Verus github.com/verus-lang/verus

Verified Rust for low-level systems code

Andrea Lattuada, Jon Howell (VMware Research), Tej Chajed (VMware/U-Wisconsin)

Chris Hawblitzel, Ziqiao Zhou, Jay Lorch, Weiteng Chen (Microsoft Research)

Travis Hance, Bryan Parno, Chanhee Cho, Jay Bosamiya, Yi Zhou (Carnegie Mellon University)

Reto Achermann (University of British Columbia)

Isitha Subasinghe, Matthias Brun (ETH Zurich)

Tenzin Low (University of Washington), Tony Zhang (U-Michigan), Xudong Sun (UIUC)

Andrea Lattuada — Researcher, vmware®

Why Verus?

Verification of systems software

to have strong correctness guarantees

- increase software reliability
- avoid data loss, unavailability, bad consequences

Why Verus?

Verification of systems software

to have strong correctness guarantees

- increase software reliability
- avoid data loss, unavailability, bad consequences

Practical verification for systems software

- use verification in development process
- make the verified artifact maintainable

Tests in Rust

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

```rust
fn max_test_1() {
    let x = 4;
    let y = 4;
    let ret = max(x, y);
    assert!(ret == 4);
}

fn max_test_2() {
    let x = 4;
    let y = 3;
    let ret = max(x, y);
    assert_eq!(ret, 4);
}
```

```rust
fn max_test_3() {
    let x = 4;
    let y = 3;
    let ret = max(x, y);
    assert!(ret == x || ret == y);
    assert!(ret >= x && ret >= y);
}

fn main() {
    max_test_1();
    // max_test_2();
    max_test_3();
    println!("success");
}
```

Static check (verification) in Verus

```
use vstd::prelude::*;

verus! {

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

}
```

Why Verus?

Languages for systems software verification

Why Verus?

Languages for systems software verification

Why Verus?

Languages for systems software verification

Rust is efficient
and predictable (like C)

imperative, no GC, in-place
mutation, zero-cost abstractions,
low-level operations (unsafe)

Why Verus?

Languages for systems software verification

(Safe) Rust is close to a functional programming language

ADTs, pattern matching, restricted side-effects

Rust is efficient and predictable (like C)

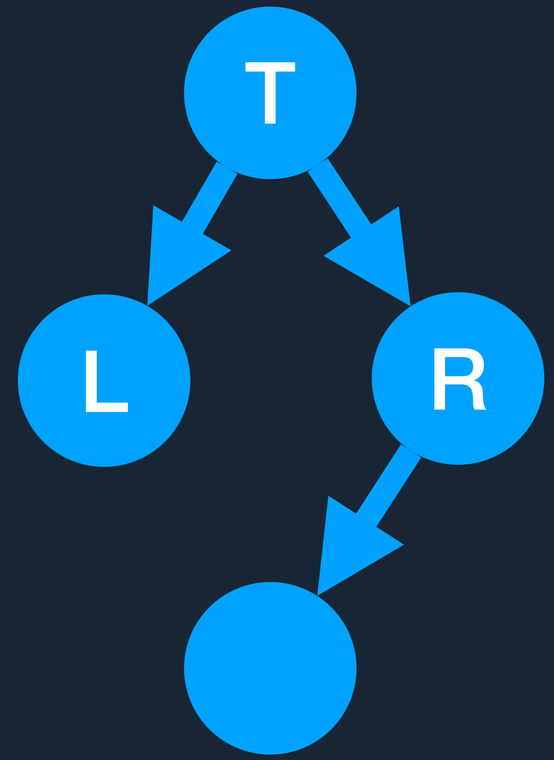
imperative, no GC, in-place mutation, zero-cost abstractions, low-level operations (unsafe)

Why Rust?

