Formal Verification of the rank Function for Succinct Data Structures

Akira Tanaka
Reynald Affeldt
Jacques Garrigue

2016-03-08

PPL2016

Motivation

More data with less memory

Why?

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

How?

- Succinct Data Structures (簡潔データ構造)
  - designed to save memory
  - 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.

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

AIST

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

dir1 = [0, 4, 10] # first-level directory

dir2 = [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 +

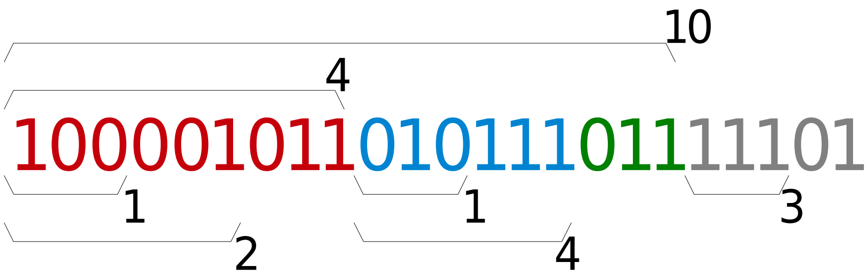
$rank_1$ 6 010111 +

$rank_1$ 2 011 =

dir1[17/9] + dir2[17/3] + $rank_1$ 2 011 = 9

Jacobson's rank Algorithm

Technical Details (dir1, dir2, etc.)



