Ruby Extension Library Verified using Coq Proof-assistant

Tanaka Akira @tanaka_akr

National Institute of Advanced Industrial Science and Technology (AIST)

RubyKaigi 2017

2017-09-20

Supplement material: https://github.com/akr/coq-html-escape

About This Talk

- Formal verification for fast & safe program in C
- Quality assurance other than test

Materials

- Ruby
- Coq
- C
- HTML escape
- Intel SSE

Do you know all of them?

Coq Proof-assistant

- Proof assistant
  - User writes a proof
  - Coq checks the proof
- Coq has ML-like language, Gallina
  - Powerful type system
  - Gallina programs can be proved
- Program Extraction to OCaml, Haskell and Scheme
- C code generation by our plugin
  https://github.com/akr/codegen

Development Flow

1. In Coq
    i. Define a specification and implementation
    ii. Verify them
    iii. Convert the implementation into C
2. In C
    i. Define supplemental code
    ii. Define glue code for Ruby
3. In Ruby
    i. Use the verified implementation

Benefits of Verification

- Correct
- Fast

Compared to:

- Ruby: safe but slow
- C: fast but dangerous

Simple Example: pow

Specification of power function in Gallina:

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

Good: Obviously correct

Bad: Naive algorithm

Bad: (non-tail) recursion

Complex but Fast pow

```
Definition uphalf' n := n – n./2.
(* fastpow_iter a k x = (a ** k) * x *)
Fixpoint fastpow_iter a k x :=
  if k is k'.+1 then
    if odd k then
      fastpow_iter a k' (a * x)
    else
      fastpow_iter (a * a) (uphalf' k') x
  else
    x.
Definition fastpow a k := fastpow_iter a k 1.
```

Complex and Fast pow (2)

Bad: Not obviously correct

Good: Fast algorithm

Good: Tail recursion

Correctness for fastpow

- We can prove equality of fastpow and pow in Coq

  ```
  Lemma fastpow_pow a k : fastpow a k = pow a k.
  Proof. (*snip*) Qed.
  ```

- This is the evidence that fastpow is correct
- The proof is snipped because Coq proof is unreadable
  (interactive environment is required to read proof)

Code Generation from fastpow

```
nat n3_fastpow_iter(nat v2_a, nat v1_k, nat v0_x) {
  n3_fastpow_iter:;
  switch (sw_nat(v1_k)) {
    case_O_nat: { return v0_x; }
    case_S_nat: {
      nat v4_k_ = field0_S_nat(v1_k);
      bool v5_b = n1_odd(v1_k);
      switch (sw_bool(v5_b)) {
      case_true_bool: {
        nat v6_n = n2_muln(v2_a, v0_x);
        v1_k = v4_k_; v0_x = v6_n; goto n3_fastpow_iter; }
      case_false_bool: {
        nat v7_n = n2_muln(v2_a, v2_a);
        nat v8_n = n1_uphalf_(v4_k_);
        v2_a = v7_n; v1_k = v8_n; goto n3_fastpow_iter; }}}}}
nat n2_fastpow(nat v10_a, nat v9_k) {
  nat v11_n = n0_O();
  nat v12_n = n1_S(v11_n);
  return n3_fastpow_iter(v10_a, v9_k, v12_n); }
```

Primitives for fastpow

- Types
  - bool: Boolean
  - nat: Peano's natural number
- Functions
  - odd: nat → bool
  - muln: nat → nat → nat

bool

- Coq definition

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

- C Implementation

```
/* bool type */
#include <stdbool.h>

/* constructors */
#define n0_true() true
#define n0_false() false
```

```
/* macros for "match" */
#define sw_bool(b) (b)
#define case_true_bool default
#define case_false_bool case false
```

nat (Peano's natural number)

- Coq definition

```
Inductive nat : Set :=
| O : nat              (* zero *)
| S : nat → nat.       (* successor function *)
```

- C Implementation

```
typedef uint64_t nat;
#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)
```

```
/* primitive functions */
#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))
#define n1_odd(n) ((n)&1)
```