Rust enforces single ownership by default

Rust's "linear type system"

```
struct Node {
  left: Option<Box<Node>>,
  right: Option<Box<Node>>,
  value: u64,
}
```

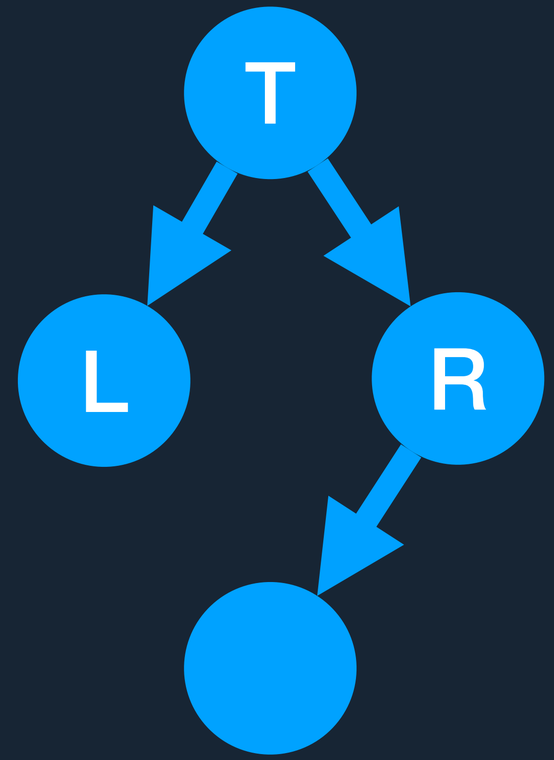


Why Rust?

Rust enforces single ownership by default

Rust's "linear type system"

```
struct Node {
  left: Option<Box<Node>>,
  right: Option<Box<Node>>,
  value: u64,
}

fn bad(mut tree: Node) -> Option<()> {
    // ...
    tree.left.as_mut()?.right = tree.right.as_mut()?.left;
    // ...
}
```

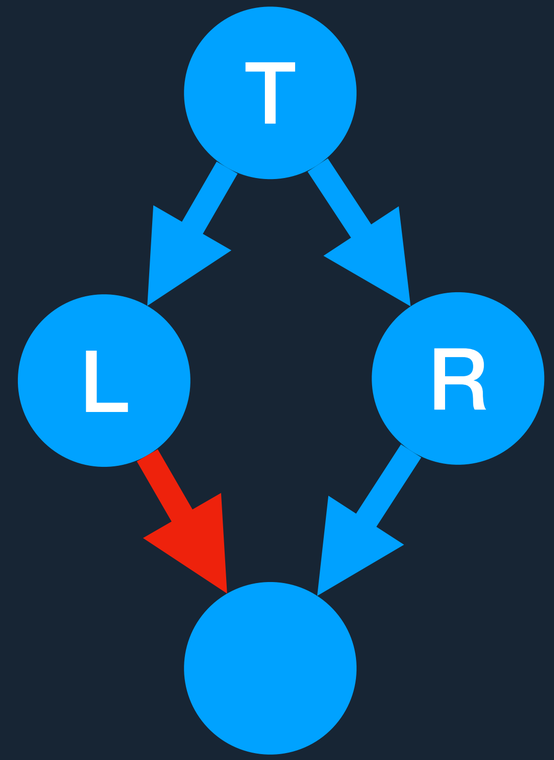


Why Rust?

Rust enforces single ownership by default

Rust's "linear type system"

```
struct Node {
  left: Option<Box<Node>>,
  right: Option<Box<Node>>,
  value: u64,
}

fn bad(mut tree: Node) -> Option<()> {
    // ...
    tree.left.as_mut()?.right = tree.right.as_mut()?.left;
    // ...
}
```

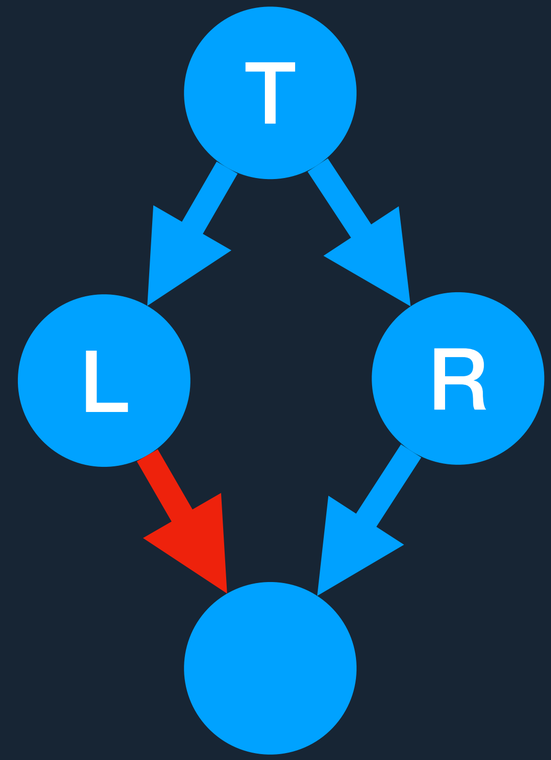


Why Rust?