s = 10000101101011101111101   n = 23

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

sz2 = 3   # sz2: small block size

dir1 = [0,4,10]   # first-level directory

dir2 = [0,1,2,0,1,4,0,3]   # second-level directory

$rank_1$ i s =

  dir1[i / sz1] +   # O(1) time

  dir2[i / sz2] +   # O(1) time

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

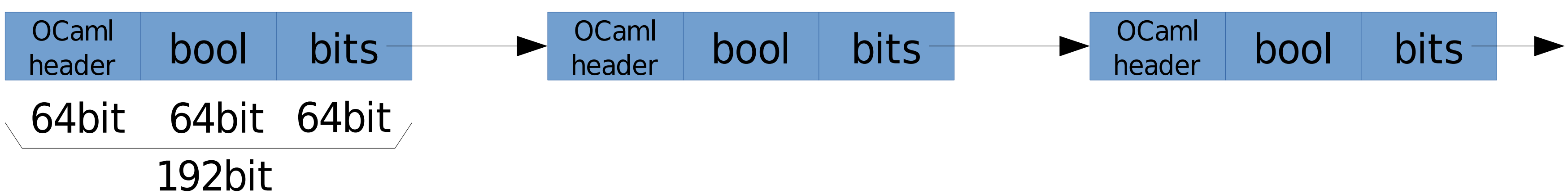
Outline

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

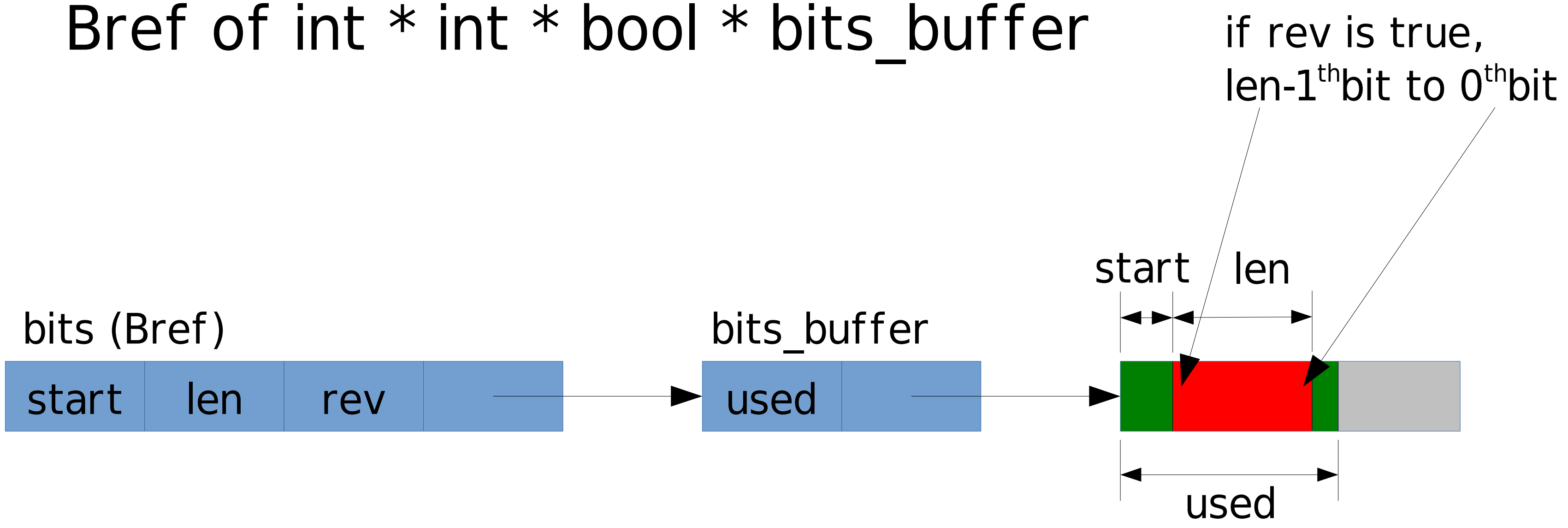
- (* Coq/Ssreflect *)
  Inductive bits : Type := bnil | bcons of bool & bits.
  Extraction bits.
  (* OCaml: *)
  type bits = | Bnil | Bcons of bool * bits



- 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; s : bytes; }
  type bits =
  Bref of int * int * bool * bits_buffer



Extraction Coq Lists to OCaml Bitstrings

- (* Coq *)
  Extract Inductive bits => "Pbits.bits"
  ["Pbits.bnil" "Pbits.bcons"] "Pbits.bmatch".
- Use OCaml definitions:
  - Pbits.bits type
  - Pbits.bnil constant
  - Pbits.bcons function
  - Pbits.bmatch function
- match s with bnil => E1 | bcons b t => E2 end (* Coq *)
  →
  Pbits.bmatch (fun () -> E1) (fun (b, t) -> E2) s (* OCaml *)

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

Generic rank lookup function

- No loop
- O(1) time expected

```
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.
```

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

Construct dir1 and dir2

- Scan s from left to right, tail recursion
- O(n) time expected

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

