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/](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

LOUDS encoding

Bitvector to be turned into a succinct data structure…
The rank Algorithm Specification

• Definition: \( \text{Rank}(n) = \text{“How many 1’s until } n \text{ (excluded)?”} \)

\[ n = 58 \]

- rank(4) = 2
- rank(36) = 17
- rank(58) = 26

• Naïve definition:
  
  Definition \( \text{rank} \) \( b_i s := \text{count_mem} \) \( b \) (take \( i \)s).

• Problem: it is linear-time…
The rank Algorithm

Auxiliary Data Structures

$sz_2 = 4$

small blocks

$sz_1 = k \times sz_2 = 4 \times 4$

big blocks

$sz_2 = 4$

1001 0100 1110 0100 1101 0000 1111 0100 1001 1001 0100 0100 0101 0101 10
The rank Algorithm

*In Constant-time*

**Computation:** \( \text{Rank}(i) = \frac{i}{sz_2} \text{th small} + \frac{i/sz_2}{k} \text{th big} + \text{local popcount} \)

**Examples:**

- \( \text{rank}(36) = 15 + 2 + 0 = 17 \)
- \( \text{rank}(58) = 21 + 4 + 1 = 26 \)
The rank Algorithm in Constant-time

Size of Auxiliary Data Structures

• Assumption: appropriate choice of block sizes

\[
\begin{align*}
sz_2 &= \log n \\
\frac{n}{\log n} \times \log(\log n) \\
\frac{n}{(\log n)^2} \times \log n \\
\end{align*}
\]
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* \( b \ s \ i : i \leq \text{bsize} \ s \rightarrow \)
  \[
  \text{rank_lookup \ (rank_init \ b \ s) \ i} = \text{rank \ b \ i \ s}.
  \]
The rank Algorithm in Constant-time

Size of Auxiliary Data Structures in CoQ

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

\begin{verbatim}
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.
\end{verbatim}

• Turned into mathematical notations:

\[
\binom{n}{\left\lfloor \log_2(n+1) \right\rfloor + 1} + 1 \quad p
\]

with:

\[
m = \left\lfloor \log_2(n+1) \right\rfloor
\]

\[
p = \left\lceil \log_2\left(\frac{n}{m+1} \cdot (m+1) + 1\right) \right\rceil
\]

where \( \div \) is the Euclidean division

• Asymptotically equal to \( \frac{n}{\log n} \) though

⇒ 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

Hand-written Gallina program

Monadification

From CoQ to C

Generated C program

Hand-written data structures (runnable)

Prove functional correctness in CoQ

Prove storage requirements in CoQ

Prove absence of integer overflow in CoQ

Prove complexity properties in CoQ
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

• Datatypes and constructors

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

  Constructor O
  Constructor S e

  Provided by the user

  Function call n0_O()
  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

```c
# 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**

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

**For construction**

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

---

Inductive `nat` : Set :=

- `O : nat`
- `S : nat → nat.`

[NB: coming from the previous slide]

**From Coq to C**

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

```c
switch ((v27_a)) {
    case 0: { ⋮; }
    default: {
        nat v29_n = ((v27_a)-1);
        /* some use of the variable v29_n */
    }
}
```
Compilation of Recursive Functions to C

Not Tail-recursive

• Map Coq functions to C functions (with return’s)

Fixpoint \texttt{add1} \ a \ b :=
match \ a \ with
| 0 \Rightarrow \ b
| S \ n \Rightarrow S (\texttt{add1} \ n \ b)
end.

\texttt{n2_add1}(\texttt{nat v27_a}, \texttt{nat v26_b})
{
switch (sw_nat(\texttt{v27_a}))
{
case \_O\_nat: \{ return \texttt{v26_b}; \}
case \_S\_nat: \{
\texttt{nat v29_n} = \texttt{field0\_S\_nat(v27_a)};
\texttt{nat v30_n} = \texttt{n2_add1(v29_n, v26_b)};
return \texttt{n1\_S(v30_n)};
\}
}\}
Compilation of Recursive Functions to C

*Tail-recursive*

- Use *labels* and *gotos*

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

