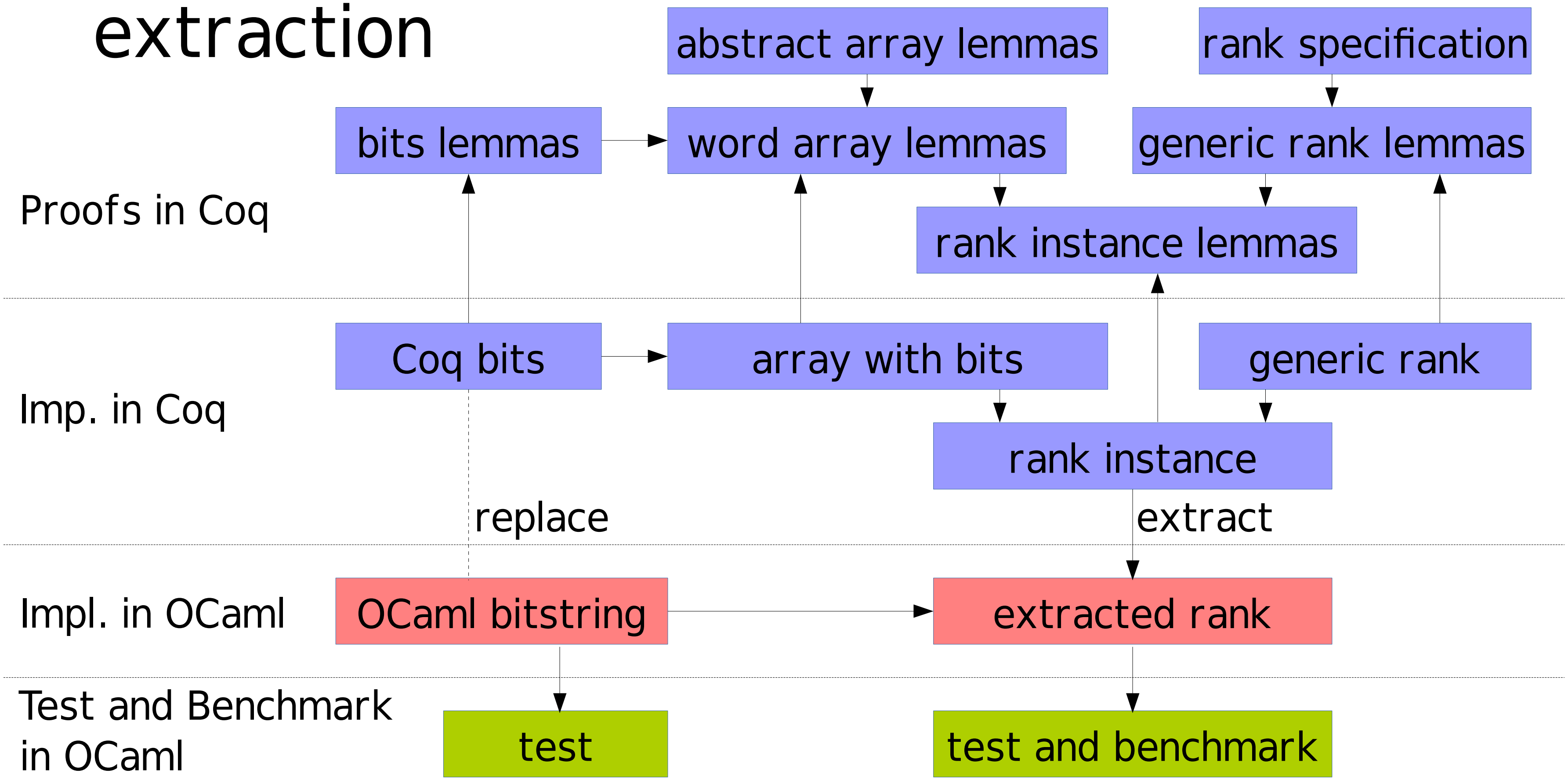
1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Array impl. using Bitstring