Instantiate rank Functions

Specify parameters to rank_{lookup,init}_gen

- Algorithm parameters: sz1, sz2, etc.
- Array functions: empty_dir1, etc.

```
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.
```

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
  dir1 and dir2 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 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.
```

- Arrays work as expected
  Array lookup returns the pushed value

Parameters and Last rank

Appropriate parameters for space complexity:

- 1988 Jacobson: $sz1=(\log_2 n)(\log n)$ $sz2=\log_2 n$
  last rank is linear scan: $O(\log_2 n)$
- 1996 Clark: $sz1=(\log_2 n)^2$ $sz2=\log_2 n$
  last rank is table lookup c times $(1 < c)$: $O(1)$
  table size: $n^{(1/c)}(\log_2 \log_2 n - \log_2 c)$
- 1999 Benoit, et al: $sz1=(\log_2 n)^2$ $sz2=(\log_2 n)/2$
  last rank is table lookup once: $O(1)$
  table size: $n^{(1/2)}(\log_2 \log_2 n - 1)$

Count one bits in a word:

- 1972 HAKMEM: Item 169 Count ones
- 2002 Hacker's Delight
- 2008 Intel SSE4.2, POPCNT instruction

Our Parameters

- $sz1 = (\text{bitlen } n + 1)^2$
- $sz2 = \text{bitlen } n + 1$

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

- $w1 = \text{bitlen } (\lfloor n / sz2 \rfloor \times sz2)$ # dir1 element size
- $w2 = \text{bitlen } ((sz1/sz2-1) \times sz2)$ # dir2 element size
- dir1 size: $(\lfloor n/sz1 \rfloor + 1) \times w1$ [bit]
- dir2 size: $(\lfloor n/sz2 \rfloor + 1) \times w2$ [bit]
- Use POPCNT, no table to count one bits

Storage Requirements

- Directory size of implementation

