Generation of Code from Coq for Succinct Data Structures

Akira Tanaka*, Reynald Affeldt*, Jacques Garrigue**

*National Institute of Advanced Industrial Science and Technology (AIST)
**Nagoya University

Project Overview

- Goal: investigate the application of Coq to big-data
- ***Succinct data structures***?
  - Compact data structures
    - Storage requirements ⇒ complex data structures
    - Speed requirements ⇒ low-level implementation
  - Used in big-data analysis
    - Text indexing
      - E.g., Succinct Apache Spark is "2.5x lower storage, and over 75x faster than native Apache Spark" (http://succinct.cs.berkeley.edu/wp/wordpress/)
    - Bioinformatics

  ⇒ Target for formal verification
- This talk
  - An approach to formal verification of succinct data structures

Outline

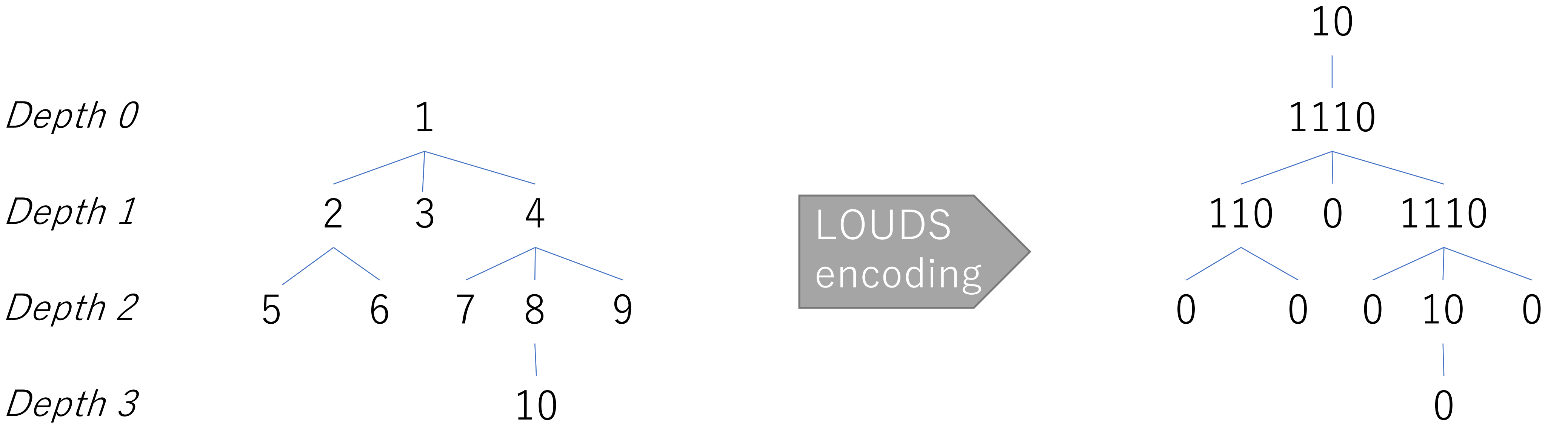
- Succinct Data Structures
- Generation of C Code
- Monadification
- Experiments
- Conclusion

Background about Succinct Data Structures

- Succinct vs. standard data structures
  - Use the least amount of memory possible
    - Sample idea: representing a tree of n nodes with n machine pointers can be wasteful
    - Take space close to the information-theoretic limit
  - Provide operations with the usual time-complexity

    $\Rightarrow$ tolerate an additional $o(n)$ storage
- Some references:
  - rank and select functions [Jacobson 1988, 1989] [Clark 1996]
  - LOUDS trees [Jacobson 1989]
  - FM-indexes (searchable compressed representation of text) [Ferragina, Manzini 2000]

Using Bitvectors to deal with Trees



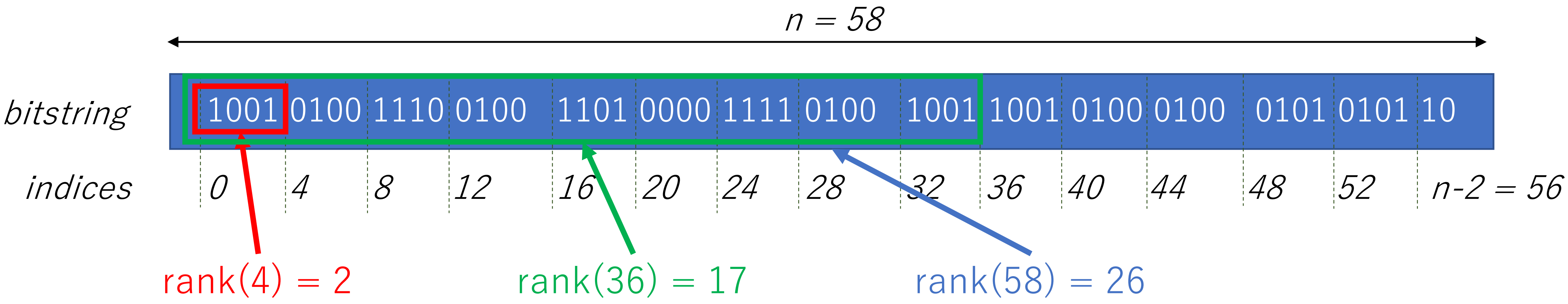
Depth 0	*Depth 1*	*Depth 2*	*Depth 3*
1	234	56789	10

