Verifiable C

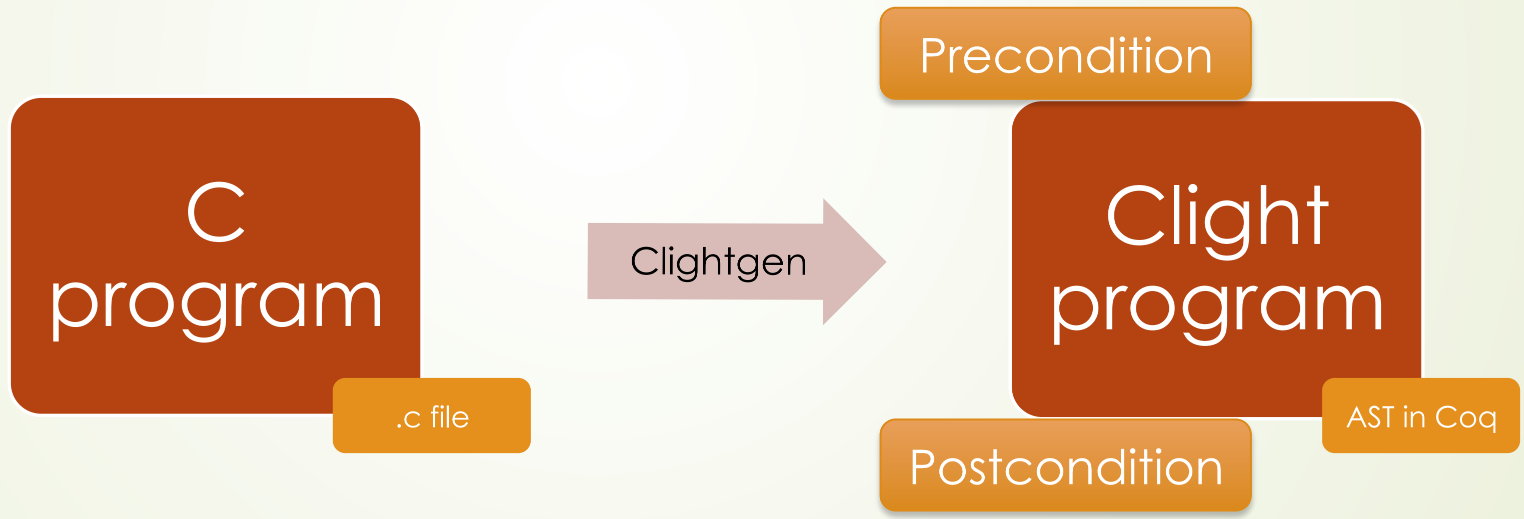
--- A logic and system for proving C programs correct