```
rank_aux_space_dir1 n =
((n %/ (bitlen n).+1) %/ (bitlen n).+1).+1 *
(bitlen (n %/ (bitlen n).+1 * (bitlen n).+1)).-1.+1

rank_aux_space_dir2 n =
(n %/ (bitlen n).+1).+1 * (bitlen (bitlen n * (bitlen n).+1)).-1.+1
```

- This is same as Clark's paper

rank_aux_space_dir1 n + rank_aux_space_dir2 n $\sim \frac{n}{\log_2 n} + \frac{2n \log_2 \log_2 n}{\log_2 n} \in o(n)$

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

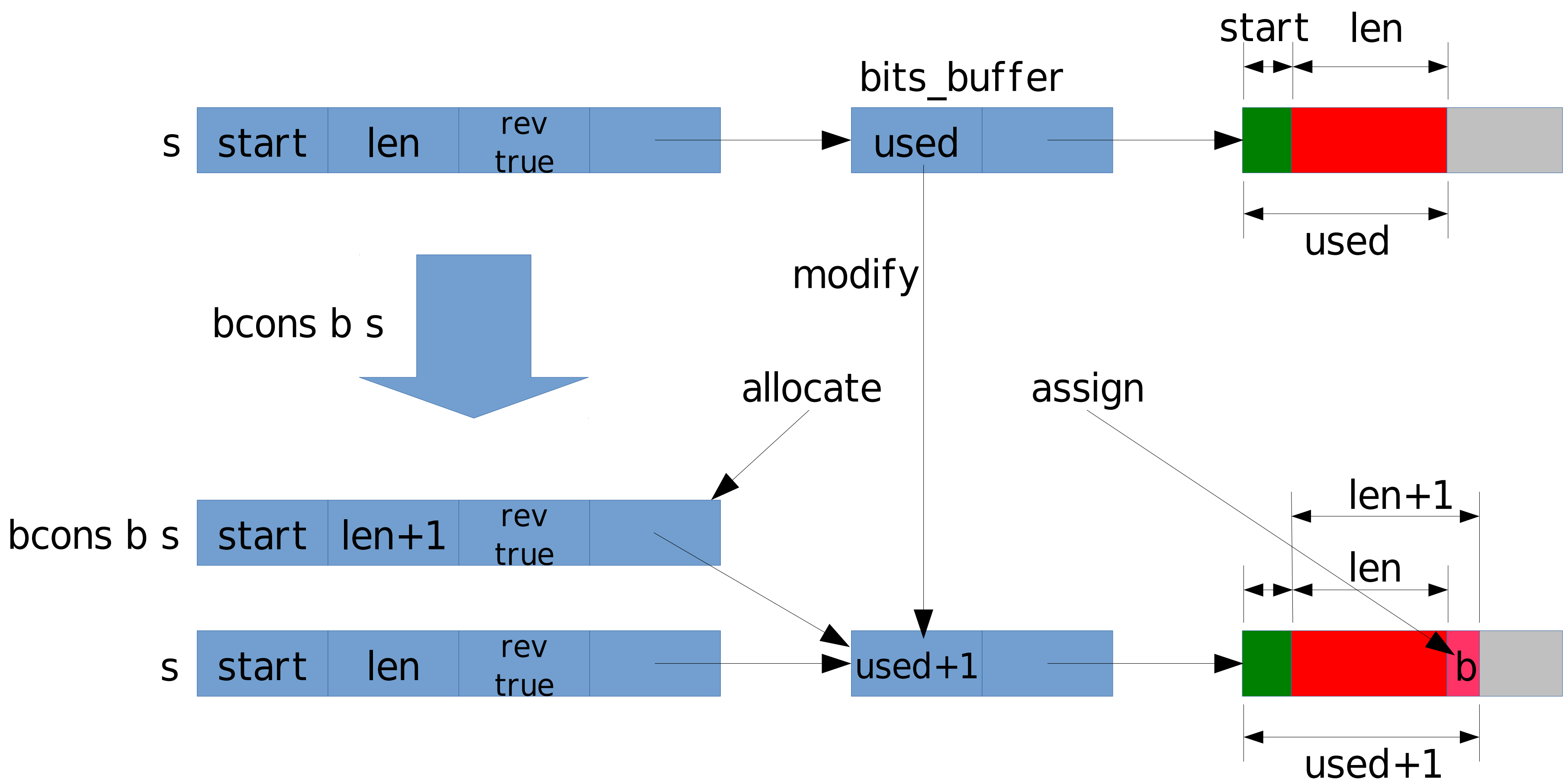
AIST

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

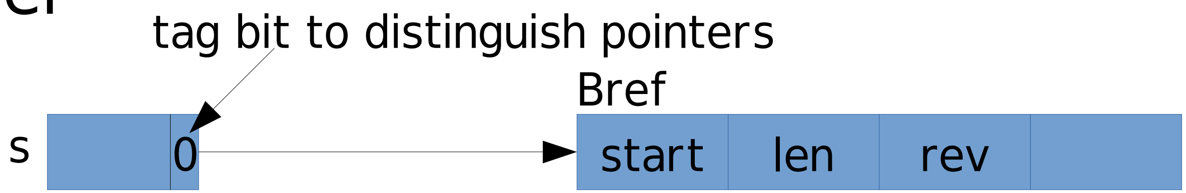
bcons Works in Constant Time

Constant time if rev=true and start+len=used

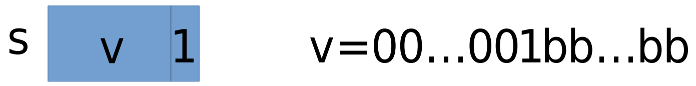


bnil and Short Bitstrings

- Non-constant constructor is implemented with a pointer



- bnil and short bitstrings 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 * int * bool * bits_buffer
```