	Depth 0	*Depth 1*	*Depth 2*	*Depth 3*
10	1110	11001110	000100	0

Bitvector to be turned into a succinct data structure…

The rank Algorithm
Specification

- Definition: Rank(n) = "How many 1's until n (exluded)?"



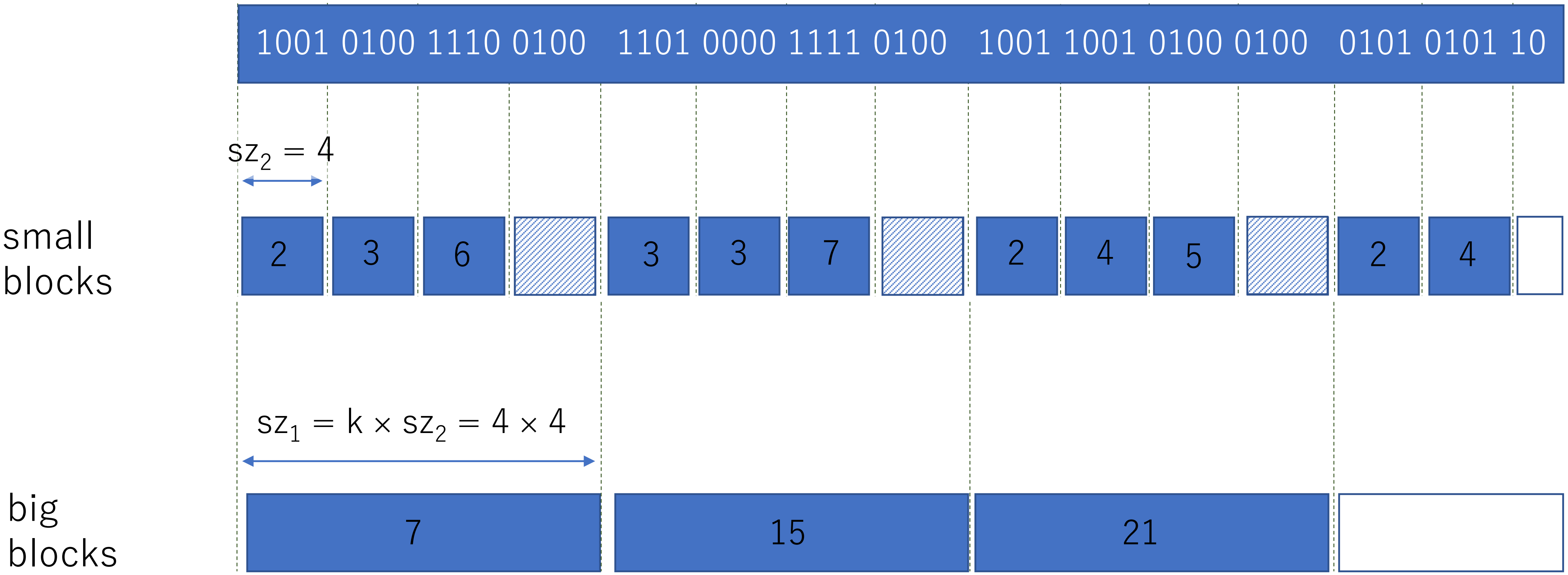
- Naïve definition:

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

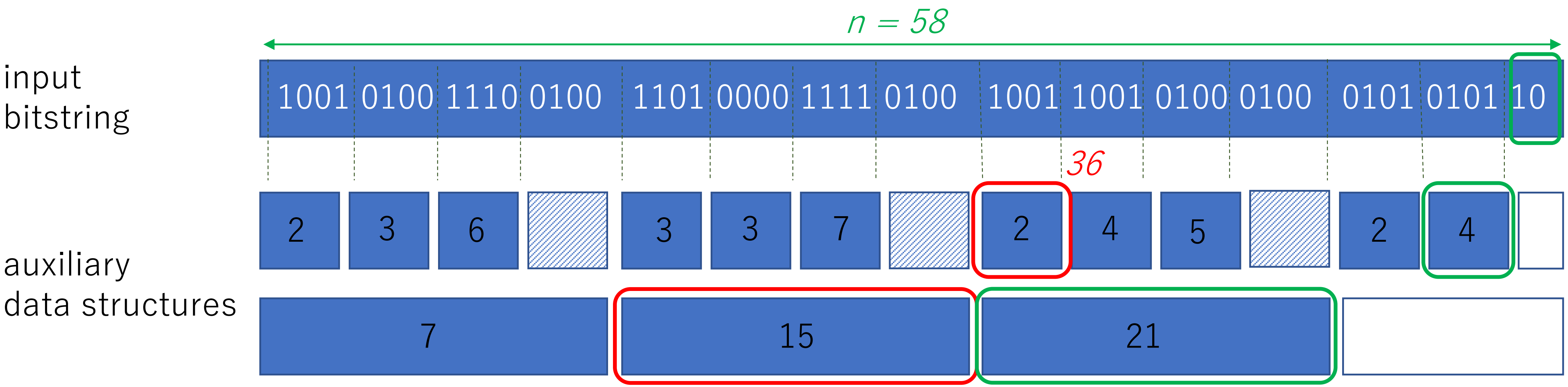
- Problem: it is linear-time…

The rank Algorithm

Auxiliary Data Structures



The rank Algorithm
In Constant-time



- Computation: $\mathrm{Rank}(i) = \frac{i}{sz_2}th$ small $+ \frac{i/sz_2}{k}th$ big + local popcount
- Examples:

rank(36) = 15 + 2 + 0 = 17

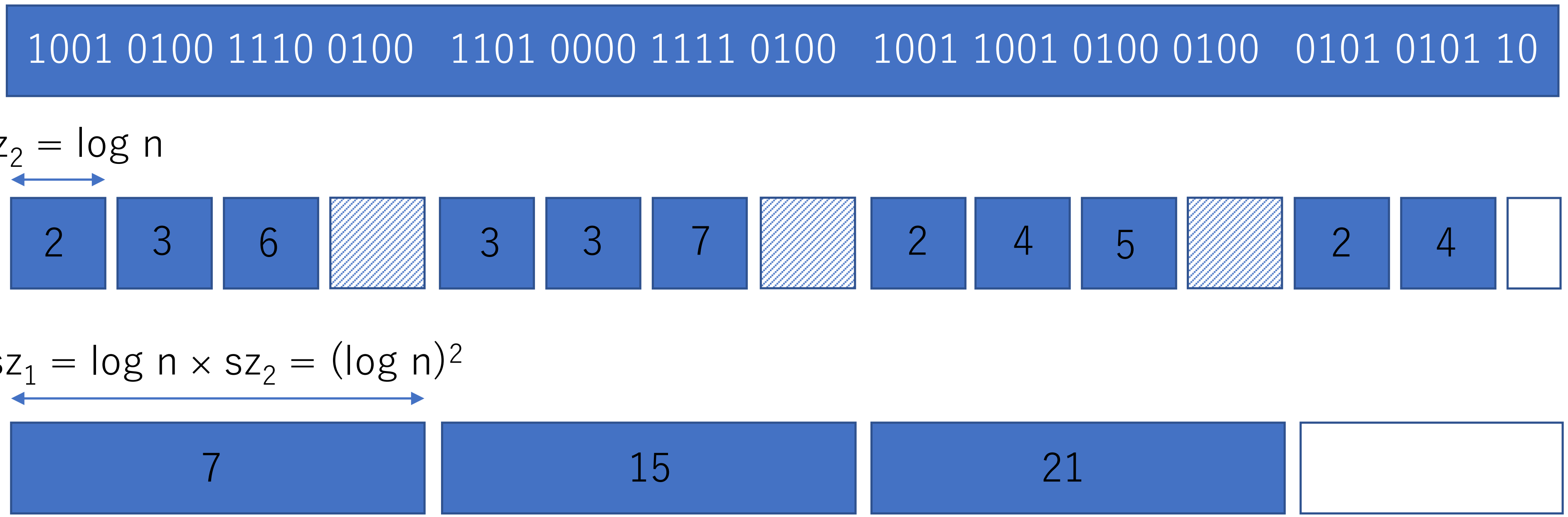
rank(58) = 21 + 4 + 1 = 26

implemented in constant-time

The rank Algorithm in Constant-time

Size of Auxiliary Data Structures

- Assumption: appropriate choice of block sizes



$o(n)$ bits

$$\frac{n}{\log n} \times \log(\log n)$$

$$\frac{n}{(\log n)^2} \times \log n$$

The rank Algorithm in Constant-time

Formal Correctness in Coq

- Implementation using two functions
  - rank_init computes the auxiliary data
  - rank_lookup computes rank
  - (We will come back to the implementation later)
- Functional correctness:

```
Lemma RankCorrect b s i : i ≤ bsize s →
  rank_lookup (rank_init b s) i = rank b i s.
```

Constant-time rank

Naïve implementation

The rank Algorithm in Constant-time

Size of Auxiliary Data Structures in CoQ

- Storage requirements (see [Tanaka et al., ICFEM 2016]):
  - Example: big blocks

```
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.
```

