Safe Low-level Code Generation in Coq using Monomorphization and Monadification

Akira Tanaka, AIST
Reynald Affeldt, AIST
Jacques Garrigue, Nagoya University

2017-06-09 IPSJ SIGPRO 114

Goal: Translate Coq to C

- Coq

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

- Easy proof

- C

```
int pow(int a, int k) {
  switch (k) {
    case 0: return 1;
    default: {
      int k_ = k-1;
      return
        a * pow(a, k_);
}}}
```

- Efficient execution

Background

- C is used in low-level infrastructure
  programming language, OS, network server, embedded devices, IoT, succinct data structures
  - C is great
    efficient, low-level features, reasonably portable, interoperability
  - C is dangerous
    buffer overrun, integer overflow, etc.
- Robust infrastructure is important
  - Absence of failures: avoid undefined behavior
  - Correctness: correct program logic

Coq Proof-assistant

- Contains Gallina (an ML-like language)
- Large proof library
- Mature proof system
- Extensible with plugin written in OCaml
- Program extraction to OCaml, Haskell, Scheme and JSON

Idea: Prove in Coq, Execute in C

- Write a program in Gallina
- Verify the program in Coq
  - Correctness
  - Absence of failures
- Translate Gallina to C
- Enjoy verified, efficient and interoperable C program

Partiality in Coq and C

Different stance on program failures
E.g., zero division, integer overflow, etc.

- Coq: All functions always succeed
  (All functions are total)
  E.g., 0 / 0 = 0
  n + 1 – 1 = n

- C: Various functions can fail
  (Functions can be partial)
  E.g., 0 / 0 is undefined (SIGFPE)
  n + 1 – 1 may overflow

→ Need to bridge the gap

Current Practice Pollutes Source Program

How to Treat Partial Functions

- Proof of "absence of failures"
  Needs to modify the source program:
  - option type everywhere (or option monad)
    Need to propagate None → Tedious programming