- Array construction and lookup functions Defined for D1 and D2
  - Definition emptyD1 := bnil.
  - Definition pushD1 w1 s n := bappend s (bword w1 n).
  - Definition lookupD1 w1 i s := wnth w1 i s.
- Utility functions
  - bword w n creates a short bitstring consists of lower w bits of n
  - wnth w i s returns i'th word in s with w bit words

Modularized Verification

- Array imp. and rank alg. are modularized.
- Modular implementation is inlined at extraction



Outline

1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Summary

- OCaml bitstring library implemented
- rank function extracted
- Formal verification on rank function
  - Functional correctness
  - Storage requirements
- Expected time complexity confirmed
  - Constant time lookup
  - Linear time initialization

Future Work

- Verify complexity using monad
  - Time complexity
  - Space complexity including intermediate data
- Avoid mapping from Coq nat to OCaml int using finite-size integers
- Implementation considering memory alignment
- Formal verification for OCaml bitstring
- Comparison to other implementations
  We already benchmarked SDSL
  It seems our implementation is not too slow
- Implement and verify other succinct data structure algorithms, such as select

AIST

Extra Slides

Extracted rank_lookup

```
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 (Obj.magic b)
                    (( * ) j2 (Pervasives.succ param0.sz2p_of))
      j3 aux0.input_bits)
```

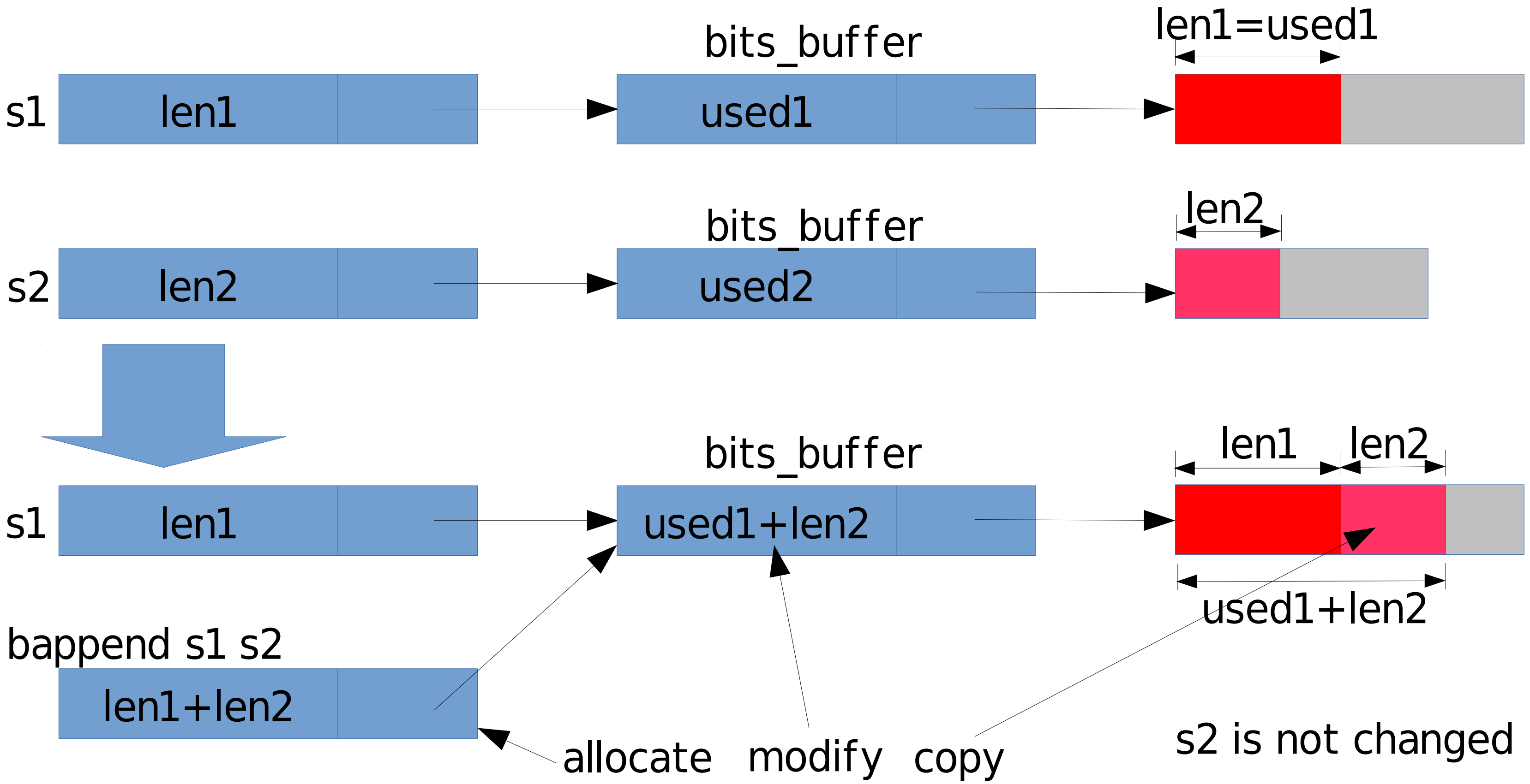
Extracted rank_init

```
let rank_init b s =
  let param0 = rank_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 rec buildDir j i n1 n2 d1 d2 =
     let m = Pbits.bcount (Obj.magic 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 d1' = wrcons w1 d1 ((+) n1 n2) in
        let d2' = wrcons w2 d2 0 in
        ((fun fO fS n -> if n=0 then fO () else fS (n-1))
           (fun _ -> (d1', d2'))
           (fun jp -> buildDir jp param0.kp_of ((+) n1 n2) m d1' d2') j))
        (fun ip ->
        let d2' = wrcons w2 d2 n2 in
        ((fun fO fS n -> if n=0 then fO () else fS (n-1))
           (fun _ -> (d1, d2'))
           (fun jp -> buildDir jp ip n1 ((+) n2 m) d1 d2') j))
        i)
   in buildDir param0.nn_of 0 0 0 Pbits.bnil Pbits.bnil) }
```