  - Turned into mathematical notations:

$$\left(\frac{\frac{n}{\lceil \log_2(n+1)\rceil+1}}{m+1}+1\right)p$$

with :
$m = \lceil \log_2(n+1)\rceil$
$p = \lceil \log_2(\frac{n}{m+1}\cdot(m+1)+1)\rceil$
where $\div$ is the Euclidean division

  - Asymptotically equal to $\frac{n}{\log n}$ though

$\Rightarrow$ We are lacking formal mathematical theories here (future work)

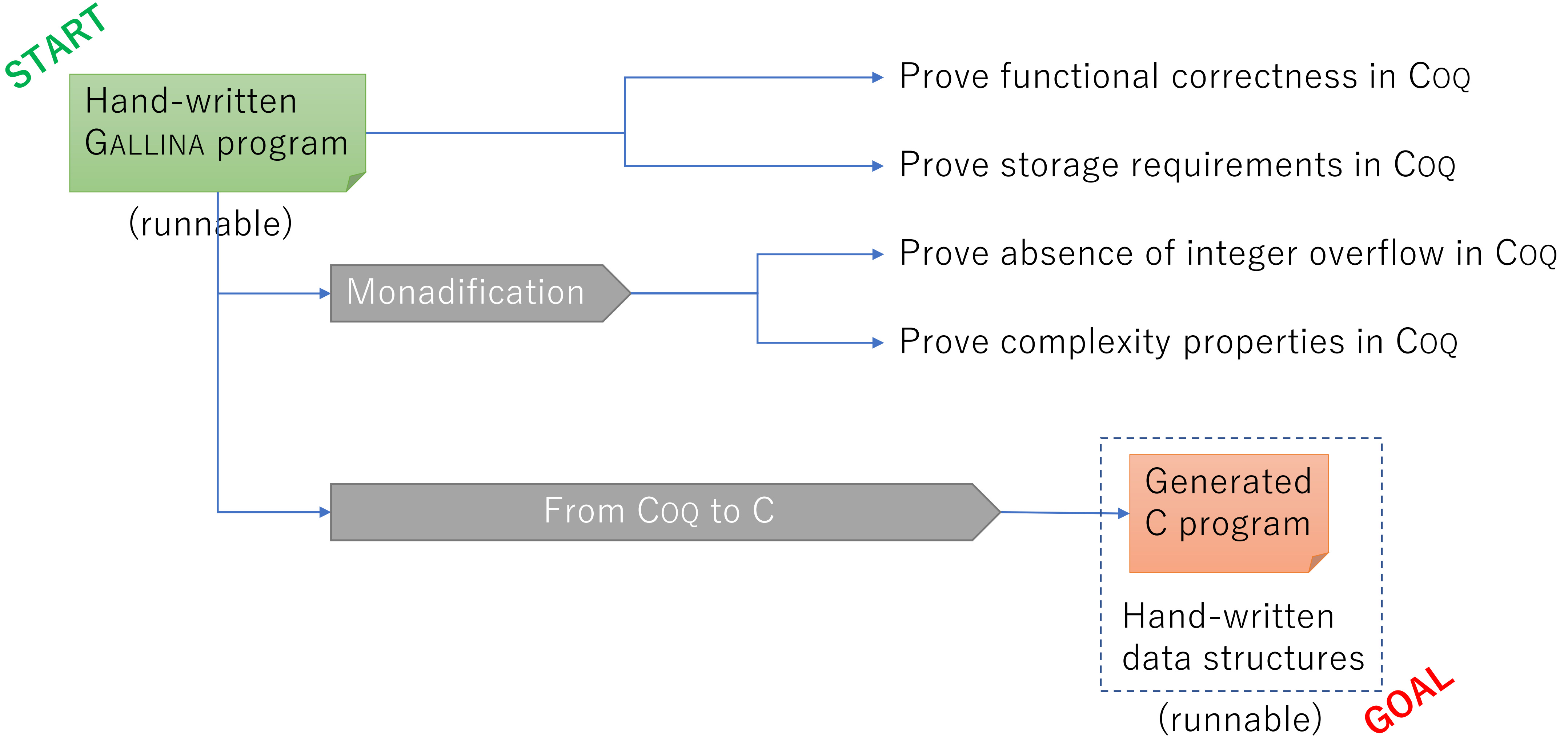
Formal Verification of rank using COQ

Approach

- Our approach: code generation from GALLINA
  - 1st experiment: Generate OCAML code and instrument [ICFEM 2016]
  - This talk: Generate C code and instrument [IPSJ PRO 2017/JIP2018]

- Other possible approaches:
  - Write pseudo-code and do a refinement?
    - Previous work: multi-precision arithmetic refined to assembly [Affeldt, ISSE 2013]
    - Can't GALLINA play the role of pseudo-code?
  - Write the code in C and verify it using Separation logic?
    - Previous work: TLS parsing function [Affeldt and Sakaguchi, JFR 2014]
    - It is costly…