Complexity of OCaml Bitstring Functions
Library Overview

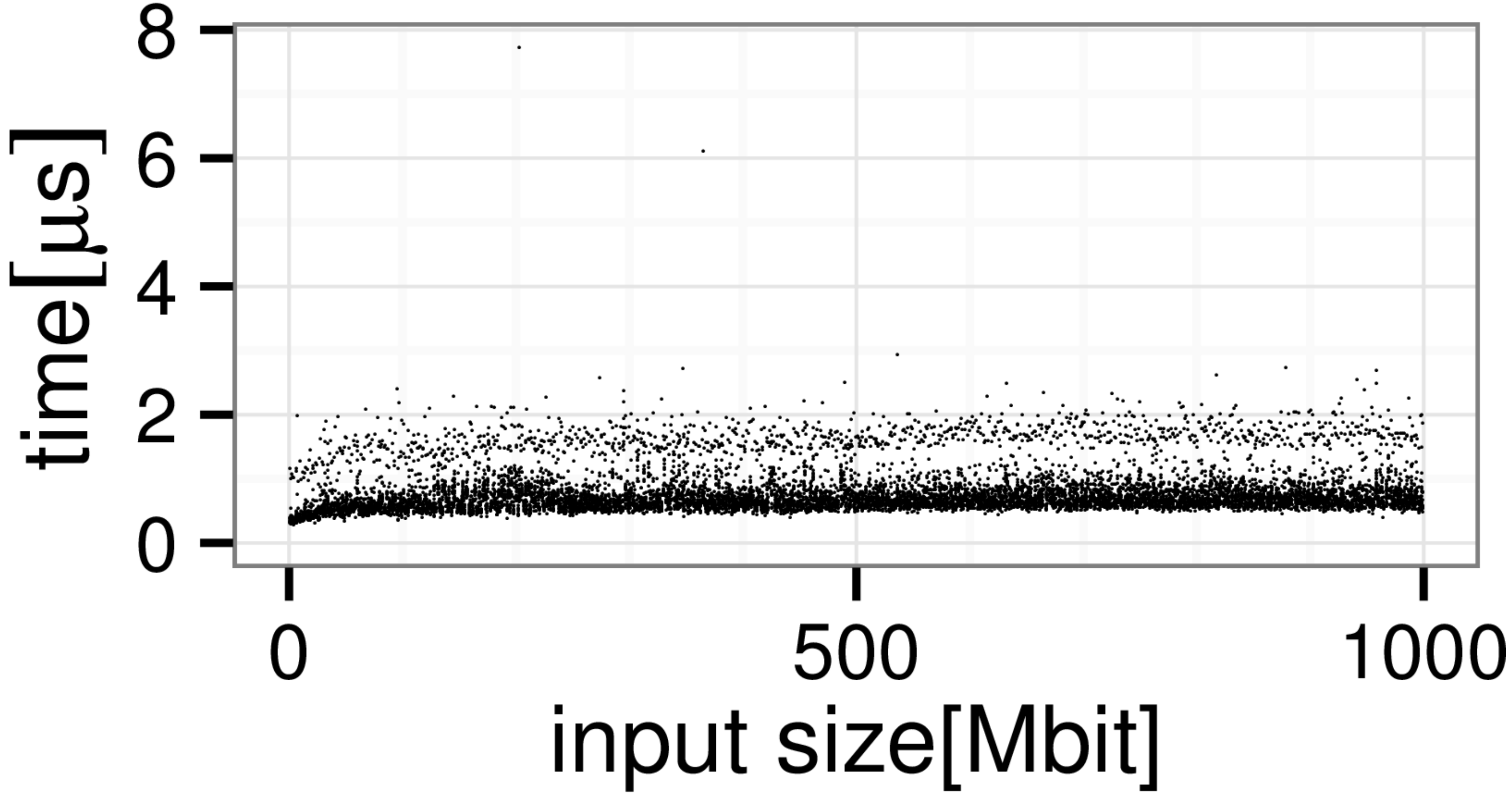
- Linear bitstring construction in linear time
  - bcons b (bcons b (… bnil …))
  - Always rev=true and start + len = used, bcons is O(1)
  - bits_buffer is doubled when bits_buffer is full
    Amortized copy cost doesn't increase complexity
- Reversing a bitstring in constant time
  - negate rev field by breverse
  - tail recursive program builds a list in reverse order in general
- Random access in constant time
  - random access in a bytes by bnth