Verified Program Development

- Describe a program in Gallina
- Describe a proposition (Gallina type)
- Describe a proof (Gallina program)
- Coq checks the proof (type check)
- Generate C code from the Gallina program
- Define supplemental C code

Curry-Howard Correspondence

They have same structure:

- proposition ~ type
- proof ~ program

"Prove a proposition" =
"Write a program of the correspond type"

Example: $A \land B$

- proof
  - A, B : propositions
  - $A \land B$: proposition of "A and B".
  - proof for $A \land B$: pair of proof for A and proof for B
- program
  - A, B : types
  - $A \land B$: pair of A and B
    type AandB = A * B
  - value of $A \land B$: pair of value of A and a value of B

Logical Formulas

- Propositional logic as usual types
  - and : type AandB = A * B
  - or : type AorB = a of A | b of B
  - imply : type AimplyB = A → B

- Predicate logic as dependent types
  - ∀x:A. B : A → B
  - ∃x:A. B : pair of x and proof of B
  - x = y : equality

Specification and Correctness

- spec(x) = obviously-correct-function
- imp(x) = complex-function

- proposition of correctness:

  $\forall x.\ imp(x) = spec(x)$

  (Other form of specification is possible...)

Code Generation to C

- C code generation by our plugin
  https://github.com/akr/codegen

- Simple mapping from Gallina subset to C
- Tail recursion is translated to goto
- Fully customizable implementation of data types

What is Verified?

Verified:

- The algorithm of fastpow

Not Explained

- Program failures (such as integer overflow)
  It is possible to prove about program failures using our monadification plugin. But we ignore this issue today.

Not Verified

- Translation mechanism to C
- Implementation of primitives:
  bool, nat, muln, odd

HTML Escape

CGI.escapeHTML substitutes five characters in a string:

& → &

< → <

> → >

" → "

' → '

We ignore non-ASCII characters for simplicity.

HTML Escape Specification

```
Definition html_escape_alist :=
  map (fun p => (p.1, seq_of_str p.2)) [::
  ("&"%char, "amp"); ("<"%char, "lt"); (">"%char, "gt");
  (""""%char, "quot"); ("'"%char, "#39") ].

Definition html_escape_byte c :=
  if assoc c html_escape_alist is Some p then
    "&" ++ p.2 ++ ";"
  else
    [:: c].

Definition html_escape s := flatten (map html_escape_byte s).
```

This seems correct but doesn't work optimal in C: list (seq) and higher order function

Primitive Types for HTML Escape

Required types for "scan a memory region and store the escaped result into a buffer"

Coq	C
bool	→ bool
nat	→ uint64_t
ascii	→ unsigned char
byteptr	→ char*
buffer	→ Ruby's VALUE (String)

ascii type (unsigned char)

- Coq definition

```
(* ascii is 8 booleans *)
Inductive ascii : Set := Ascii (_ _ _ _ _ _ _ _ : bool).
```

- C Implementation

```
typedef unsigned char ascii;
```

byteptr type (char*)

- Required operations to scan a memory region: advance a pointer, dereference a pointer
- Coq definition
  "char*" is represented using a list of ascii and an index in it
  Inductive byteptr := bptr : nat → seq ascii → byteptr.
  bptradd (bptr i s) n = bptr (i + n) s
  bptrget (bptr i s) = nth "000"%char s i
- C Implementation

```
typedef const char *byteptr;
#define n2_bptradd(p, n) (p + n)
#define n1_bptrget(p) (*(unsigned char *)p)
```

buffer type (VALUE)

- Required operation for result buffer:
  add data at end of buffer
- Coq definition
  Inductive buffer := bufctr of seq ascii.
  Definition bufaddmem buf ptr n := ...
- C Implementation
  - buffer: VALUE (String)
  - bufaddmem: rb_str_buf_cat
- bufaddmem is pure but rb_str_buf_cat is destructive.
  This problem is solved by copying the string when necessary

Tail Recursive HTML Escape Translatable to C

```
Fixpoint trec_html_escape buf ptr n :=
  match n with
  | 0 => buf
  | n'.+1 =>
      let: (escptr, escn) :=
        html_escape_byte_table (bptrget ptr) in
      trec_html_escape
        (bufaddmem buf escptr escn)
        (bptradd ptr 1)
        n'
  end.
```

Correctness of Tail Recursive HTML Escape

- Definition trec_html_escape_stub s :=
  s_of_buf (trec_html_escape
  (bufctr [::]) (bptr 0 s) (size s)).
- Lemma trec_html_escape_correct s :
  trec_html_escape_stub s = html_escape s.
  Proof. (*snip*) Qed.