  or

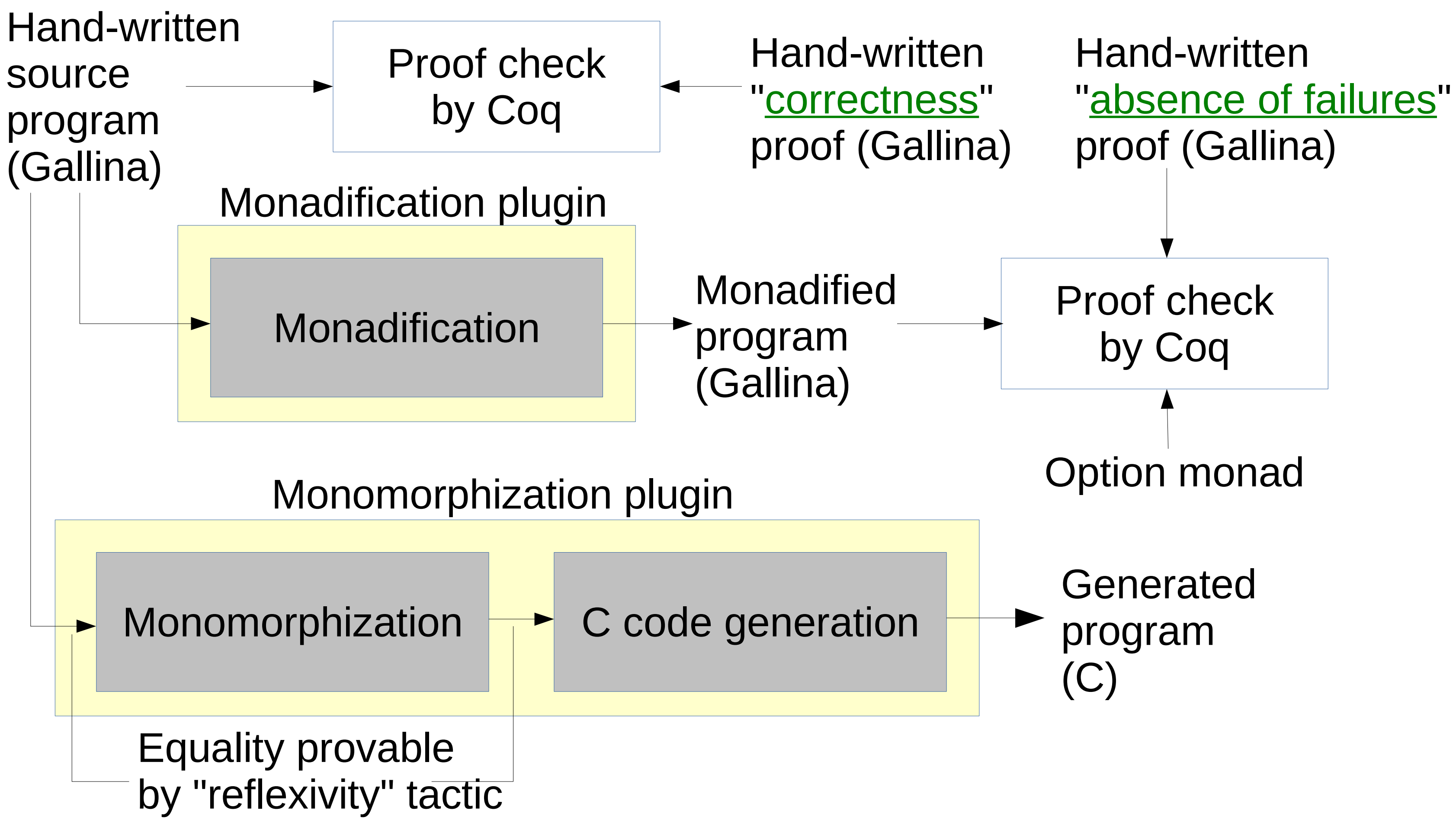
  - partial function takes a proof of the precondition
    Certified programming needs dependent type
- Proof of correctness
  Difficult with the modified program
- Inefficient code extraction
  None-propagation causes overhead
  It is difficult to delete all dependent types

Our Solution: Automatic Monadification

- Separate proofs in Coq
  (Separation of concerns)
  - Proof of correctness with original source program
    E.g., tail recursive pow = naive pow
  - Proof of "absence of failures" with automatically generated monadic program
    E.g., no integer-overflow with int
- Efficient C code generation
  - Fully-customizable datatype implementation
    E.g., replace nat to int
  - No runtime overhead
    E.g., no dynamic integer-overflow detection

Our Translation Scheme

Two Coq plugins: Monomorphization and Monadification



We don't Use Coq Extraction

- Coq extraction doesn't support C
  Difficult to use low-level features
  - 64-bit integer
  - SSE, AVX, etc.
  - goto (for proper tail-recursion)
- Coq extraction inhibits type specific implementation
  Optimization according to type is difficult
  - Dependent type support
  - Lack of type annotation in MiniML (intermediate language of extraction)
    i.e., Type inference on MiniML required
- Modularity
  Extraction is too big for us and difficult to deploy
  - Useless features for us: dependent type support, proof erasure, etc.
  - Coq itself must be built to use a modified extraction

Translation Steps

- Monomorphization
  - Remove polymorphism
  - The result is equal to the original (automatic formal proof)
- C code generation
  - Direct translation (no closures yet)
  - Fully-customizable data representation
- Monadification
  - For proof of "absence of failures" (program never fails)
  - Possible to use it for other proofs on computation
    E.g., complexity

Overview

Monomorphization

C Code Generation

Monadification

Experiments

Trusted Base

Conclusion

Monomorphization Example

- polymorphic functions

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

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

- monomorphic functions

```
Definition _pair_bool_bool :=
  @pair 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.
```

```
Goal swap_bb = _swap_bb. Proof. reflexivity. Qed.
```