Qinxiang Cao
Princeton University

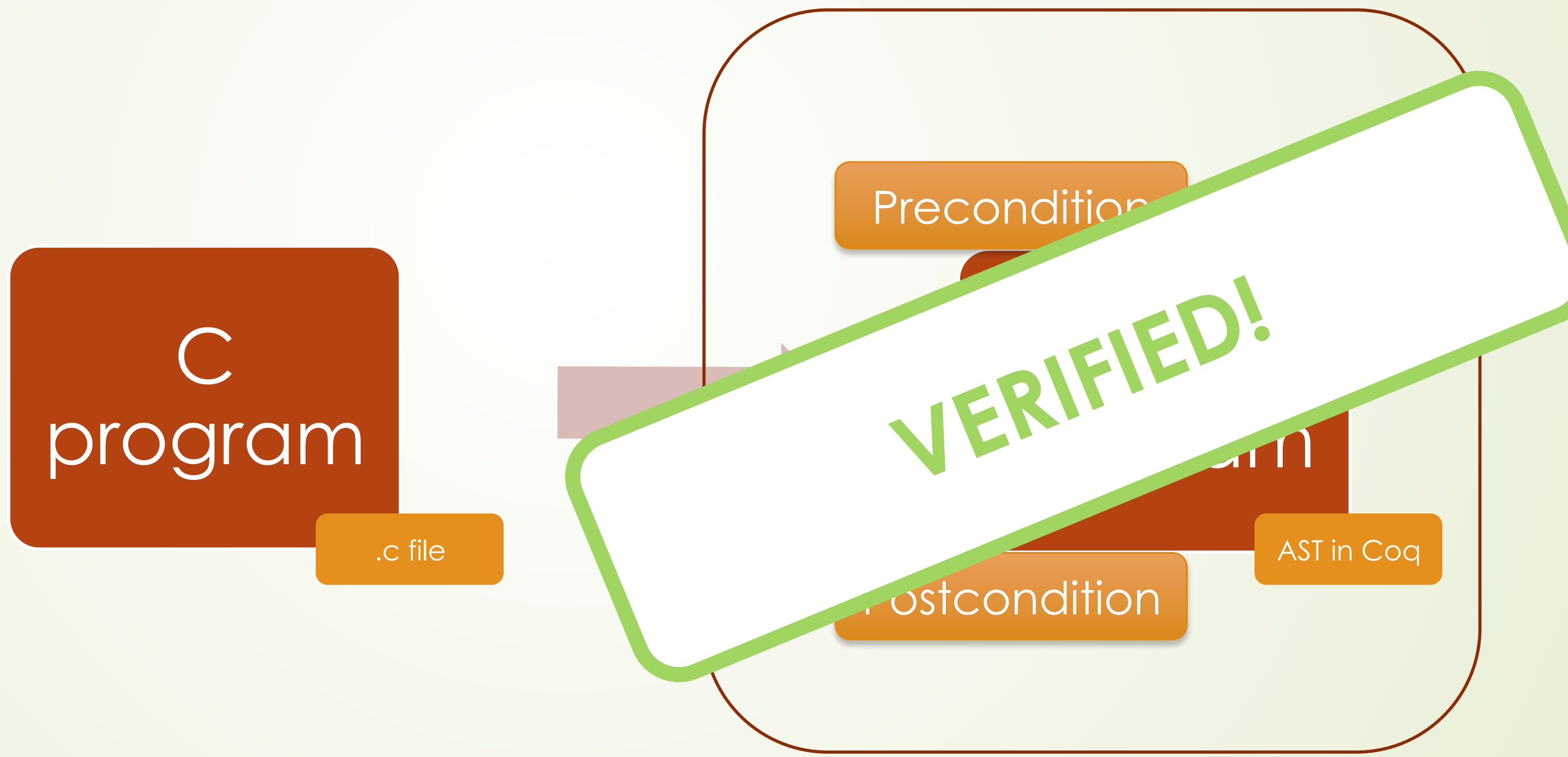
Talk Outline

- What is Verifiable C?
- How to use Verifiable C?
- Verifiable C's soundness.

What is Verifiable C?

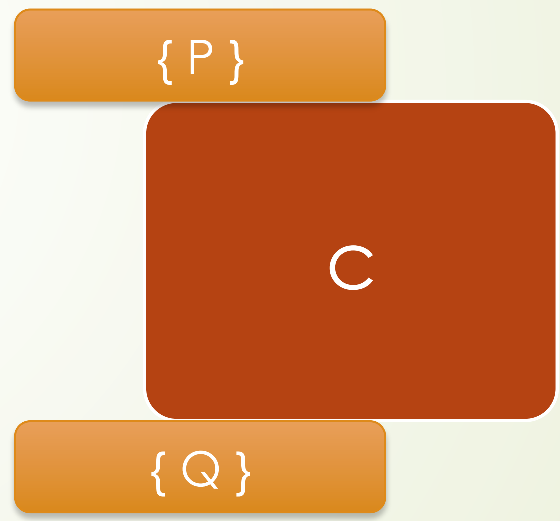


What is Verifiable C?



What is Verifiable C?

1. If **P**, executing **c** is safe.
2. If **P**, the termination of **c** ensures **Q**.



Proof rules of Verifiable C

- If $\{P\}c_1\{Q\}$ and $\{Q\}c_2\{R\}$, then $\{P\}c_1;c_2\{R\}$.
- If $\{P\}c\{Q\}$, $P' \vdash P$ and $Q \vdash Q'$, then $\{P'\}c\{Q'\}$.
- …

Programs verified by Verifiable C

- Linked list reverse: for any σ,
- $\{listrep(\sigma,[\![p]\!])\}$
-   reverse(p)
- $\{listrep(rev(\sigma),[\![return]\!])\}$

Denotation of program variable p.

```
struct list {
  unsigned head;
  struct list *tail;
};

struct list *reverse (struct list *p);
```

A list σ is stored at address $[\![p]\!]$ as a linked list.