Translated trec_html_escape in C

```
buffer n3_trec_html_escape(buffer v2_buf, byteptr v1_ptr, nat v0_n) {
  n3_trec_html_escape:;
  switch (sw_nat(v0_n)) {
    case_O_nat: { return v2_buf; }
    case_S_nat: {
      nat v4_n_ = field0_S_nat(v0_n);
      ascii v5_a = n1_bptrget(v1_ptr);
      prod_byteptr_nat v6_p = n1_html_escape_byte_table(v5_a);
      byteptr v7_escptr = field0_pair_prod_byteptr_nat(v6_p);
      nat v8_escn = field1_pair_prod_byteptr_nat(v6_p);
      buffer v9_b = n3_bufaddmem(v2_buf, v7_escptr, v8_escn);
      nat v10_n = n0_O();
      nat v11_n = n1_S(v10_n);
      byteptr v12_b = n2_bptradd(v1_ptr, v11_n);
      v2_buf = v9_b;
      v1_ptr = v12_b;
      v0_n = v4_n_;
      goto n3_trec_html_escape; } } }
```

branch by
switch statement

Jump by
goto statement

Primitive Type for SSE

- m128 → __m128i
- __m128i is defined by intrinsics for SSE

m128 type

- m128 consists 16 bytes. (SSE register is 128 bits)
- Coq definition

```
Inductive m128 := c128 :
    ascii → ascii → ascii → ascii → ascii → ascii → ascii → ascii →
    ascii → ascii → ascii → ascii → ascii → ascii → ascii → ascii →
    m128.
Definition m128_of_seq s := c128
    (nth "000"%char s 0) ...(snip)... (nth "000"%char s 15).
Definition m128_of_bptr ptr :=
    m128_of_seq (drop (i_of_bptr ptr) (s_of_bptr ptr)).
```

- C Implementation

```
typedef __m128i m128;
#define n1_m128_of_bptr(p) _mm_loadu_si128((__m128i const*)(p))
```

SSE4.2 pcmpestri instruction

- pcmpestri:
  Packed Compare Explicit Length Strings, Return Index
- Coq definition

```
Definition cmpestri_ubyte_eqany_ppol_lsig
    (a : m128) (la : nat) (b : m128) (lb : nat) :=
  let sa := take la (seq_of_m128 a) in
  let sb := take lb (seq_of_m128 b) in
  let p := mem sa in
  if has p sb then find p sb else 16.
```

- C Implementation

```
#define n4_cmpestri_ubyte_eqany_ppol_lsig(a, la, b, lb) \
  _mm_cmpestri(a, la, b, lb, \
      _SIDD_UBYTE_OPS|_SIDD_CMP_EQUAL_ANY| \
      _SIDD_POSITIVE_POLARITY|_SIDD_LEAST_SIGNIFICANT)
```

- _mm_cmpestri is SSE intrinsic function which generates pcmpestri.