Structure of Our Code Generation Scheme



Outline

- Succinct Data Structures
- Generation of C Code
- Monadification
- Experiments
- Conclusion

Extraction from Coq to C

- Main goal: output C code
  - We want the output to be readable
    - E.g., to accommodate
      - gcc builtins (e.g., __builtin_popcount)
      - SSE instructions
      - custom datatypes
  - We want the compiler to be robust
    - In particular to preserve tail-recursion
- We are not trying to build a full-fledged Coq compiler
  - ML-polymorphic subset of Gallina ought to be enough
  - We want the various parts to be loosely coupled

Compilation of Inductive Types in C

Constructors

Provided by the user

- Datatypes and constructors

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

```
#define nat uint64_t
#define n0_O() ((nat)0)
#define n1_S(n) ((n)+1)
```

Constructor **O** → From Coq to C → *Function call* n0_O()

Constructor **S** ***e*** → From Coq to C → *Function call* n1_S(***e***)

- Basic operations are mapped to their native counterparts
  - For example:

```
Definition addn := ⋯
```

```
#define n2_addn(a,b) ((a)+(b))
```

Compilation of Inductive Types in C
Destruction

- match expressions are turned into switch statements

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

[NB: coming from the previous slide]

```
#define nat uint64_t
#define n0_O() ((nat)0)
#define n1_S(n) ((n)+1)
```

For construction

```
match a with
| 0 ⇒ ⋯
| S n ⇒ (* some use of
          the variable n *)
end
```

From Coq to C

```
switch ((v27_a)) {
  case 0: { ⋯; }
  default: {
   nat v29_n = ((v27_a)-1);
   /* some use of the variable v29_n */
  }}
```

```
#define sw_nat(n) (n)
#define case_O_nat case 0
#define case_S_nat default
#define field0_S_nat(n) ((n)-1)
```

For destruction

Provided by the user

Compilation of Recursive Functions to C
Not Tail-recursive

- Map Coq functions to C functions (with return's)

```
Fixpoint add1 a b :=
  match a with
  | 0 ⇒ b
  | S n ⇒ S (add1 n b)
  end.
```

From Coq to C

```
nat
n2_add1(nat v27_a, nat v26_b)
{
  switch (sw_nat(v27_a))
  {
    case_O_nat: { return v26_b; }
    case_S_nat: {
      nat v29_n = field0_S_nat(v27_a);
      nat v30_n = n2_add1(v29_n, v26_b);
      return n1_S(v30_n);
    }
  }
}
```

Compilation of Recursive Functions to C

Tail-recursive

- Use *label*s and gotos

```
Fixpoint add2 a b :=
  match a with
  | 0 ⇒ b
  | S n ⇒ add2 n (S b)
  end.
```

From Coq to C

```
nat
n2_add2(nat v32_a, nat v31_b)
{
 n2_add2:;
  switch (sw_nat(v32_a))
  {
   case_O_nat: { return v31_b; }
   case_S_nat: {
    nat v34_n = field0_S_nat(v32_a);
    nat v35_n = n1_S(v31_b);
    v32_a = v34_n;
    v31_b = v35_n;
    goto n2_add2;
   }
  }
}
```

Assignments to local variables

Preprocessing
Monomorphization

- Basic idea: get rid of polymorphism that C does not enjoy
  - Example: pairs of Booleans

(Polymorphic) definitions:

```
Inductive prod (A B :Type) : Type :=
  pair : A → B → prod A B.

Definition swap (A B) (p : A * B) :=
let (a, b) := p in (b, a).
```

(Monomorphic) definitions:

```
Definition _pair_bool_bool :=
@pair bool bool : bool → bool → bool * bool

Definition _swap_bool_bool (p : bool * bool) :=
let (a, b) := p in _pair_bool_bool b a.
```

Monomorph.

Application point:

```
Definition swap_bb p := @swap bool bool p.
```

→

```
Definition _swap_bb p := _swap_bool_bool p.
```

Example: Pairs of Booleans (cont'd)

From Preprocessing to C

(Monomorphic) definitions:

[NB: coming from the previous slide]

```
Definition _pair_bool_bool :=
@pair bool bool : bool → bool → bool * bool

Definition _swap_bool_bool (p : bool * bool) :=
let (a, b) := p in _pair_bool_bool b a.

Definition _swap_bb p := _swap_bool_bool p.
```

GenC

```
prod_bool_bool
n1_swap_bool_bool(prod_bool_bool v0_p)
{
  bool v1_a = field0_pair_prod_bool_bool(v0_p);
  bool v2_b = field1_pair_prod_bool_bool(v0_p);
  return n2_pair_bool_bool(v2_b, v1_a);
}

prod_bool_bool
n1_swap_bb(prod_bool_bool v3_p)
{
  return n1_swap_bool_bool(v3_p);
}
```

pair of Booleans implemented as an int (provided by the user)

```
#define prod_bool_bool int
#define field0_pair_prod_bool_bool(v) ((v) & 1)
#define field1_pair_prod_bool_bool(v) (((v) & 2) >> 1)
#define n2_pair_bool_bool(x, y) ((x) | ((y) << 1))
```