Programs verified by Verifiable C

- Linked list concatenation: for any σ and σ',
- $\{listrep(\sigma,[\![p]\!]) * listrep(\sigma' ,[\![q]\!])\}$
- concate(p)
- $\{listrep(\sigma \cdot \sigma' ,[\![return]\!])\}$

Separating conjunction! Tw linked lists occupy disjoint pieces of memory.

```
struct list {
  unsigned head;
  struct list *tail;
};

struct list * concate(struct list *p,
struct list *q);
```

Programs verified by Verifiable C

Logical variable ***p*.**

- Merge sort (verified by Jean-Marie Madiot):
- $\{ [\![p]\!] = p \wedge listrep(\sigma, p) \}$
- mergesort(p)
- $\{ \exists \sigma' .\ Permutation(\sigma, \sigma') \wedge Sorted(\sigma') \wedge listrep(\sigma', p) \}$

```
struct list {
  unsigned head;
  struct list *tail;
};

void mergesort(struct list *p);
```

Programs verified by Verifiable C

- In-place reverse array:
- $\{[\![a]\!]=p \wedge [\![n]\!]=n \wedge len(\sigma) = n \wedge p\mapsto\sigma) \}$
- reverse(a, n)
- $\{p\mapsto rev(\sigma)\}$

```
void reverse(int a[], int n) {
   int lo, hi, s, t;
   lo=0;
   hi=n;
   while (lo<hi-1) {
      t = a[lo];
      s = a[hi-1];
      a[hi-1] = t;
      a[lo] = s;
      lo++; hi--;
   }
}
```

Programs verified by Verifiable C

- FIFO queue operations (implemented by array):
  - Enqueue, dequeue.
- BST operations:
  - Insert, look-up, delete
- Union-find (by Shengyi Wang).
- Knuth-Morris-Pratt algorithm (by Luke Chen and Qinxiang Cao).

Programs verified by Verifiable C

- Cryptography primitives
  - SHA-256, OpenSSL open-source version (Verification of a Cryptographic Primitive: SHA-256, by Andrew Appel, TOPLAS 2015)
  - HMAC, OpenSSL open-source version (Verified Correctness and Security of OpenSSL HMAC, by Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. In USENIX Security Symposium 2015)
  - DRBG, mbedTLS version (Verified Correctness and Security of mbedTLS HMAC-DRBG by Katherine Q. Ye, Matthe Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. CCS'17)
  - AES, mbedTLS version (By Samuel Grutter)

Programs verified by Verifiable C

- A B-tree implementation for data base
- Memory management library (By David A. Naumann)
- Part of the C string library (By William Mansky)

Talk Outline

- What is Verifiable C?
- How to use Verifiable C?
- Verifiable C's soundness.

Canonical assertion

- PROP clauses: pure assertions
- LOCAL clauses: ``denotations'' of program variables
- SEP clauses: data stored in memory

Canonical assertion

- Precondition:

- Postcondition:

Linked list reverse:

$\{listrep(\sigma, [\![p]\!])\}$
reverse(p)
$\{listrep(rev(\sigma), [\![return]\!])\}$

Canonical assertion

- Precondition:
  - PROP ()
  - LOCAL (temp _p p)
  - SEP (listrep sigma p)

- Postcondition:
  - EX q,
  - PROP ()
  - LOCAL (temp ret_temp q)
  - SEP (listrep (rev sigma) q)

Linked list reverse:

$\{[\![p]\!]=p \wedge listrep(\sigma,p)\}$

reverse(p)

$\{ \exists q . [\![return]\!] = q$
$listrep (rev (\sigma) , q$

Canonical assertion

- Precondition:
  - PROP ()
  - LOCAL (temp _p p; temp _q q)
  - SEP (listrep sigma p; listrep sigma’ q)

- Postcondition:
  - EX r,
  - PROP ()
  - LOCAL (temp ret_temp r)
  - SEP (listrep (sigma ++ sigma’) r)

Linked list concatenation:

$\{[\![p]\!]=p \wedge [\![q]\!]=q \wedge listrep(\sigma,p) * listrep(\sigma\upharpoonleft' ,$

concate(p)

$\{\exists r. [\![return]\!]=r \wedge listrep(\sigma\cdot\sigma\upharpoonleft' ,r)\}$

Canonical assertion

- Precondition:
  - PROP ()
  - LOCAL (temp _p p)
  - SEP (listrep sigma p)

Merge sort:

$\{⟦p⟧ = p \land listrep(\sigma, p)\ \}$

mergesort(p)

$\{\blacksquare \exists \sigma' .\ Permutation(\sigma, \sigma')\land$ $\square$

$Sorted(\sigma') \land listrep(\sigma', p)\ \}$

- Postcondition:
  - EX sigma’,
  - PROP (Permutation sigma sigma’, sorted sigma’)
  - LOCAL ()
  - SEP (listrep sigma’ p)