Monomorphization

- Specialize functions w.r.t. type args
- In addition, insert let-bindings
  (similar to A-normal form for code generation)
- The result is equal to the original, modulo the conversion rule of Coq
  - β-reduction: function application
  - ζ-reduction: remove let-binding
- "reflexivity" tactic checks term equality by the conversion rule

Target Language of Monomorphization

- ML-polymorphic subset of Gallina
- Full Gallina is impossible to monomorphize
  - polymorphic recursion
  - dependent type
- Possible to monomorphize ML program
  cf. MLton
- ML is powerful enough

Overview

Monomorphization

C Code Generation

Monadification

Experiments

Trusted Base

Conclusion

Highlight of C Code Generation

- Data representation is fully customizable
  - mapping nat to a fixed integer type is possible
- Proper tail recursion using "goto"

C Code Generation Example: pow

- Generated C function

```
nat n2_pow(nat v88_a, nat v87_k)
{
 switch (sw_nat(v87_k)) {
   case_O_nat: {
     nat v90_n = n0_O();
     return n1_S(v90_n); }
   case_S_nat: {
     nat v91_k_ =
       field0_S_nat(v87_k);
     nat v92_n =
       n2_pow(v88_a, v91_k_);
     return n2_muln(v88_a, v92_n);
   }
  }
}
```

- Hand-written datatype implementation

```
#define nat uint64_t
#define n0_O() ((nat)0)
#define n1_S(n) ((n)+1)
#define sw_nat(n) (n)
#define case_O_nat case 0
#define case_S_nat default
#define field0_S_nat(n) ((n)-1)

#define n2_addn(a,b) ((a)+(b))
#define n2_subn(a,b) ((a)-(b))
#define n2_muln(a,b) ((a)*(b))
#define n2_divn(a,b) ((a)/(b))
#define n2_modn(a,b) ((a)%(b))
```

See the paper for details

AIST

Monadification

Absence of failures become provable

E.g., integer-overflow

- direct style

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

- monadic style

```
Fixpoint powM a k :=
  match k with
  | 0 => SM 0
  | k'.+1 =>
      powM a k' >>=
      mulM a
  end.
```

Why Monadification?

Proof about computation

E.g.,

- The program uses only nat values smaller than 2^{64}
- The index of array access is always less than the size
- The program invokes "cons" n times

Why Automatic Monadification?

- Hand-monadification is tedious
- Different proof needs different monads and monadic actions
  - "cons" constructor to count cons invocations
  - "S" constructor for integer overflow
- Need to monadify library, not only application

Configuration of Monadification Plugin

1. Set the monadic triple
   Monadify Type M.
   Monadify Return f.
   Monadify Bind f.

2. Register monadic actions
   Monadify Action f => fM.

3. Monadify a function and its dependencies
   Monadification f.

Option Monad for Program Failures

```
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.
Monadify Type option.
Monadify Return @ret.
Monadify Bind @bind.
(* Notations for ">>=" and "return" *)
```

Integer Overflow Detection

Registration of actions to detect integer-overflow:

```
Definition check x :=
  if Nat.log2 x < 32 then Some x else None.
Definition SM a := check a.+1.
Definition mulM a b := check (a * b).
Monadify Action S => SM.
Monadify Action muln => mulM.
```

Concrete Example of Monadification

- direct style (source)

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

- monadic style (generated)

```
Fixpoint powM a k :=
  match k with
  | 0 => SM 0
  | k'.+1 =>
      powM a k' >>=
      mulM a
  end.
```

Proof of "No Integer-Overflow"

- In general, we want to prove that the "program never fails" under *condition*, i.e.:
  forall x, *condition* → fM x = Some (f x)
- E.g., proof for "no integer overflow in pow":

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

Monad for Complexity

(Another application of monadification)

- Counter monad:
```
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).
```
- Count cons invocations:
```
Definition consM {T} (hd : T) tl := (1, cons hd tl).
Monadify Action cons => @consM.
```
- E.g., we proved that naive list reversal needs n(n+1)/2 invocations and tail-recursive list reversal needs only n