HTML Escape using SSE

```
Fixpoint sse_html_escape buf ptr m n :=
  match n with
  | 0 => bufaddmem buf ptr m
  | n'.+1 =>
      let p1 := bptradd ptr m in
      if n <= 15 then
        trec_html_escape (bufaddmem buf ptr m) p1 n
      else
        let i := cmpestri_ubyte_eqany_ppol_lsig
            chars_to_escape num_chars_to_escape
            (m128_of_bptr p1) 16 in
        if 16 <= i then
          sse_html_escape buf ptr (m + 16) (n' - 15)
        else
          let buf2 := bufaddmem buf ptr (m + i) in
          let p2 := bptradd ptr (m + i) in
          let c := bptrget p2 in
          let p3 := bptradd p2 1 in
          let: (escptr, escn) := html_escape_byte_table c in
          let buf3 := bufaddmem buf2 escptr escn in
          sse_html_escape buf3 p3 0 (n' - i)
  end.
```

Correctness of HTML Escape using SSE

- Definition sse_html_escape_stub s :=
  s_of_buf (sse_html_escape
    (bufctr [::]) (bptr 0 s) 0 (size s)).
- Lemma sse_html_escape_correct s :
  sse_html_escape_stub s = html_escape s.
  Proof. (*snip*) Qed.

- This verification doesn't need real CPU which support SSE4.2

Glue Code for Ruby Extension

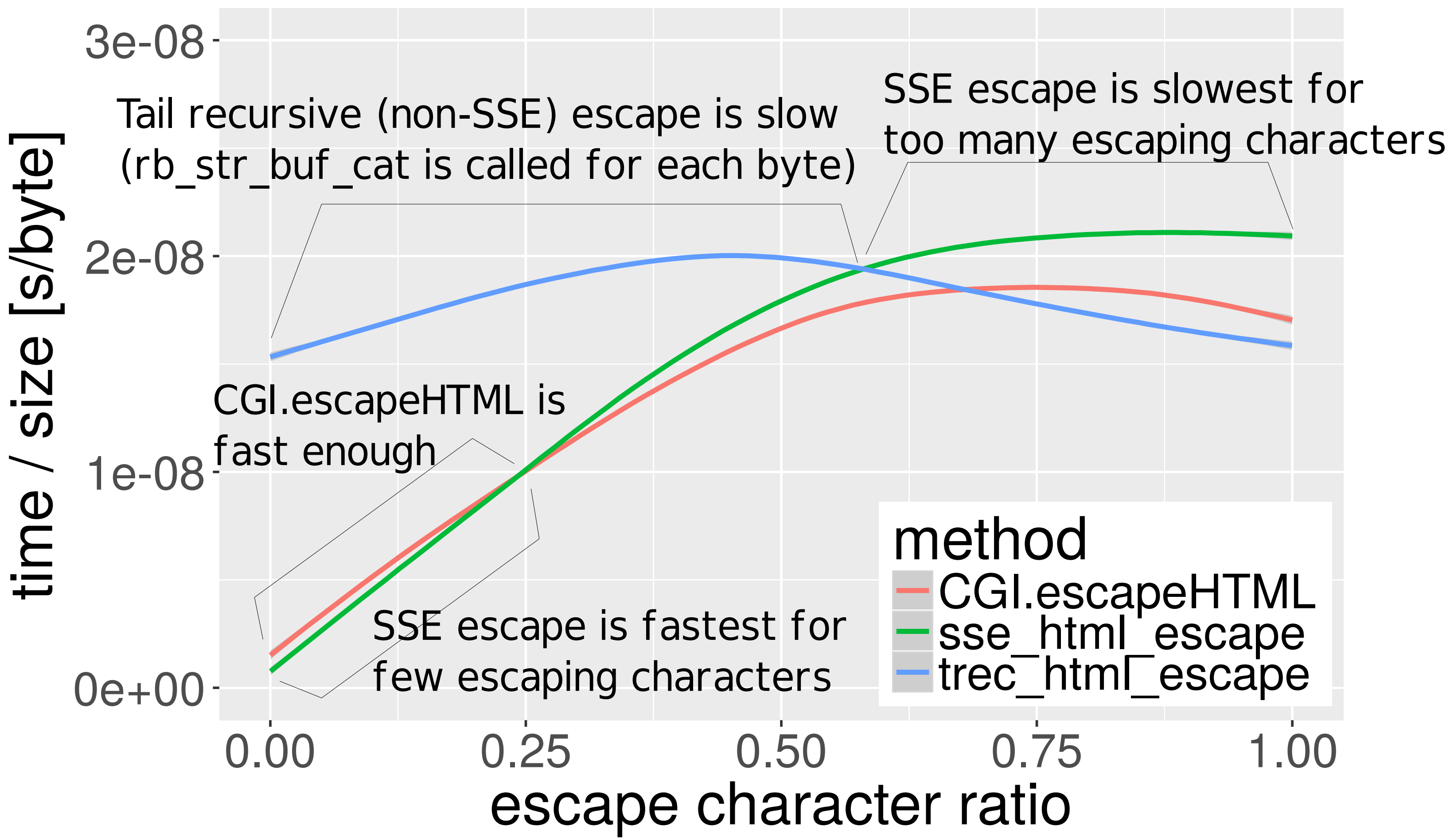
```
VALUE
sse_html_escape(VALUE self, VALUE str)
{
  buffer buf;
  StringValue(str);
  RB_GC_GUARD(str);
  buf = buffer_new(RSTRING_LEN(str));
  n4_sse_html_escape(buf, RSTRING_PTR(str), 0, RSTRING_LEN(str));
  return buf.str;
}

void
Init_verified_html_escape()
{
  rb_define_global_function("sse_html_escape", sse_html_escape, 1);
}
```