Apply Hoare logic in Coq

- Toy example #1:

```
unsigned int last_foo(struct list * p) {
  unsigned int res;
  p = reverse (p);
  res = p -> head;
  return res;
}
```

Apply Hoare logic in Coq

Argument list of C function

```
Definition last_foo_spec :=
 DECLARE _last_foo
  WITH sigma : list int, p: val, sigma': list int, x: int
   PRE  [ _p OF (tptr t_struct_list) ]
     PROP (sigma = sigma' ++ x :: nil)
     LOCAL (temp _p p)
     SEP (listrep sigma p)
   POST [ tuint ]
     PROP () LOCAL (temp ret_temp (Vint x))
     SEP (TT).
```

Return type: unsigned int

For any p, σ, σ' and x:

$\{■\sigma=\sigma' \cdot [x] \wedge □ [\![p]\!]=p \wedge □$
$listrep(\sigma,p)\ \}$

ret = last_foo(p)

$\{[\![ret]\!]=x\}$

DEMO

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
      p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p' and q,
// PROP ( )  LOCAL (temp _p p')  SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
      res = p -> head;
// PROP ( )  LOCAL (temp _p p'; temp _res (Vint x))
//           SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
      return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
        p = reverse (p);
        res = p -> head;
        return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

25

ward_call
sigma' ++ [x], p).

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
        p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
        res = p -> head;
        return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

ward_call
sigma' ++ [x], p).
ros p'.

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
      p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p'.
// PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
      res = p -> head;
      return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

ward_call
sigma' ++ [x], p).
ros p'.
(* about list *)

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
      p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p'.
// PROP ( )  LOCAL (temp _p p')  SEP (listrep (x :: sigma') p')
      res = p -> head;
      return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

ward_call
sigma' ++ [x], p).
ros p'.
(* about list *)

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
        p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p'.
// PROP ( )  LOCAL (temp _p p')  SEP (EX q', p' |----> (Vint x, q) * listrep (rev sigma'
q)
        res = p -> head;
        return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

ward_call
sigma' ++ [x], p).
os p'.
(* about list *)
os q.

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
        p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p', q.
// PROP ( )  LOCAL (temp _p p')  SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
        res = p -> head;
        return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

0

ward_call
sigma' ++ [x], p).
ros p'.
(* about list *)
ros q.
ward.

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
        p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p', q.
// PROP ( )  LOCAL (temp _p p')  SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
        res = p -> head;
 // PROP ( )  LOCAL (temp _p p'; temp _res (Vint x))
//              SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
        return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

31

ward_call
sigma' ++ [x], p).
ros p'.
(* about list *)
ros q.
ward.
ward.

```
// PROP ( )  LOCAL (temp _p p)  SEP (listrep (sigma' ++ [x]) p)
        p = reverse (p);
// EX p', PROP ( )  LOCAL (temp _p p')  SEP (listrep (rev (sigma' ++ [x])) p')
// Given p', q.
// PROP ( )  LOCAL (temp _p p')  SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
        res = p -> head;
 // PROP ( )  LOCAL (temp _p p'; temp _res (Vint x))
//              SEP (p' |----> (Vint x, q); listrep (rev sigma') q)
       return res;
// PROP ( )  LOCAL (temp _ret_temp (Vint x))  SEP (TT)
```

List of forward verification tactics:

- For C singleton commands:
  - Forward, forward_call.
- For extracting existential quantifiers:
  - Intros.
- For weakening precondition:
  - Coq's built-in (rewrite, unfold, etc)

List of forward verification tactics:

- For C singleton commands:
  - Forward, forward_call.
- For extracting existential quantifiers:
  - Intros.
- For weakening precondition:
  - **Gather_SEP**, **replace_SEP**, **sep_apply**, Coq's built-in (rewrite, unfold, etc)
- For solving entailments:
  - **Entailer!**, **Exists**.

Apply Hoare logic in Coq

- Toy example #2 (Fibonacci numbers):

```
int fib_loop(int n) {
  int a0 = 0, a1 = 1, a2;
  int i;
  for (i = 0; i < n; ++ i) {
    a2 = a0 + a1;
    a0 = a1;
    a1 = a2;
  }
  return a0;
}
```

```
int fib_loop_save_var(int n) {
  int a0 = 0, a1 = 1;
  for (; n > 0; -- n) {
    a1 = a0 + a1;
    a0 = a1 - a0;
  }
  return a0;
}
```

```
int fib_rec(int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return fib_rec(n - 2) +
         fib_rec(n - 1);
}
```

DEMO

List of forward verification tactics:

- For C singleton commands:
  - Forward, forward_call.
- For extracting existential quantifiers:
  - Intros.
- For weakening precondition:
  - Gather_SEP, replace_SEP, sep_apply, Coq's built-in (rewrite, unfold, etc)
- For solving entailments:
  - Entailer!, Exists.