Rust enforces single ownership by default

Rust's "linear type system"

```
struct Node {
  left: Option<Box<Node>>,
  right: Option<Box<Node>>,
  value: u64,
}


fn bad(mut tree: Node) -> Option<()> {
    // ...
    tree.left.as_mut()?.right = tree.right.as_mut()?.left;
    // ...                          ^^^^^^^^^^^^^^^^^^^^^^^^
}
```




```
error[E0507]: cannot move out of a mutable reference
```

Why Rust?

Rust's *linear* type system

tracks ownership, prevents aliased mutable references

- no data races
- no use-after-free

Why Rust?

Rust's *linear* type system

tracks ownership, prevents aliased mutable references

- no data races
- no use-after-free

Aliasing is a challenge in functional verification

- requires explicit handling of the heap
- → rely on Rust's linear type system instead

Why Rust?

https://doi.org/10.1145/3360573

Rust's *linear*
tracks owner
no data rac
no use-afte

Leveraging Rust Types for Modular Specification and Verification

VYTAUTAS ASTRAUSKAS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
FEDERICO POLI, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, ETH Zurich, Switzerland

Aliasing is a challenge in functional verification
requires explicit handling of the heap
→ rely on Rust's linear type system instead

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount, ...
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}
```

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount, ...
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}
```

unique mutable references

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount,
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}
```

unique mutable references

only true if `orig` and `dest` are not aliases

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount,
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}

fn main(mut account: Account)
  // requires account.balance >= 100
{
  let mut account = account;
  transfer(&mut account, &mut account, 100);
}
```

unique mutable references

only true if `orig` and `dest` are not aliases

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount, ...
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}

fn main(mut account: Account)
  // requires account.balance >= 100
{
  let mut account = account;
  transfer(&mut account, &mut account, 100);
}
```

unique mutable references

only true if `orig` and `dest` are not aliases

aliased

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount,
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}
```

unique mutable references

only true if `orig` and `dest` are not aliases

```
error[E0499]: cannot borrow `account` as mutable more than once at a time
fn main(mut account: Account)
  // requires account.balance >= 100
{
  let mut account = account;
  transfer(&mut account, &mut account, 100);
  --------- ------------  ^^^^^^^^^^^^ second mutable borrow occurs here
  |         |
  |         first mutable borrow occurs here
  first borrow later used by call
```

Rust prevents aliased mutable references

```
pub struct Account { pub balance: u64 }

pub fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  // requires
  //   old(orig).balance >= amount,
  // ensures
  //   dest.balance == old(dest).balance + amount,
  //   orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}
```

unique mutable references

only true if `orig` and `dest` are not aliases ✅

```
error[E0499]: cannot borrow `account` as mutable more than once at a time
fn main(mut account: Account)
  // requires account.balance >= 100
{
  let mut account = account;
  transfer(&mut account, &mut account, 100);
|     -------- ------------  ^^^^^^^^^^^^ second mutable borrow occurs here
|     |        |
|     |        first mutable borrow occurs here
|     first borrow later used by call
```

Why Verus?

Practical verification for systems software

use verification in development process

make the verified artifact maintainable

Why Verus?

Practical verification for systems software

use verification in development process
make the verified artifact maintainable

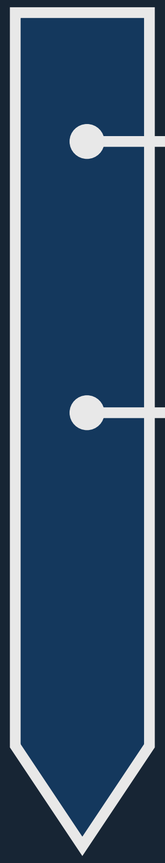
Dafny

imperative language, GC, ~C#,
integrated verification

Why Verus?

Practical verification for systems software

use verification in development process
make the verified artifact maintainable



Dafny

imperative language, GC, ~C#,
integrated verification

Linear Dafny

linear type system for Dafny,
(some) deterministic memory management

Why Verus?