```
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;
    }
  }
}
```

From Coq to C

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.

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

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.

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

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);
}

#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 \( \Lambda \)-normal form
  • Provably correct thanks to COQ conversion rule
    • Monomorphization checked by \( \beta \)-reduction
    • let-expressions checked by \( \zeta \)-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)

<table>
<thead>
<tr>
<th>Filename</th>
<th>l.o.c.</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_monomorph.ml4</td>
<td>30</td>
<td>Register commands</td>
</tr>
<tr>
<td>monoutil.ml</td>
<td>136</td>
<td>Utilities</td>
</tr>
<tr>
<td>monomorph.ml</td>
<td>710</td>
<td>Monomorphization</td>
</tr>
<tr>
<td>genc.ml</td>
<td>696</td>
<td>C code generator</td>
</tr>
</tbody>
</table>

https://github.com/akr/codegen
Structure of Our Code Generation Scheme

- Hand-written **GALLINA** program
  - (runnable)

- Monadification
  - Prove functional correctness in CoQ
  - Prove storage requirements in CoQ
  - Prove absence of integer overflow in CoQ
  - Prove complexity properties in CoQ

- From CoQ to C
  - Monomorphization (provable)
  - GenC (trusted)

- Generated **C** program
  - Hand-written data structures
    - (runnable)
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 \( S \Rightarrow SM \).
  Monadify Action \( \text{muln} \Rightarrow \text{mulM} \).

- Call the monadification tactic:

  - Fixpoint \( \text{pow} \ a \ k := \)
  
    match \( k \) with
    
    | 0 \( \Rightarrow 1 \)
    | \( k'.+1 \) \( \Rightarrow a \ast \text{pow} \ a \ k' \)
    
  end.

  - \( \text{powM} = \)

    fix \( \text{pow} \ (a \ k : \text{nat}) \ {\text{struct k}} : \text{option nat} := \)
    
    match \( k \) with
    
    | 0 \( \Rightarrow SM 0 \)
    | \( k'.+1 \) \( \Rightarrow \text{pow} \ a \ k' >\gg = \text{mulM} a \)
    
  end.
Monadification

Implementation

• **Tactic Monadification**
  • Basic idea: $f : t_0 \to t_1 \to t_2 \to t_3$
    $f' : M (t_0 \to M (t_1 \to M(t_2 \to M t_3)))$
  • Monadic operations defined by the user
    • **Monadic Action** $f \Rightarrow fM$.
  • 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

<table>
<thead>
<tr>
<th>Filename</th>
<th>L.o.c.</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>g Monadification.ml</td>
<td>40</td>
<td>Register commands</td>
</tr>
<tr>
<td>Monadification.ml</td>
<td>924</td>
<td>Monadification</td>
</tr>
</tbody>
</table>

https://github.com/akr/monadification
Use Monadification to Prove Properties of Programs

• Prove that the program does not fail as follows:

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

• Example:

Theorem powM_ok :
forall a b, Nat.log2 (pow a b) < 32 \rightarrow
powM a b = \text{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.

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

\(\frac{n(n+1)}{2}\) invocations of “cons”

\(n\) 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 \textit{let}-expressions

\[
\text{Fixpoint } \text{buildDir2} \ (b : \text{bool}) \ (s : \text{bits}) \ (sz2 \ c \ i : \text{nat}) \ (D2 : \text{DArr}) \ (m2 : \text{nat}) \ : \text{DArr} \times \text{nat} :=
\]

\[
\begin{align*}
\text{match } c & \text{ with} \\
| 0 & \Rightarrow \text{pair}_\text{DArr}_\text{nat} D2 \ m2 \\
| cp.+1 & \Rightarrow \begin{align*}
\text{let } m & := \text{bcount} b i sz2 s \text{ in} \\
\text{let } n & := \text{addn} i sz2 \text{ in} \\
\text{let } d & := \text{pushD} D2 m2 \text{ in} \\
\text{let } n0 & := \text{addn} m2 m \text{ in} \\
\text{buildDir2} b s sz2 cp n d n0
\end{align*}
\end{align*}
\]

\[
\text{Monomorph.}
\]

- list of bools
- list of small naturals

\[
\text{Monomorphization}
\]
Generation of C code

```
[GenC]

[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
```

```
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 _pair_DArr_nat(v5_D2, v4_m2);
    case_S_nat:
      let m = _bcount(v10_b, v6_i, v8_sz2, v9_s) in
      let n = _addn(v6_i, v8_sz2) in
      let d = _pushD(v5_D2, v4_m2) in
      let n0 = _addn(v4_m2, m) in
      buildDir2 v10_b v9_s v8_sz2 v7_c v6_i v5_D2 v4_m2;
  }
}
```

```
[nb: coming from previous slide]

_genC bitstring (?)
array of small integers (?)
```
User-provided Datatypes

• Lists of bools implemented as a array of bits:

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

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

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

Primitives

• Length in bits of naturals

```c
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:

\[
\text{Lemma RankSuccess } b \ s \ i : \quad \text{let } n := \text{bsize } s \text{ in } \\
\log_2 n < W \rightarrow i \leq n \rightarrow \\
(\text{rank_initM } b \ s \gg= \text{fun } \text{aux } \Rightarrow \text{rank_lookupM } \text{aux } i) = \\
\text{Some } (\text{rank_lookup } (\text{rank_init } b \ s) \ i).
\]

Definition \( \text{pushDM } D \ v := \text{let } \text{darr} w \ d := D \text{ in} \)
\[
\text{if } v < 2^w \text{ then Some } (\text{pushD } D \ v) \text{ else None.}
\]
Monadify Action \( \text{pushD } \Rightarrow \text{pushDM}. \)

Definition \( \text{lookupDM } D \ i := \)
\[
\text{if } i < \text{sizeD } D \text{ then check } (\text{lookupD } D \ i) \text{ else None.}
\]
Monadify Action \( \text{lookupD } \Rightarrow \text{lookupDM}. \)

• “Absence of failures” lemma:

Lemma RankSuccess \( b \ s \ i : \)
\[
\text{let } n := \text{bsize } s \text{ in} \\
\log_2 n < W \rightarrow i \leq n \rightarrow \\
(\text{rank_initM } b \ s \gg= \text{fun } \text{aux } \Rightarrow \text{rank_lookupM } \text{aux } i) = \\
\text{Some } (\text{rank_lookup } (\text{rank_init } b \ s) \ i).
\]
Other Experiments

• Check the complexity of the rank function using monadification
  • By looking at the number of bits inspected by bcount (≈ 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