Extraction from Coq to C

Implementation

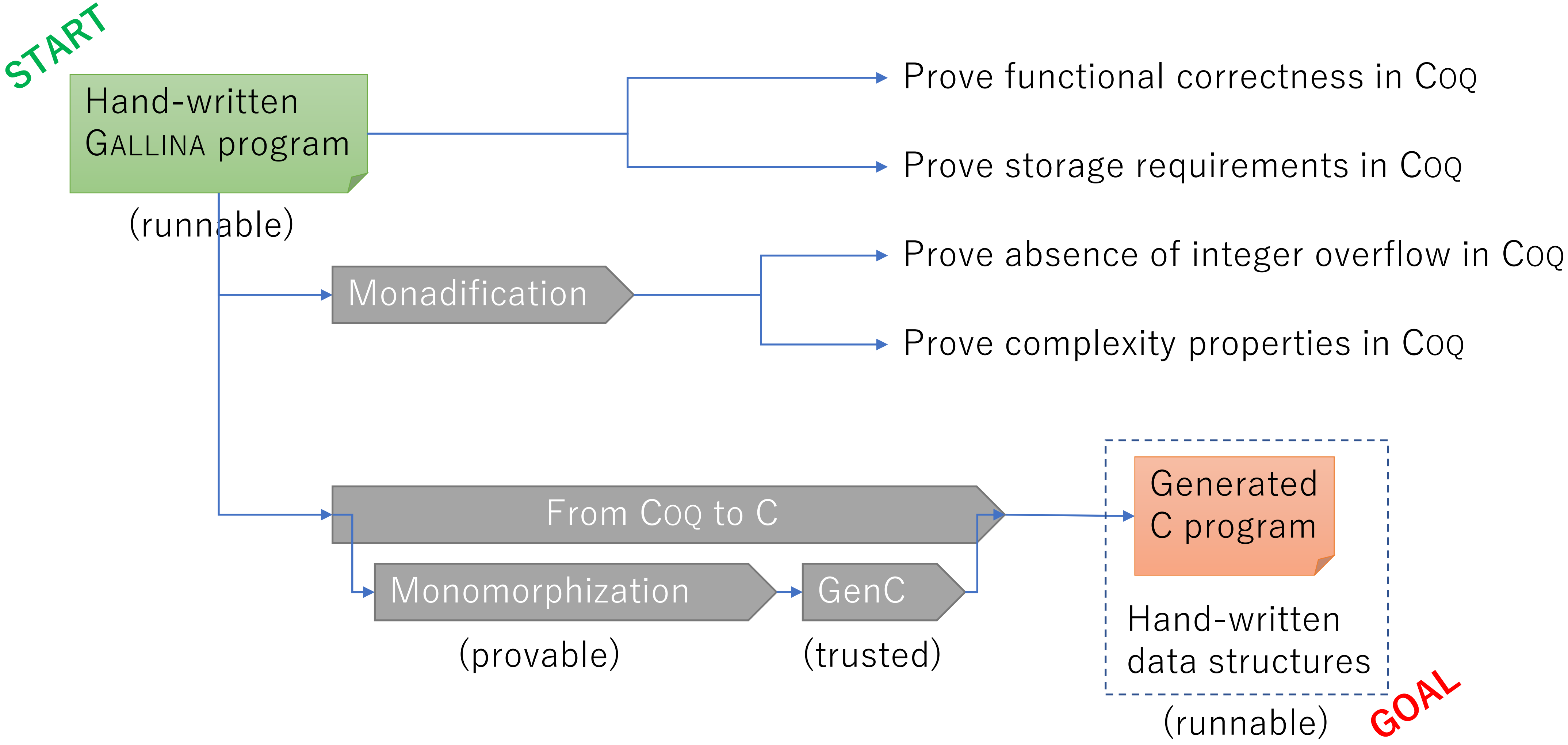
- Tactic Monomorphize
  - Monomorphize polymorphic constructs
  - Introduce let-expressions

    ⇒ Output a A-normal form
  - Provably correct thanks to Coq conversion rule
    - Monomorphization checked by β-reduction
    - let-expressions checked by ζ-reduction
  - Can control the processing of selected functions
    - Functions that contain opaque constructs
    - Functions to be mapped to primitive operations
- Tactic GenC
  - Destruction of inductive types into switch's
  - Tail-recursion compiled to goto's
  - (Formal verification deferred to future work)

Filename	l.o.c.	Contents
g_monomorph.ml4	30	Register commands
monoutil.ml	136	Utilities
monomorph.ml	710	Monomorphization
genc.ml	696	C code generator

https://github.com/akr/codegen

Structure of Our Code Generation Scheme



Outline

- Succinct Data Structures
- Generation of C Code
- Monadification
- Experiments
- Conclusion

Why Monadification?