Practical verification for systems software

use verification in development process
make the verified artifact maintainable

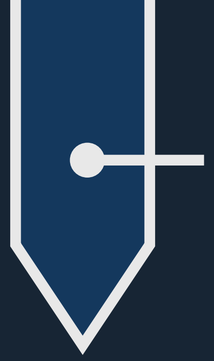
Dafny

imperative language, GC, ~C#,
integrated verification

Linear Dafny

linear type system for Dafny,
(some) deterministic memory management

Rust?

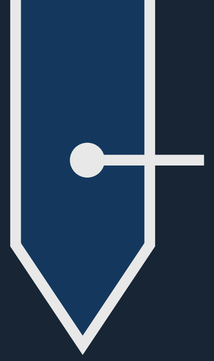


Verus

SMT-based, intrinsic verification of safe[1] Rust[2] code

[1]can do some “unsafe”

[2]a dialect of Rust



Verus

SMT-based, intrinsic verification of safe[1] Rust[2] code

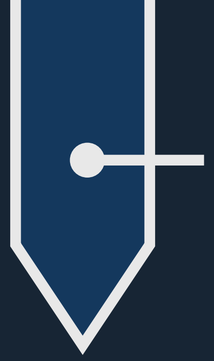
[1]can do some “unsafe”
[2]a dialect of Rust

each function has a Hoare logic specification

Intrinsic verification

```
pub exec fn transfer(orig: &mut Account, dest: &mut Account, amount: u64)
  requires
    old(orig).balance >= amount,
    old(dest).balance + amount < u64::MAX,
  ensures
    dest.balance == old(dest).balance + amount,
    orig.balance == old(orig).balance - amount
{
  orig.balance = orig.balance - amount;
  dest.balance = dest.balance + amount;
}
```

Hoare logic specification



Verus

SMT-based, intrinsic verification of safe[1] Rust[2] code

[1]can do some “unsafe”

[2]a dialect of Rust

SAT Boolean satisfiability problem

SAT Boolean satisfiability problem

```
(x || y) && (!x || y) && (!x)
```

SAT Boolean satisfiability problem

```
∃ x, y :: (x || y) && (!x || y) && (!x)
```

SAT Boolean satisfiability problem

```
∃ x, y :: (x || y) && (!x || y) && (!x)
```

z3

```
x == false
y == true
```

SAT Boolean satisfiability problem

is there an assignment to variables to a boolean formula such that it’s is true

SMT Satisfiability Modulo Theories

generalizes SAT to more complex formulas

real numbers, integers, and/or various data structures,

quantifiers: $\forall, \exists$

SAT Boolean satisfiability problem

is there an assignment to variables to a boolean formula such that it’s is true

SMT Satisfiability Modulo Theories

generalizes SAT to more complex formulas

real numbers, integers, and/or various data structures,

quantifiers: ∀, ∃

∀a, b :: a * b == b * a

SMT-based verification

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

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

SMT-based verification

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

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

∀ a,b

SMT-based verification

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

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

∀ a,b

∀ a,b

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

SMT-based verification

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

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

$\forall$ a,b

$\forall$ a,b

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

$!\ \exists$ a,b $!$

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

SMT-based verification

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

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

∀ a,b

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

! ∃ a,b ! (max(a,b) == a || max(a,b) == b) &&
(max(a,b) >= a && max(a,b) >= b)

↓ z3

SMT-based verification

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

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

∀ a,b

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

! ∃ a,b ! (max(a,b) == a || max(a,b) == b) &&
(max(a,b) >= a && max(a,b) >= b)

⬇ z3

sat/unknown → failure

SMT-based verification

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

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

∀ a,b

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

! ∃ a,b ! (max(a,b) == a || max(a,b) == b) &&
(max(a,b) >= a && max(a,b) >= b)