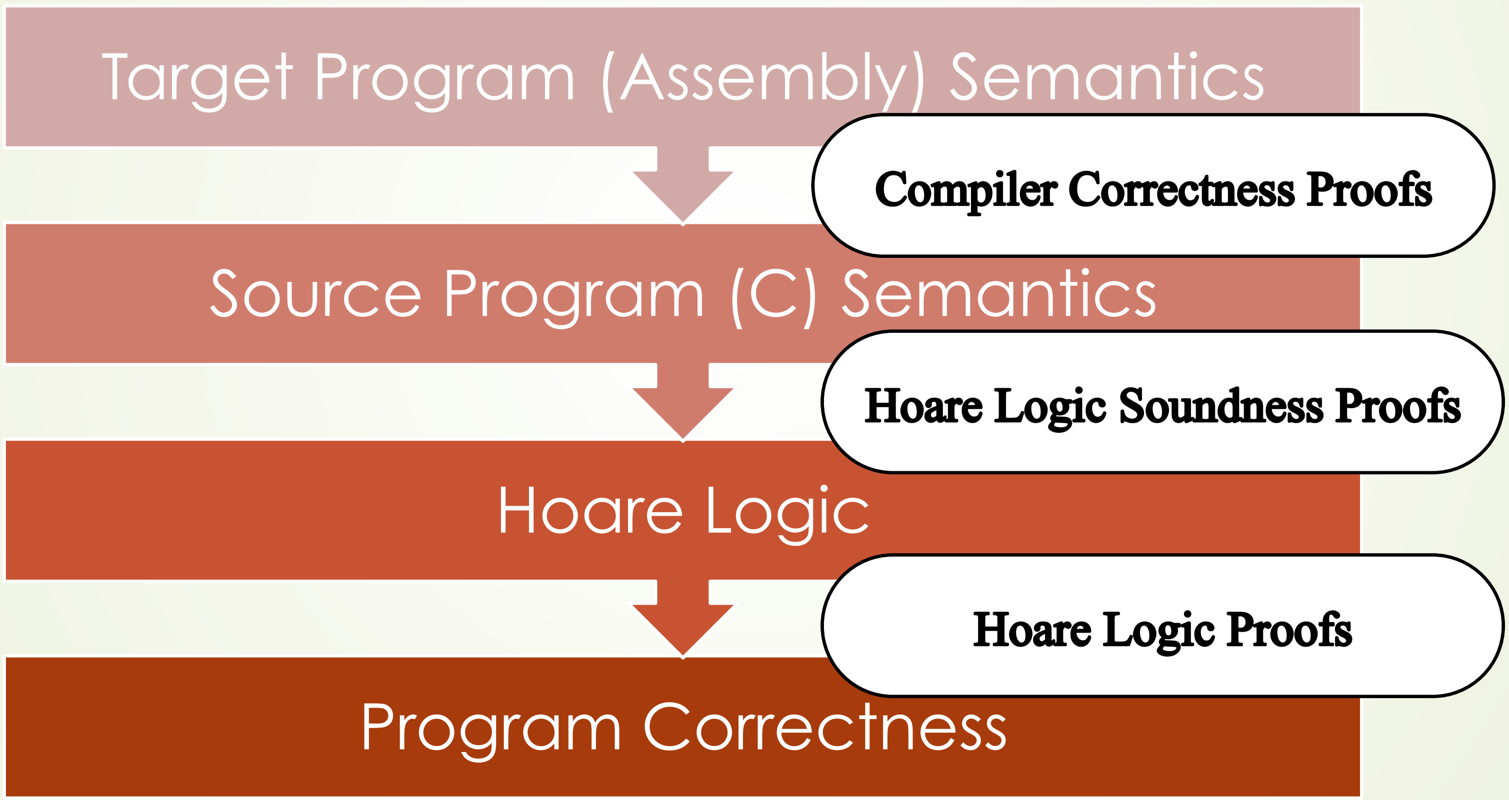
List of forward verification tactics:

- For C singleton commands:
  - Forward, forward_call.
- For control flow commands:
  - **Forward_if**, **forward_loop**, **forward_for_simple_bound**.
- For extracting existential quantifiers:
  - Intros.
- For weakening precondition:
  - Gather_SEP, replace_SEP, sep_apply, Coq's built-in (rewrite, unfold, etc)
- For solving entailments:
  - Entailer!, Exists.