Idea of the Monadification Algorithm

- Insert fewer monads to ease the proof (it is better when fM is similar to f)
  - Best: $t_1 \to t_2 \to t_3$ (same as original)
  - Good: $t_1 \to t_2 \to M\ t_3$
  - Bad: $M\ (t_1 \to M\ (t_2 \to M\ t_3))$
- For most C functions, one M is enough
  - functions have no effect before the last argument is given
- Our algorithm infers the number of args before the first effect ("*impure arity*")

Insert Monads using "*Impure Arity*"

- fM is the monadified function of f

  $f: t_1 \to \ldots \to t_k \to t_{k+1} \to \ldots \to t_n$
  $fM : t_1 \to \ldots \to t_k \to$
  $\quad M\ (t_{k+1} \to \ldots \to M\ (t_{n-1} \to M\ t_n)\ldots)$

- We call k as impure arity
- Our algorithm chooses k as big as possible to ease the proof
- Two concrete problems that require a bigger k
  - Coq can reject definitions with k=0
    (decreasing argument and type argument)
- See the paper for details

Overview

Monomorphization

C Code Generation

Monadification

Experiments

Trusted Base

Conclusion

AIST

Experiments

1. Monadification of an existing, realistic theory
   - seq.v: SSReflect's list theory
   - Tried to monadify 49 functions
   - 7 is pure, 36 succeeds and 6 couldn't
     (dependent type, higher order constructor)
2. rank function for succinct data structure
   - Using monadification, we proved (in addition to <u>correctness</u>):
     - <u>absence of failures</u>
     - complexity
   - We generated C code;
     it uses customized datatype implementations:
     - bitstring implementation
     - array of small integers

Overview

Monomorphization

C Code Generation

Monadification

Experiments

Trusted Base

Conclusion

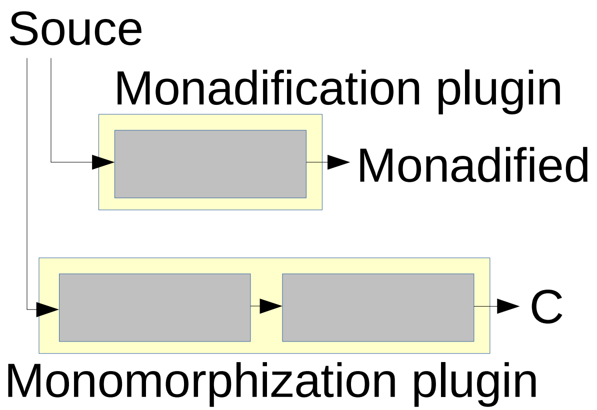
Our Trusted Base Smaller than Coq's Extraction

- Our C code gen.
  - g_monomorph.ml4 30
  - monoutil.ml 136
  - genc.ml 696
- Less than 1000 lines
- Monomorphization is not counted since the result is formally provable

- Coq 8.6 extraction
  - g_extraction.ml4 152
  - common.ml 648
  - extract_env.ml 682
  - extraction.ml 1098
  - mlutil.ml 1524
  - modutil.ml 411
  - table.ml 921
  - ocaml.ml 773
- Over 6000 lines

Summary

- We could generate verified, low-level C programs with a small trusted base
- Monomorphization
  - Remove polymorphism
  - The correctness is proved by the "reflexivity" tactic
- C code generation
  - Fully customizable data representation
  - Proper tail recursion using "goto"
- Monadification
  - New algorithm usable in Coq
  - Proof about computation: program failures, complexity



Future Work

- Release the plugins
- Datatype implementation generation
- More algorithms for succinct data structure
  select, wavelet tree, etc.
- Specialization w.r.t. non-type args
  (i.e., partial evaluation)
- Pluggable GC and closures
- Linear types