⬇ z3

sat/unknown → failure

unsat → ∀ a,b proof

SMT-based verification

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

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

∀ a,b

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

! ∃ a,b ! (max(a,b) == a || max(a,b) == b) &&
(max(a,b) >= a && max(a,b) >= b)

z3

sat/unknown → failure

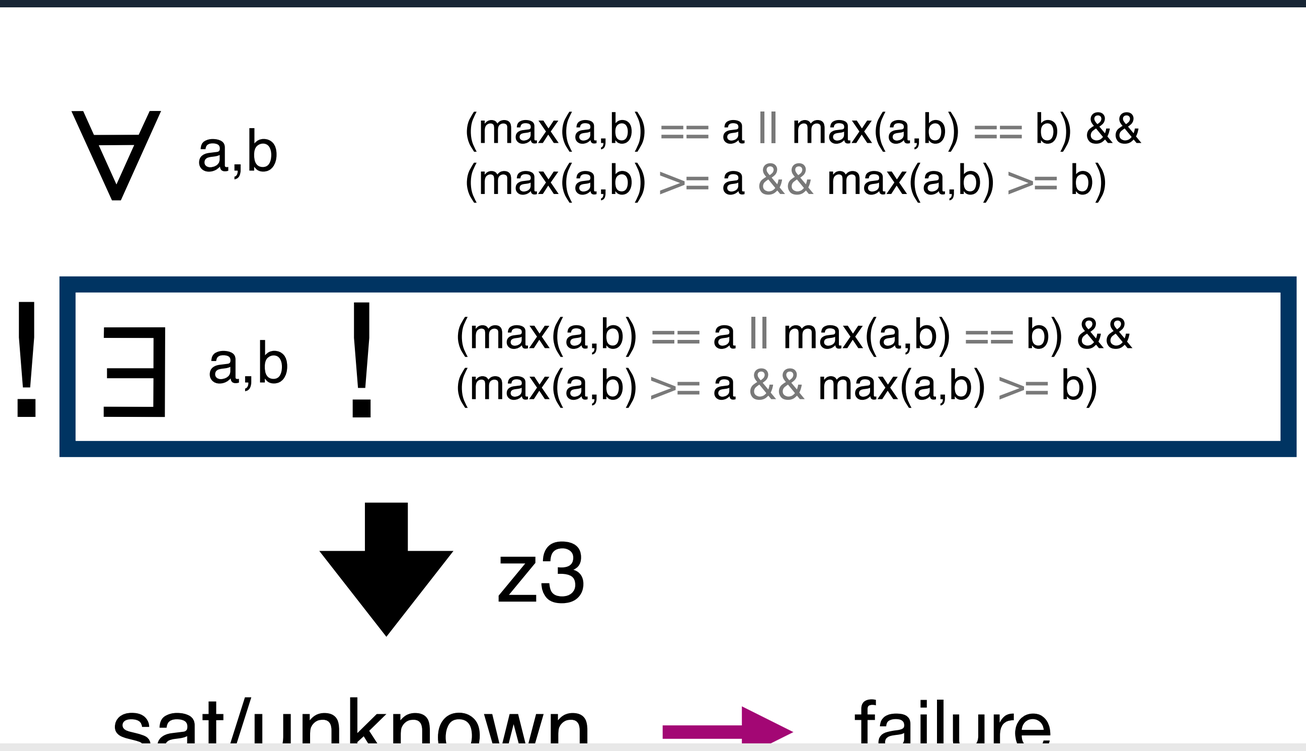
unsat → ∀ a,b proof

??? → ???

SMT-based verification

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

#[test]
fn max_test() {
    let x = 4;
    let y = 3;
    let ret = max(x, y);
    assert(ret == 4 && ret >= y);
}
```

∀ a,b



Verus prioritizes efficient encoding to SMT “queries”

- verification performance

SMT-based verification

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

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

∀ a,b

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

! ∃ a,b ! (max(a,b) == a || max(a,b) == b) &&
(max(a,b) >= a && max(a,b) >= b)

z3

sat/unknown → failure

unsat → ∀ a,b proof

??? → ???

Verifying code with Verus

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}

proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
   if i < 2 && j < 2 {
   } else if i == j {
   } else if i == j - 1 {
       reveal_with_fuel(fibo, 2);
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
   } else {
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
       lemma_fibo_is_monotonic(i, (j - 2) as nat);
   }
}
```

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}

proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
   if i < 2 && j < 2 {
   } else if i == j {
   } else if i == j - 1 {
       reveal_with_fuel(fibo, 2);
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
   } else {
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
       lemma_fibo_is_monotonic(i, (j - 2) as nat);
   }
}
```

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```

exec mode

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}

proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
   if i < 2 && j < 2 {
   } else if i == j {
   } else if i == j - 1 {
       reveal_with_fuel(fibo, 2);
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
   } else {
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
       lemma_fibo_is_monotonic(i, (j - 2) as nat);
   }
}
```

