An introduction to Verus

Andrea Lattuada
(and the Verus Team)

Max Planck Institute
for Software Systems (MPI-SWS)

https://verus.rs/

```
fn max_of_three(a: u64, b: u64, c: u64) -> u64 {
  pepr6geiedzyfws@39h71@u9tru438g~b#)5f#wxq~0!6b6~)22c@rhkru6yih5~v(7lkq
  3e4km@0~fgs((n4c0r9noj3ovt)s54x!yq5$o#xaeqguh~it740#hqvp7!)h$itu@$mh)l
  ~f559~bwa9q1yku(~ix#2wtcnt7e7ni5geodt26~zl(@ls((jwtojc9e0lx5a9cm9~w(~@
  675t8(k7gx(i5i0@yvf4q3d0dqoydmnx5husrb5xywcbg)34774bx1gipqu@i3pflrqpv4
  08ic4i(rrb85rea905j$3lc4pai!m$z!@u6i@pe3gonwt395!()9hq!9$p71!ora#24cx8
}
```

```
fn max_of_three(a: u64, b: u64, c: u64) -> u64 {
 pepr6geiedzyfws@39h71@u9tru438g~b#)5f#wxq~0!6b6~)22c@rhkru6yih5~v(7lkq
 3e4km@0~fgs((n4c0r9noj3ovt)s54x!yq5$o#xaeqguh~it740#hqvp7!)h$itu@$mh)l
 ~f559~bwa9q1yku(~ix#2wtcnt7e7ni5geodt26~zl(@ls((jwtojc9e0lx5a9cm9~w(~@
 675t8(k7gx(i5i0@yvf4q3d0dqoydmnx5husrb5xywcbg)34774bx1gipqu@i3pflrqpv4
 08ic4i(rrb85rea905j$3lc4pai!m$z!@u6i@pe3gonwt395!()9hq!9$p71!ora#24cx8
}



#[test] fn test_max_of_three() {
 let test_cases = [
  ((1, 3, 6), 6),
  ((4, 2, 1), 4),
  ((4, 4, 1), 4),

 ];

 for ((a, b, c), r_expected) in test_cases {
  let r_actual = max_of_three(a, b, c);
  assert!(r_actual == r_expected);
}}
```

```rust
fn max_of_three(a: u64, b: u64, c: u64) -> u64 {
 pepr6geiedzyfws@39h71@u9tru438g~b#)5f#wxq~0!6b6~)22c@rhkru6yih5~v(7lkq
 3e4km@0~fgs((n4c0r9noj3ovt)s54x!yq5$o#xaeqguh~it740#hqvp7!)h$itu@$mh)l
 ~f559~bwa9q1yku(~ix#2wtcnt7e7ni5geodt26~zl(@ls((jwtojc9e0lx5a9cm9~w(~@
 675t8(k7gx(i5i0@yvf4q3d0dqoydmnx5husrb5xywcbg)34774bx1gipqu@i3pflrqpv4
 08ic4i(rrb85rea905j$3lc4pai!m$z!@u6i@pe3gonwt395!()9hq!9$p71!ora#24cx8
}
```

```rust
#[test] fn test_max_of_three() {
 let test_cases = [
  ((1, 3, 6), 6),
  ((4, 2, 1), 4),
  ((4, 4, 1), 4),

 ];

 for ((a, b, c), r_expected) in test_cases {
  let r_actual = max_of_three(a, b, c);
  assert!(r_actual == r_expected);
}}
```

```
$cargo test

running 1 test
test test_max_of_three ... ok

test result: ok. 1 passed; ...
```

```rust
fn max_of_three(a: u64, b: u64, c: u64) -> u64 {
  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
  }
}
```

```rust
#[test] fn test_max_of_three() {
  let test_cases = [
    ((1, 3, 6), 6),
    ((4, 2, 1), 4),
    ((4, 4, 1), 4),

  ];

  for ((a, b, c), r_expected) in test_cases {
    let r_actual = max_of_three(a, b, c);
    assert!(r_actual == r_expected);
  }
}
```

```
$cargo test

running 1 test
test test_max_of_three ... ok

test result: ok. 1 passed; ...
```

```rust
fn max_of_three(a: u64, b: u64, c: u64) -> u64 {
  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
  }
}
```

```rust
#[test] fn test_max_of_three() {
  let test_cases = [
    ((1, 3, 6), 6),
    ((4, 2, 1), 4),
    ((4, 4, 1), 4),
    ((3, 4, 1), 4),
  ];

  for ((a, b, c), r_expected) in test_cases {
    let r_actual = max_of_three(a, b, c);
    assert!(r_actual == r_expected);
  }
}
```

```
$cargo test

running 1 test
test test_max_of_three ... FAILED

test result: FAILED. ... 1 failed ...
```

```
fn max_of_three(a: u64, b: u64, c: u64) -> u64
{
  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
  }
}
```

```
fn max_of_three(a: u64, b: u64, c: u64) -> u64



{

  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
}}
```

```
use vstd::prelude::*; verus! {

fn max_of_three(a: u64, b: u64, c: u64) -> u64



{

  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
}}

} // verus!
```

```
use vstd::prelude::*; verus! {

fn max_of_three(a: u64, b: u64, c: u64) -> u64
  ensures
    (r == a || r == b || r == c)
    (r >= a && r >= b && r >= c)
{

  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
}}

} // verus!
```