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

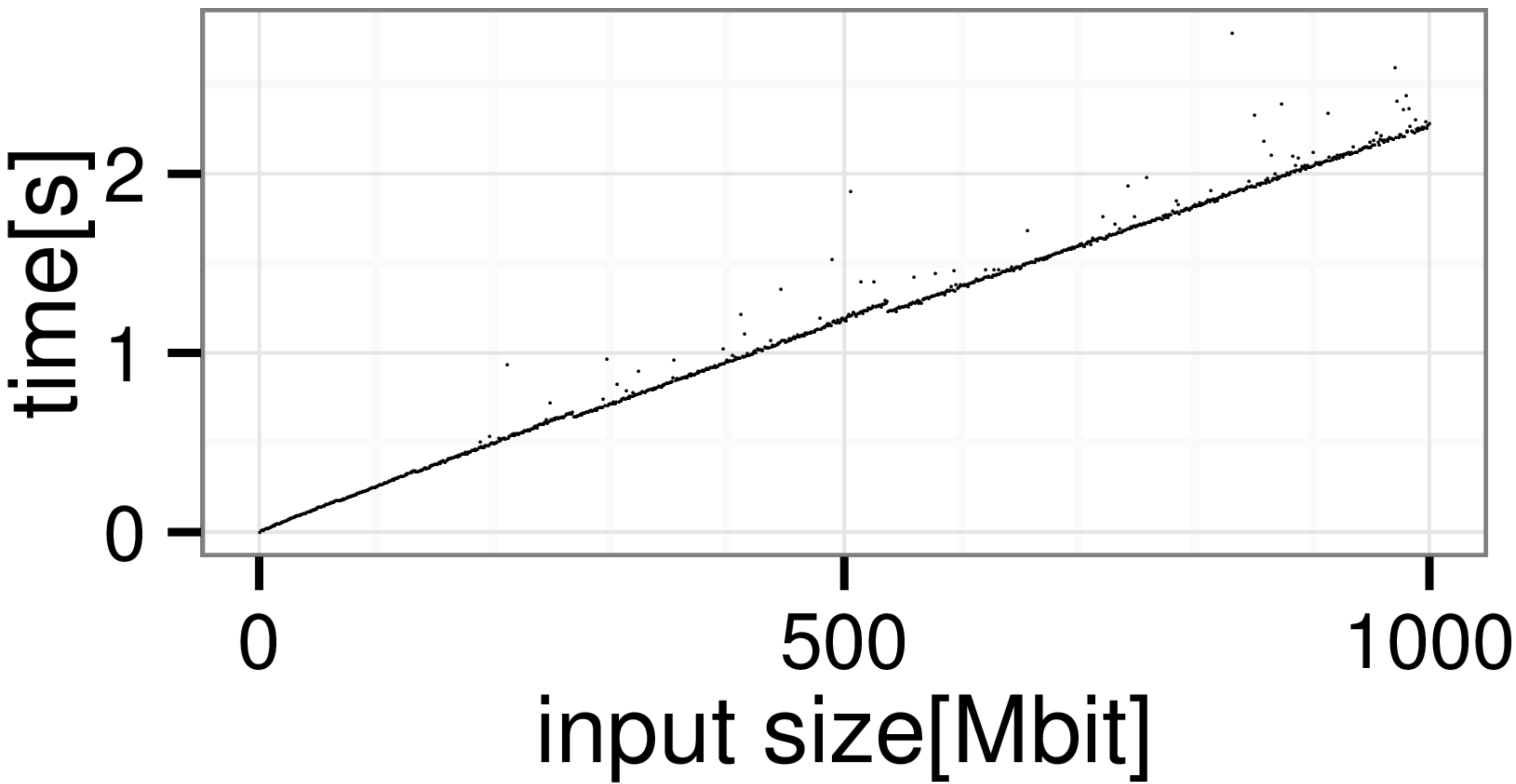
rank_lookup Benchmark

- Lookup seems O(1). Average 0.8[μs]
- Memory cache effect for small input



rank_init Benchmark

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



Outline

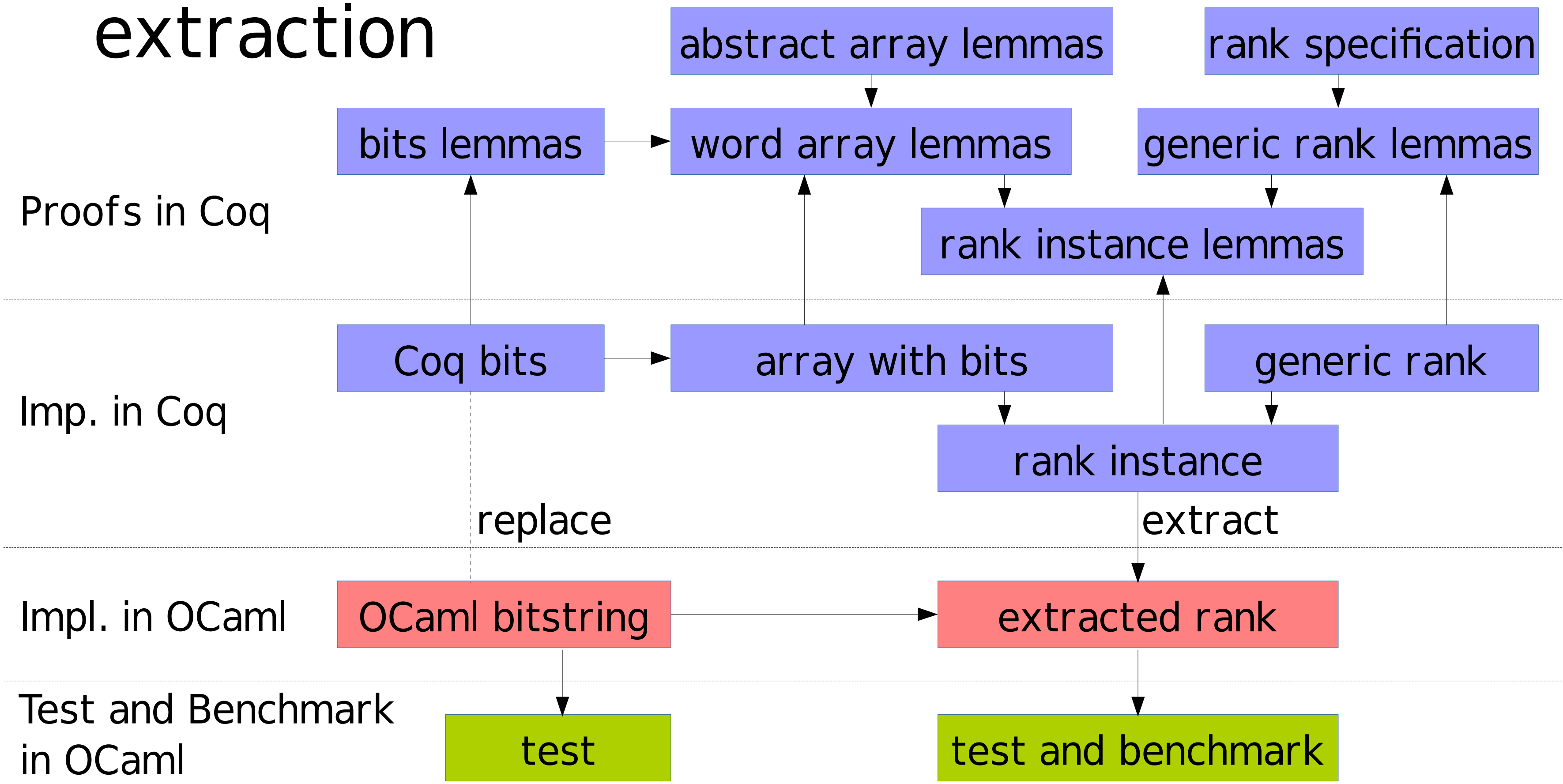
1. Background on Succinct Data Structures
2. Extraction of Coq lists to OCaml bitstrings
3. rank Formalization in Coq
4. Formal verification in Coq
5. OCaml bitstrings library
6. Benchmark
7. Modularized Proof
8. Conclusion

Array impl. using Bitstring

- Array construction and lookup functions
  Defined for dir1 and dir2
  - Definition empty_dir1 := bnil.
  - Definition push_dir1 w1 s n := wcons w1 (wbitrev w1 n) s.
  - Definition finalize_dir1 s := breverse s.
  - Definition lookup_dir1 w1 i s := wnth w1 i s.
- Utility functions
  - wbitrev w n returns lower n bits in reverse order
  - wcons w n s prepends lower w bits of n to s
  - 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