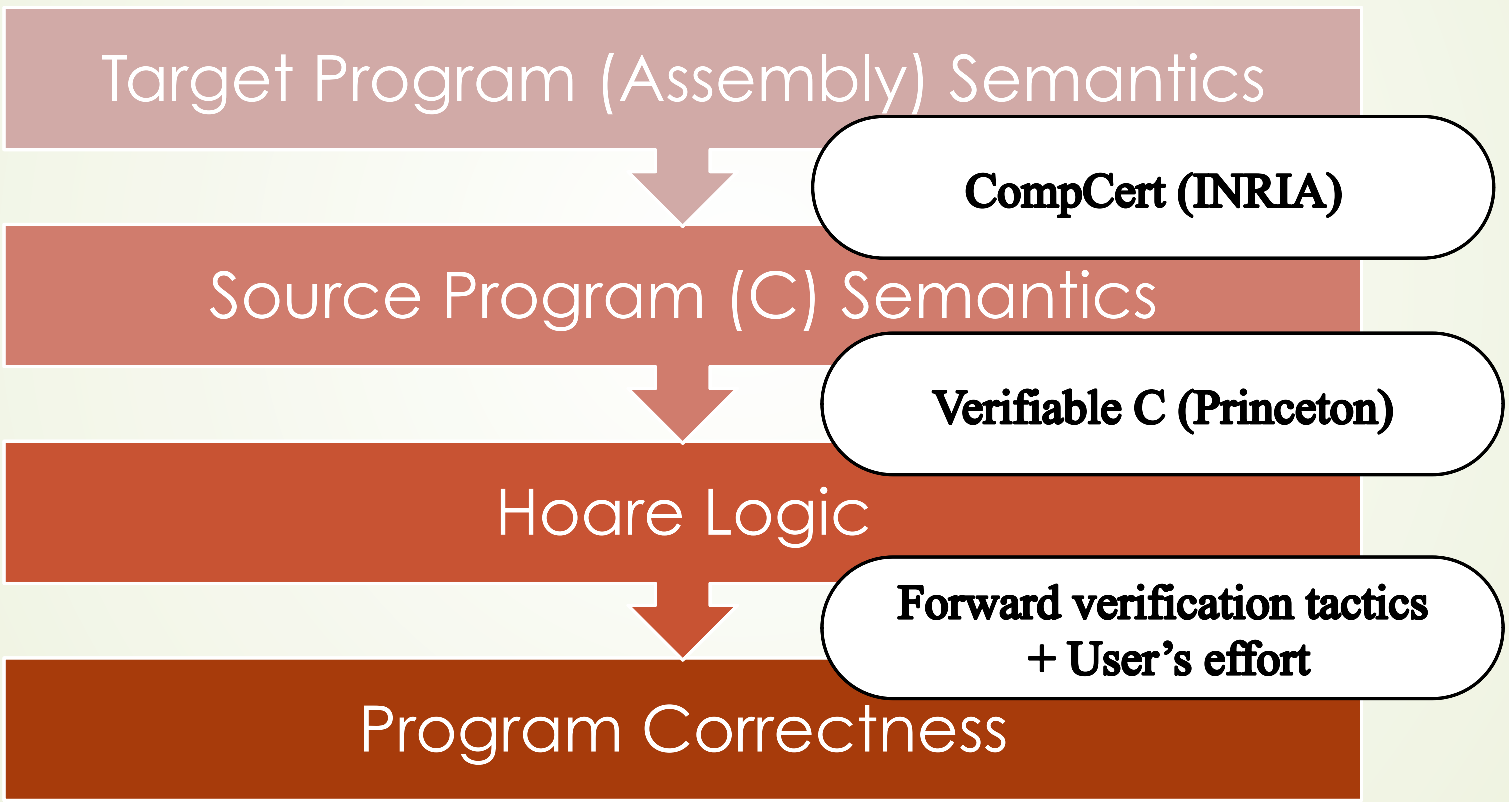
Talk Outline

- What is Verifiable C?
- How to use Verifiable C?
- Verifiable C's soundness.

Verified Software Toolchain --- An overview



Verified Software Toolchain --- An overview



Thank you!