(r: u64)

unreached()

```
$   verus max-three.rs


error: postcondition not satisfied
  --> max-three.rs:7:1
   |
5  | /    (r == a || r == b || r == c) &&
6  | |    (r >= a && r >= b && r >= c)
   | |_____________________________________- failed this postcondition
7  | / {
8  | |   if a >= b {
9  | |     if a >= c { a } else { if b >= c { unreached() } else { c } }
10 | |   } else {
11 | |     if a >= c { a } else { if b >= c { b } else { c } }
12 | |   }
13 | | }
   | |_^ at the end of the function body
```

```
use vstd::prelude::*; verus! {

fn max_of_three(a: u64, b: u64, c: u64) -> u64                    (r: u64)
  ensures
    (r == a || r == b || r == c)
{   (r >= a && r >= b && r >= c)

  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }      unreached()
  } else {
    if a >= c { a } else { if b >= c { b } else { c } }
}}

} // verus!
```

```
$   verus max-three.rs

verification results:: 1 verified, 0 errors
```

```
use vstd::prelude::*; verus! {

fn max_of_three(a: u64, b: u64, c: u64) -> u64                    (r: u64)
  ensures
    (r == a || r == b || r == c)
    (r >= a && r >= b && r >= c)
{

  if a >= b {
    if a >= c { a } else { if b >= c { unreachable!() } else { c } }      unreached()
  } else {
    if a >= c { a } else { if b = c { b } else { c } }
}}

} // verus!
```

```
#[test] fn test_max_of_three() {
        ...
```

```
$cargo test

running 1 test
test test_max_of_three ... ok

test result: ok. 1 passed; ...
```

SMT-based verification

```
fn max(a: u64, b: u64) -> u64 {
  if a >= b {
     a
  } else {
     b
  }
}

#[test]
fn max_test() {
  let a = 3;
  let b = 4;
  let ret = max(a, b);
  assert!(ret == a || ret == b);
  assert!(ret >= a && ret >= b);
}
```

∀ a,b

∀ a,b

```
(max(a,b) == a || max(a,b) == b) &&
(max(a,b) >= a && max(a,b) >= b)
```

Verus's imperative and functional language

```
use vstd::prelude::*; verus! {

fn max(a: u64, b: u64) -> (r: u64)
  ensures
    r == a || r == b,
    r >= a && r >= b,
{
  if a >= b {
    a
  } else {
    b
  }
}

} // verus!
```

exec mode	spec mode
code to verify	functional (mathematical)
checked for ownership and borrowing	copying always allowed
compiled	erased

demo01.rs, --compile

Function syntax

function mode

function parameters

```
spec fn linear(m: int, b: int, x: int) -> (r: int)
{
  m * x + b
}
```

a rust block is an expression

Modes

exec — default

proof, spec — Ghost

```
exec fn is_positive(m: i64) -> (r: bool) {
  m >= 0
}
```