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```

proof mode exec mode

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}
```

```
proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
   if i < 2 && j < 2 {
   } else if i == j {
   } else if i == j - 1 {
       reveal_with_fuel(fibo, 2);
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
   } else {
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
       lemma_fibo_is_monotonic(i, (j - 2) as nat);
   }
}
```

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```

spec mode proof mode exec mode

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}
```

```
proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
   if i < 2 && j < 2 {
   } else if i == j {
   } else if i == j - 1 {
       reveal_with_fuel(fibo, 2);
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
   } else {
       lemma_fibo_is_monotonic(i, (j - 1) as nat);
       lemma_fibo_is_monotonic(i, (j - 2) as nat);
   }
}
```

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```

ghost **spec mode**

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}
```

ghost **proof mode**

```
proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
    if i < 2 && j < 2 {
    } else if i == j {
    } else if i == j - 1 {
        reveal_with_fuel(fibo, 2);
        lemma_fibo_is_monotonic(i, (j - 1) as nat);
    } else {
        lemma_fibo_is_monotonic(i, (j - 1) as nat);
        lemma_fibo_is_monotonic(i, (j - 2) as nat);
    }
}
```

exec mode

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```

ghost **spec mode**

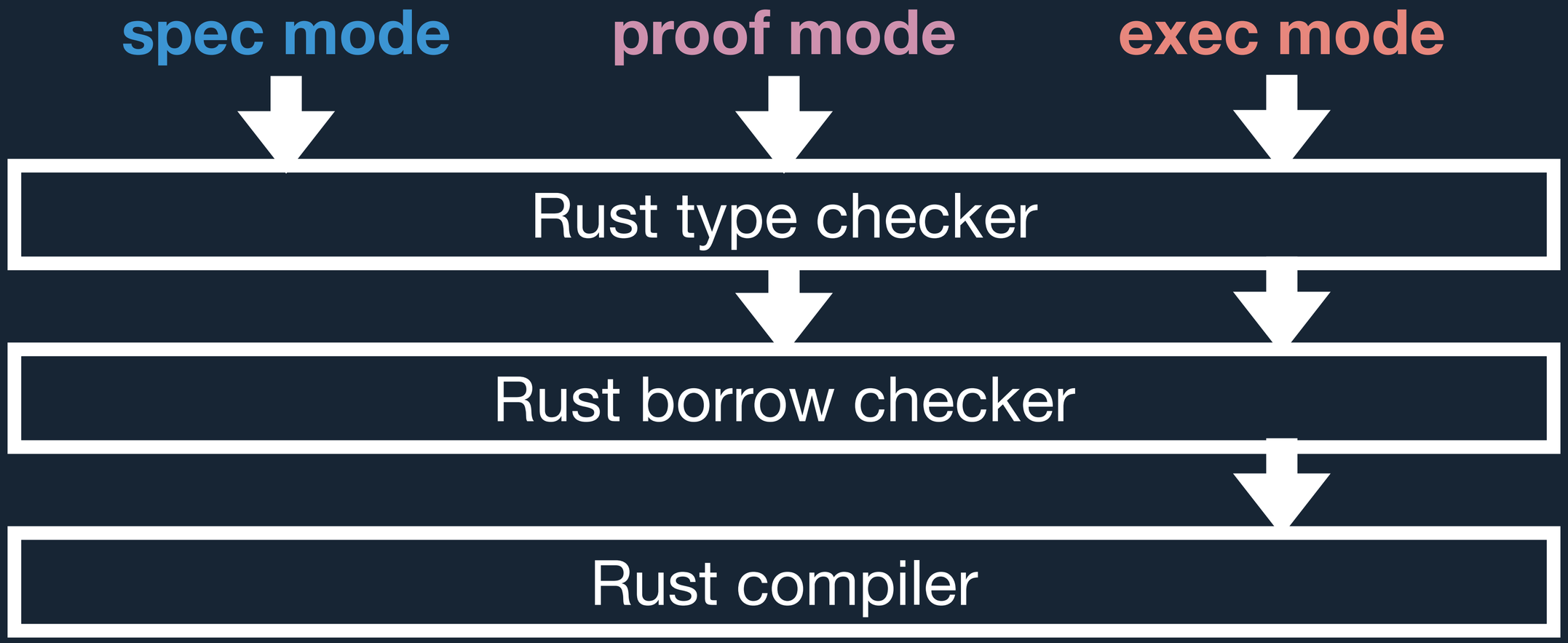
```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 { 0 } else if n == 1 { 1 }
    else { fibo((n - 2) as nat) + fibo((n - 1) as nat) }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}
```

ghost **proof mode**