Extra Slides

Extracted rank_lookup

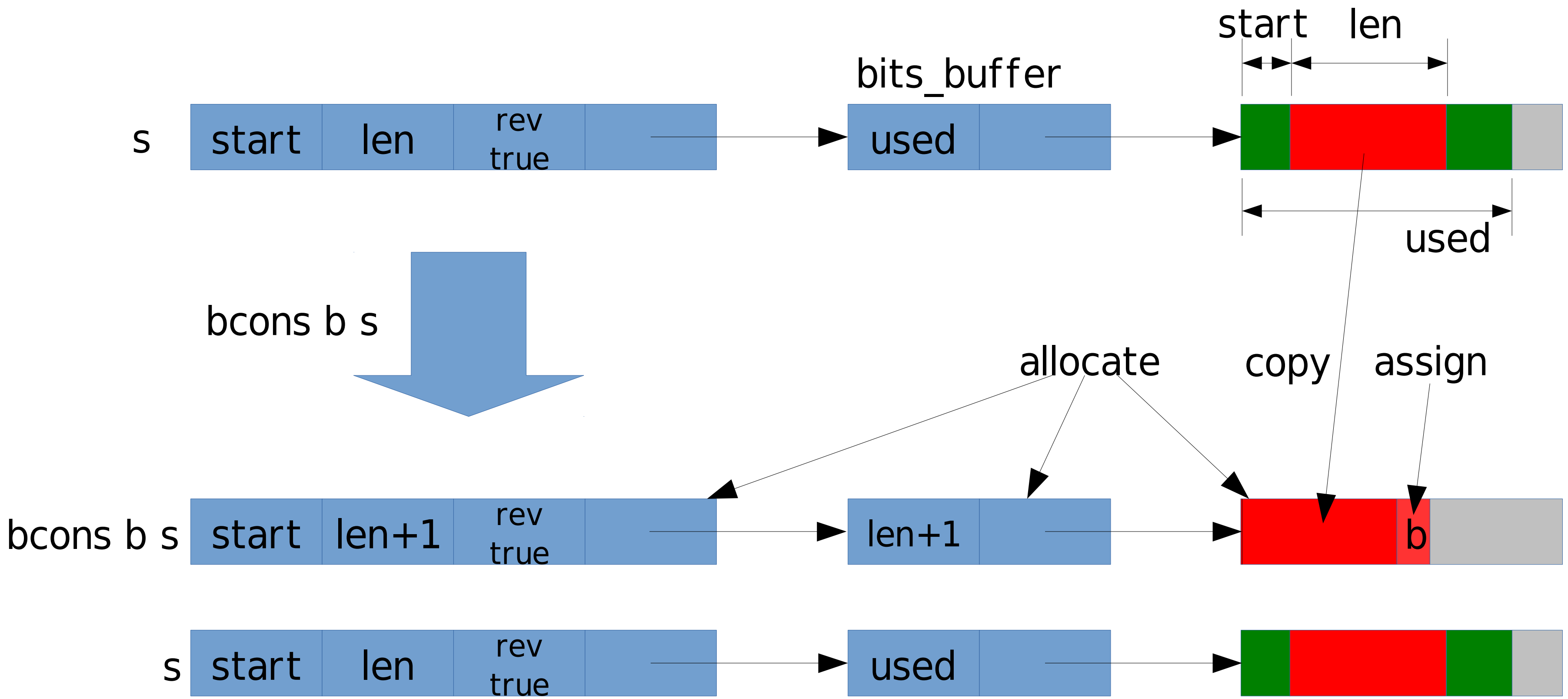
```
let rank_lookup aux0 i =
  let b = aux0.query_bit in
  let param0 = aux0.parameter in
  let w1 = param0.w1_of in
  let w2 = param0.w2_of in
  let dirpair = aux0.directories in
  let j2 = (/) i (Pervasives.succ param0.sz2p_of) in
  let j3 = (mod) i (Pervasives.succ param0.sz2p_of) in
  let j1 = (/)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

Extracted rank_init

```
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))) }
```

bcons Works in Linear Time

Copy bits_buffer if start+len != used



Functions Implemented in OCaml and C

- breverse s
  negate rev. O(1)
- bnth i s
  random access in a bytes. O(1)
- bappend s1 s2
  O(min(n1,n2)) for array construction
  O(n1+n2) in general (generalization of bcons)
- bcount b j i s
  bcount b j i s counts b in i bits from j'th bit in s
  bcount uses a gcc builtin, __builtin_popcountl (POPCNT)
  O(1) if i < machine word size
  O(i) in general