Extra Slides

Prove a Program Never Fail

- Option monad
  - Automatic monadification and proof
    → Needs plugin
  - Write a program in monadic style with option monad (No program in direct style)
    → Tedious programming
    Difficult to remove option monad at extraction (Runtime overhead)
    - functor and identity monad
      → modules are not expanded in OCaml extraction
    - section and identity monad
      → needs to inline all functions (Too much code duplication)
- Don't fail in C
  `#define n2_divn(a,b) ((b) == 0 ? 0 : (a)/(b))`
  Use GMP for integer overflow
  → Runtime overhead
- Certified Programming (cf. CPDT)
  → Very difficult proof
  Needs extraction (proof erasure)
  Need to decide uint64_t or GMP at beginning
- Deep embedding using template-coq
  → Difficult proof
- Return an unknown value, u, for failures
  → wrong proof
  (let x := u in 0) = 0. (u – u) = 0 for u:nat. (if u then e else e) = e for u:bool.

Details of C Code Generation

Monomorphization before C Code Generation Example

- source program

```
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).
```

- monomorphized program

```
Fixpoint _buildDir2 b s sz2
    c i D2 m2 :=
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.
```

C Code Generation Example

- monomorphized program

```
Fixpoint _buildDir2 b s sz2
    c i D2 m2 :=
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.
```

- C program

```
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;
}}}
```

C Code Generation is Direct

- monomorphized type name is used as-is
- function name is prefixed with the arity
  _buildDir2 → n7_buildDir2
- variable → variable
- let → variable initialization
- application → function call
  or goto for tail recursion
- match → switch

Data Type Implementation

- Data representation is fully customizable
- bool in Coq:
  ```
  Inductive bool : Set := true : bool | false : bool.
  ```
- bool implementation in C:
  ```
  #include <stdbool.h>
  #define n0_true() true
  #define n0_false() false
  #define sw_bool(b) (b)
  #define case_true_bool default
  #define case_false_bool case false
  ```

nat Implementation

- natural number in Coq: nat
  Inductive nat : Set := O : nat | S : nat → nat.
- nat implementation in C:
```
#define nat uint64_t
#define n0_O() ((nat)0)
#define n1_S(n) ((n)+1)
#define sw_nat(n) (n)
#define case_O_nat case 0
#define case_S_nat default
#define field0_S_nat(n) ((n)-1)
#define n2_addn(a,b) ((a)+(b))
```
- Integer overflow on uint64_t doesn't occur
  if we prove it using monadification

match → switch

- Coq

```
Inductive I :=
...
| Ci : ... → tij → ... → I
...
match v with
...
| Ci ...xij... => e
...
end
```

- C

```
switch (sw_I(v)) {
  ...
  case_Ci_I: {
    ...
    tij xij = field(i-1)_I(v);
    ...
    /* code for ei */
  }
  ...
}
```

Experiment
Monadification of SSReflect's seq.v

Monadification of seq.v

- Monadify 49 functions: all, allpairs, behead, belast, cat, catrev, constant, count, drop, filter, find, flatten, foldl, foldr, has, head, incr_nth, index, iota, iter, last, map, mask, mkseq, ncons, nilp, nth, ohead, pairmap, perm_eq, pmap, rem, reshape, rev, rot, rotr, scanl, seqn, set_nth, shape, size, subseq, sumn, take, undup, uniq, unzip1, unzip2 and zip.
- Monadic action: S and cons
- 7 is pure: behead, drop, head, last, nth, ohead and subseq
- 36 is successfully monadified
- 6 couldn't: constant, index, perm_eq, undup, uniq and seqn
  - seqn uses dependent type
  - others use higher order constructor (nat_eqType and seq_eqType also have same problem)

AIST

Experiment
rank function for succinct data structure

rank Function

- "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).

rank for Succinct Data Structure

- "rank_init b s" precomputes the auxiliary data: o(n) size in O(n) time
- "rank_lookup aux i" compute rank: O(1) time
- Functional correctness proved

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

- It never fail if $n < 2^{64}$

```
Lemma RankSuccess b s i :
  let n := bsize s in log2 n < 64 → i <= n →
  (rank_initM b s >>= fun aux => rank_lookupM aux i)
  = Some (rank_lookup (rank_init b s) i).
```

- We also proved the time complexity