Ghost

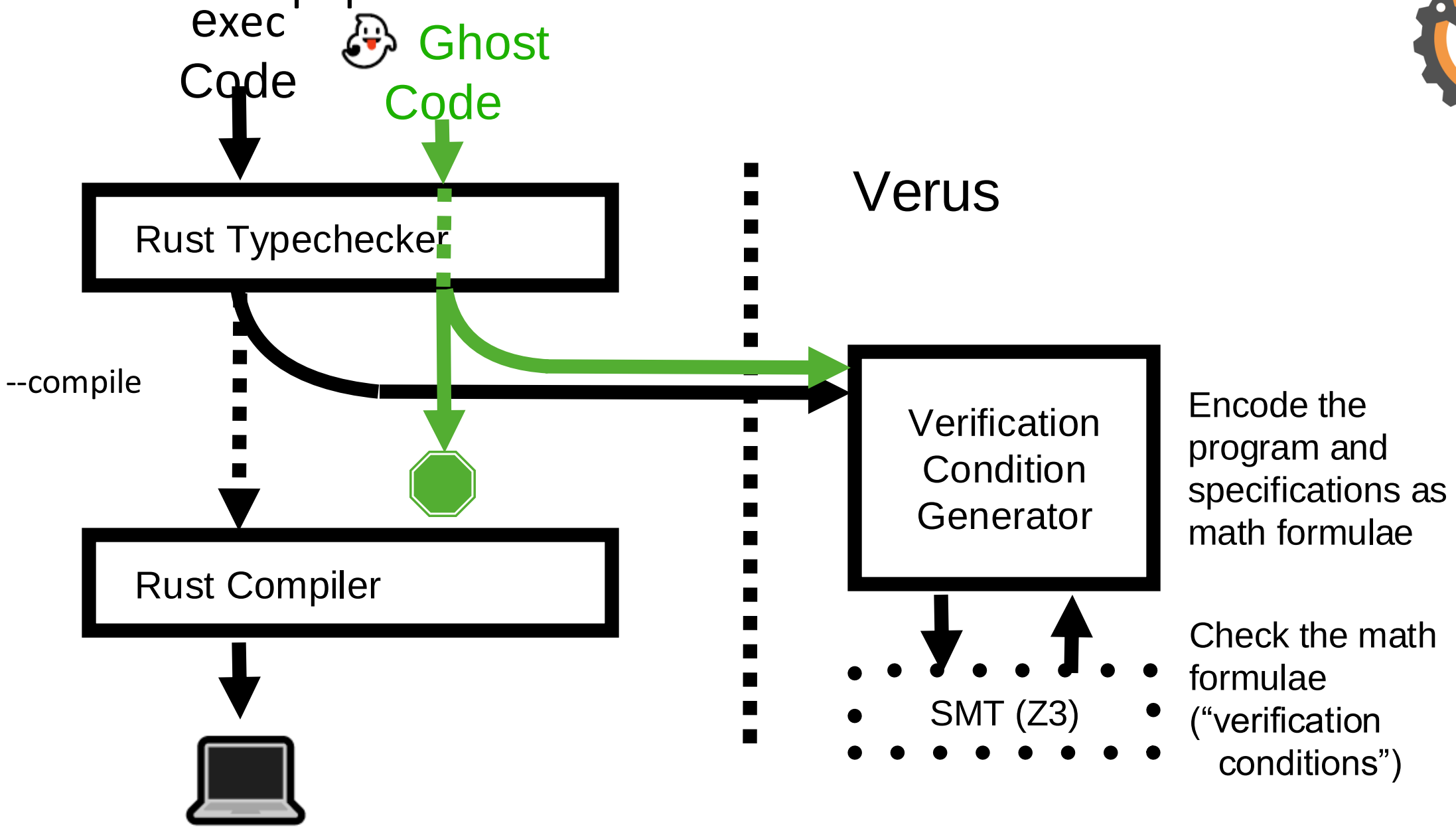
lemma

```
proof fn u64_is_positive(m: u64) {
  assert(m >= 0);
}
```

Ghost

```
spec fn linear(m: int, b: int, x: int) -> (r: int) {
  m * x + b
}
```

The Verus pipeline



Mathematical language (spec code)

- Basic primitives
  int, nat, bool
- Immutable compounds
  - Seq<A>
  - Set<A>, Multiset<A>
  - Map<A, B>
  - datatypes

e.g. Seq<nat>

e.g. Set<Seq<nat>>

e.g. Map<int, bool>

Mathematical language (**spec** code): datatypes ← rust's **struct** and **enum**

```
struct Point {
  x: int,
  y: int,
}
```

```
enum HAlign {
  Left,
  Center,
  Right,
}

enum VAlign {
  Top,
  Middle,
  Bottom,
}
```

```
struct TextAlign {
  hAlign: HAlign,
  vAlign: VAlign,
}
```

demo12.rs

Requires and ensures

- Allowed on proof and exec functions, not on spec functions

Ghost

```
proof fn top_and_left(ta: TextAlign)
  requires
    ta.v_align == VAlign::Top,
    ta.h_align == HAlign::Left,
  ensures
    top_left(ta),
{
}
```

precondition

postcondition

Opacity

- spec functions are “transparent” by default
- proof, exec functions are always “opaque”
  - only requires and ensures visible externally

Datatype methods

exercise_move_point.rs

```
struct Point { x: i64, }

impl Point {
  fn move_x(self, dx: i64) -> (r: Point)
    requires i64::MIN <= self.x + dx < i64::MAX,
    ensures r == self
  {
    Point { x: self.x + dx }
  }

  spec fn center(self) -> bool {
    self.x == 0
  }
}
```

Quantifiers

```
spec fn max_spec(a: nat, b: nat) -> nat {
  if a > b { a } else { b }
}
```

- ∀ a,b

```
(max_spec(a,b) == a || max_spec(a,b) == b) &&
(max_spec(a,b) >= a && max_spec(a,b) >= b)
```

```
forall|a: nat, b: nat|
  (max_spec(a,b) == a || max_spec(a,b) == b) &&
  (max_spec(a,b) >= a && max_spec(a,b) >= b)
```

- ∃ a,b

```
max_spec(a,b) >= a
```

```
exists|a: nat, b: nat| max_spec(a, b) >= a
```

‼ exists (often) needs a “witness”

demo15.rs

Loop invariants

exercise_smallest_elt.rs

- required on loops
- need to be true:
  - on entry
  - after each iteration
    - thus, on exit