- A source of concern:
  - The ***change of data structures*** can cause unexpected behaviors
  - E.g.: overflows because Peano integers are replaced by machine integers
- Solution: insert checks at appropriate locations
  - To prove that C invariants are enforced
- Problems:
  - Manual insertion of checks is error-prone, overkill
  - We do not want to pollute the generated C code
  - We do not want to make the proofs in COQ more difficult
- Solution: ***automatic monadification***
  - See a GALLINA term as a program that can have "effects"
    - "effects" being encapsulated in a monad
  - There are in fact many "effects" that can be checked with monads
    - Overflows, division by zero, out-of-bounds accesses
    - Complexity (e.g., number of invocation of the "cons" constructor)

Monadification

Preparatory Steps

- Define a monad. For example, the Maybe monad:

```
Definition ret {A} (x : A) := Some x.
Definition bind {A} {B} (x' : option A) (f : A → option B) :=
  match x' with
  | None ⇒ None
  | Some x ⇒ f x
  end. (* Notation: >>= *)
```

- Define monadic operations. For example:

```
Definition W := 64.
Definition check x := if log2 x < W then Some x else None.
Definition SM a := check a.+1.
Definition addM a b := check (a + b).
```

Monadification

Practical Example using Our Tactics

- Register the monad and its operations:

```
Monadify Type option.
Monadify Return @ret.
Monadify Bind @bind.

Monadify Action S ⇒ SM.
Monadify Action muln ⇒ mulM.
```

- Call the monadification tactic:

```
Fixpoint pow a k :=
 match k with
 | 0 ⇒ 1
 | k'.+1 ⇒ a * pow a k'
 end.
```

Monadification

```
powM =
fix pow (a k : nat) {struct k} : option nat :=
 match k with
 | 0 ⇒ SM 0
 | k'.+1 ⇒ pow a k' >>= mulM a
 end.
```

Monadification

Implementation

- Tactic Monadification
  - Basic idea: $f : t_0 \to t_1 \to t_2 \to t_3$
    - $f' : M\ (t_0 \to M\ (t_1 \to M(t_2 \to M\ t_3)))$
    - $f' : t_0 \to t_1 \to t_2 \to M\ t_3$
  - Monadic operations defined by the user
    - Monadic Action f ⇒ fM.
  - Can control the processing of selected functions
    - E.g., functions that maps to C primitives
- No formal proof. Yet:
  - The result goes through the COQ type-checker
  - Application to the identity monad

Filename	l.o.c.	Contents
g_monadification.ml4	40	Register commands
Monadification.ml	924	Monadification

https://github.com/akr/monadification

Use Monadification to Prove Properties of Programs

- Prove that the program does not fail as follows:

$$\forall\ x,\ \textit{condition} \rightarrow \text{fM } x = \text{Some } (\text{f } x).$$

Result of monadification   Original GALLINA function

- Example:

```
Theorem powM_ok :
  forall a b, Nat.log2 (pow a b) < 32 →
   powM a b = Some (pow a b).
```

Monadification

Establish Complexity Properties

- Count the number of explicit "cons" invocations:

```
Definition counter_with A : Type := nat * A.
Definition ret {A} (x : A) := (0, x).
Definition bind {A} {B} (x' : counter_with A) (f : A -> counter_with B) :=
  let (m, x) := x' in
  let (n, y) := f x in
  (m+n, y).
```

- Example: compare textbook append and its tail-recursive version

```
Fixpoint rev (l : list A) : list A :=
  match l with
   | [] => []
   | x :: l' => rev l' ++ [x]
  end.
```

$\frac{n(n+1)}{2}$ invocations of "cons"

```
Fixpoint rev_append (l l' : list A) : list A :=
  match l with
   | [] => l'
   | a :: l => rev_append l (a :: l')
  end.
```

n invocations of "cons"

Outline

- Succinct Data Structures
- Generation of C Code
- Monadification
- Experiments
- Conclusion

Generation of C Code for the rank Algorithm

- Reminder: implementation of the rank algorithm
  - rank_init computes the auxiliary data
  - rank_lookup computes rank
- Let us just look at the function buildDir2
  - the construction of the small blocks

Monomorphization

- Instantiate polymorphic functions and insert let-expressions

list of bools

list of small naturals

```
Fixpoint buildDir2 b s sz2 c i D2 m2 :=
 if c is cp.+1 then
  let m := bcount b i sz2 s in
  buildDir2 b s sz2
   cp (i + sz2) (pushD D2 m2) (m2 + m)
 else
  (D2, m2).
```

Monomorph.

```
_buildDir2 =
fix buildDir2 (b : bool) (s : bits) (sz2 c i : nat) (D2 : DArr)
(m2 : nat) {struct c} :
DArr * nat :=
 match c with
 | 0 ⇒ _pair_DArr_nat D2 m2
 | cp.+1 ⇒
   let m := _bcount b i sz2 s in
   let n := _addn i sz2 in
   let d := _pushD D2 m2 in
   let n0 := _addn m2 m in
   buildDir2 b s sz2 cp n d n0
 end
```