bappend s1 s2 Works in O(len2) time

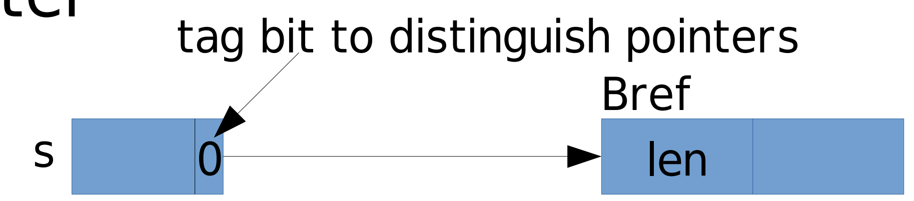
bappend s1 s2 is O(len2) time if len1=used1

where len1 = size s1, len2 = size s2

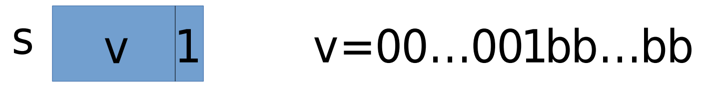


Short Bitstrings

- Non-constant constructor is implemented with a pointer



- Short bitstrings including bnil are implemented with unboxed integers to avoid allocations (Obj.magic is used)
  - It can represent up to 62bit bitstrings on 64bit environment



- Bdummy0 and Bdummy1 avoid SEGV

type bits = Bdummy0 | Bdummy1 |
  Bref of int * bits_buffer

Extraction Coq Lists to OCaml Bitstrings

- (* Coq/Ssreflect *)
  Inductive bits : Type := bseq of seq bool.
  Extract Inductive bits => "Pbits.bits" ["Pbits.bseq"] "Pbits.bmatch".
- Use OCaml definitions:
  - Pbits.bits type
  - Pbits.bseq function converts bool list to Pbits.bits
  - Pbits.bmatch function converts Pbits.bits to bool list
- Several Coq functions are replaced by functions defined in OCaml
  - bsize s : just returning "len" field which is O(1) time
  - bappend s1 s2 : append bits destructively if possible
  - bcount b i l s : count bits using POPCNT instruction

  rank implementation uses bappend and bcount
  bseq and bmatch is not used to avoid waste of memory