Test

```
% ruby -I. -rverified_html_escape \
    -e 'p sse_html_escape("x < y")'
"x &lt; y"
```

Benchmark



3e-08
2e-08
1e-08
0e+00
time / size [s/byte]
Tail recursive (non-SSE) escape is slow
(rb_str_buf_cat is called for each byte)
SSE escape is slowest for
too many escaping characters
CGI.escapeHTML is
fast enough
SSE escape is fastest for
few escaping characters
method
CGI.escapeHTML
sse_html_escape
trec_html_escape
0.00
0.25
0.50
0.75
1.00
escape character ratio

Some Thoughts

- pcmpestrm instruction would be faster than pcmpestri.

Encouragement of Coq

- Realistic programming is possible
- You will learn about "correctness" more precisely

How to write a correct program

Think program's behavior seriously

No tool can support non-thinking person

Think Seriously

Most real programs are too complex to think in a brain

We need an external tool to think the behavior:

- Write the behavior
- Read it and re-thinking

Write the behavior in ...

- Natural language
  - Good: Very flexible
  - Bad: Too flexible, no automatic checking
- Programming language
  - Good: Runnable and testable
  - Bad: Ad-hoc test is very sparse
- Test driven development (TDD)
  - Good: Many examples make us more thinking
  - Bad: Not possible to test all (infinite) inputs
- Formal verification
  - Good: Coq forces us to think correctness for all inputs
  - Bad: Proof is tedious

Importance of learning formal verification

- You will learn how to describe correctness very preciously.
- As you learned how to describe behavior very preciously by learning programming

Summary

- Correct and fast C function can be generated from Coq
- The function is usable from Ruby
- Encourage Coq to learn about correctness

Extra Slides

Benchmark Script

```
require 'cgi'
require 'verified_html_escape'
methods = %w[sse_html_escape trec_html_escape CGI.escapeHTML]
puts "size[byte],method,esc_ratio,time[s]"
max_size = 40000
num_sizes = 200
num_ratios = 50
code = []
methods.each {|meth|
  num_sizes.times {
    num_ratios.times {
      sz = 1+rand(max_size-1)
      esc_ratio = rand
      code << <<~End
        sz = #{sz}
        meth = #{meth.dump}
        esc_ratio = #{esc_ratio}
        num_escape = (sz * esc_ratio).to_i
        src = (['a'] * (sz - num_escape) + ['&'] * num_escape).shuffle.join
        GC.disable
        t1 = Process.clock_gettime(Process::CLOCK_THREAD_CPUTIME_ID)
        dst = #{meth}(src)
        t2 = Process.clock_gettime(Process::CLOCK_THREAD_CPUTIME_ID)
        GC.enable
        t = t2-t1
        puts "\#{sz},\#{meth},\#{esc_ratio},\#{t}"
      End
    }
  }
}
eval code.shuffle.join
```