Generation of C code

[NB: coming from previous slide]

```
_buildDir2 =
fix buildDir2 (b : bool) (s : bits) (sz2 c i : nat)
  (D2 : DArr) (m2 : nat) {struct c} : DArr * nat :=
  match c with
  | 0 ⇒ _pair_DArr_nat D2 m2
  | cp.+1 ⇒
     let m := _bcount b i sz2 s in
     let n := _addn i sz2 in
     let d := _pushD D2 m2 in
     let n0 := _addn m2 m in
     buildDir2 b s sz2 cp n d n0
  end
```

GenC

```
prod_DArr_nat
n7_buildDir2(bool v10_b,
             bits v9_s,
             nat v8_sz2,
             nat v7_c,
             nat v6_i,
             DArr v5_D2,
             nat v4_m2) {
 n7_buildDir2:;
 switch (sw_nat(v7_c)) {
  case_O_nat: { return n2_pair_DArr_nat(v5_D2, v4_m2); }
  case_S_nat: {
   nat v12_cp = field0_S_nat(v7_c);
   nat v13_m = n4_bcount(v10_b, v6_i, v8_sz2, v9_s);
   nat v14_n = n2_addn(v6_i, v8_sz2);
   DArr v15_d = n2_pushD(v5_D2, v4_m2);
   nat v16_n = n2_addn(v4_m2, v13_m);
   v7_c = v12_cp;
   v6_i = v14_n;
   v5_D2 = v15_d;
   v4_m2 = v16_n;
   goto n7_buildDir2;
  }
 }
}
```

bitstring (?)

array of small integers (?)

User-provided Datatypes

- Lists of bools implemented as a array of bits:

```
typedef struct {
  uint64_t *buf;
  nat len;
  nat max;
} bits_heap;
```

```
typedef struct {
  bits_heap *heap;
  nat len;
} bits;
```

- Array of natural smaller than 2^w implemented as a bistring:

```
typedef struct {
  nat w;
  bits s;
} DArr;
```

Primitives

- Length in bits of naturals

```
static inline nat
n1_bitlen(nat n)
{
  if (n == 0) return 0;
  assert(64 <= sizeof(long) * CHAR_BIT);
  return 64 - __builtin_clzl(n);
}
```

Monadification of the rank Algorithm

- Various checks besides arithmetic. For example:

```
Definition pushDM D v := let: darr w d := D in
  if v < 2 ^ w then Some (pushD D v) else None.
Monadify Action pushD ⇒ pushDM.

Definition lookupDM D i :=
  if i < sizeD D then check (lookupD D i) else None.
Monadify Action lookupD ⇒ lookupDM.
```

- “Absence of failures” lemma:

```
Lemma RankSuccess b s i :
  let n := bsize s in
  log2 n < W → i ≤ n →
  (rank_initM b s >>= fun aux ⇒ rank_lookupM aux i) =
  Some (rank_lookup (rank_init b s) i).
```

Other Experiments

- Check the complexity of the rank function using monadification
  - By looking at the number of bits inspected by bcount ($\approx$ population count)
- Monadification of seq.v (SSREFLECT library of lists)
  - To test robustness
- HTML escape [Tanaka, TPP2017]
  - A use-case for SSE instructions

Outline

- Succinct Data Structures
- Generation of C Code
- Monadification
- Experiments
- Conclusion

Related Work

- Many (minor?) extraction facilities
  - Scala, SML, Erlang, Rust, Ruby, Python
- Digger (previously coq2c) [Oudjail and Hym, 2017]
  - In Haskell, using JSON
- GALLINA compilers
  - CertiCoq [Anand et al., CoqPL 2017]
    - Full-fledged compiler
  - Œuf [Mullen et al., CPP 2018]
    - Target the ML subset with recursion using eliminators
    - Correctness by translation validation
    - Peano integers, etc. compiled as they are
- Monadification
  - [Hatcliff and Danvy, 1994] Many binds, failure to type-check/detect termination in COQ
  - [Erwig and Ren, 2004] Relies on exceptions

Summary and Future Work

- Application of Coq to ***succinct data structures***
  - Formal verification of the constant-time rank algorithm
- ***Generation of C code*** using Coq plugins:
  - Monomorphization
  - GenC
    - robust (tail-recursion preserved)
    - flexible (user-defined datatypes and primitives)
    - trusted but less than 1000 l.o.c. of OCaml
  - Monadification
    - to ensure safety of user-defined datatypes
    - to establish complexity properties
- Work in progress:
  - The constant-time select algorithm
  - A formal theory of succinct data structures
  - A formal theory of compression
    - on top of https://github.com/affeldt-aist/infotheo