```
proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i
{
    if i < 2 && j < 2 {
    } else if i == j {
    } else if i == j - 1 {
        reveal_with_fuel(fibo, 2);
        lemma_fibo_is_monotonic(i, (j - 1) as nat);
    } else {
        lemma_fibo_is_monotonic(i, (j - 1) as nat);
        lemma_fibo_is_monotonic(i, (j - 2) as nat);
    }
}
```

exec mode

```
exec fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat),
    ensures result == fibo(n as nat),
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i <= n,
            fibo_fits_u64(n as nat), fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;
        proof {
            lemma_fibo_is_monotonic(i as nat, n as nat);
        }
        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }
    cur
}
```


spec mode
proof mode
exec mode
Rust type checker
Rust borrow checker
Rust compiler

Verus relies on Rust's type system
but allows going beyond what Rust supports (without UB! 😬)

Rust allows `unsafe` code to break out of linearity

- regular Rust makes assumptions about `unsafe`
- Verus allows access to primitives
  memory- and thread-safe when combined with proof obligations

Verus relies on Rust's type system but allows going beyond what Rust supports (without UB! 😬)

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());
  assert(perm@.value == Option::None);
  pptr.put(Tracked(&mut perm), 5);
  assert(perm@.value == Option::Some(5));

  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

Verus relies on Rust's type system
but allows going beyond what Rust supports (without UB! 😬)

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());
  assert(perm@.value == Option::None);
  pptr.put(Tracked(&mut perm), 5);
  assert(perm@.value == Option::Some(5));

  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

like a *mut u64

Verus relies on Rust's type system but allows going beyond what Rust supports (without UB! 😬)

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());
  assert(perm@.value == Option::None);
  pptr.put(Tracked(&mut perm), 5);
  assert(perm@.value == Option::Some(5));

  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

like a *mut u64

like *pptr = 5;

Verus relies on Rust's type system but allows going beyond what Rust supports (without UB! 😬)

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());
  assert(perm@.value == Option::None);
  pptr.put(Tracked(&mut perm), 5);
  assert(perm@.value == Option::Some(5));

  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

like a *mut u64

like *pptr = 5;

like *pptr = ptr::null();

Verus relies on Rust's type system but allows going beyond what Rust supports (without UB! 😬)

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());
  assert(perm@.value == Option::None);
  pptr.put(Tracked(&mut perm), 5);
  assert(perm@.value == Option::Some(5));

  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

like a *mut u64

```
error: borrow of moved value: `perm`
  --> pptr_verus_edit.rs:12:22
   |
12 |   let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsT
   |                      ^^^^^^^^
...
21 |   proof { transfer_permission(perm); }
   |                               ^^^^
22 |
23 |   pptr.take(Tracked(&mut perm));
   |                     ^^^^^^^^^
```

Verus relies on Rust's type system but allows going beyond what Rust supports using proof mode

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());

  pptr.put(Tracked(&mut perm), 5);


  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

Verus relies on Rust's type system but allows going beyond what Rust supports using ghost proof mode

```
#[verifier(external_body)]
proof fn transfer_permission(tracked p: PointsTo<u64>) { /* ... */ }

#[verifier(external_body)]
fn store_copy_of_pointer(p: PPtr<u64>) { /* ... */ }

fn main() {
  let (pptr, Tracked(mut perm)): (PPtr<u64>, Tracked<PointsTo<u64>>) =
    PPtr::<u64>::empty();

  store_copy_of_pointer(pptr.clone());

  pptr.put(Tracked(&mut perm), 5);

  proof { transfer_permission(perm); }
  pptr.take(Tracked(&mut perm));
}
```

Verus as a pragmatic “toolbox” for verification

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

- non-linear arithmetic
- bit-vector reasoning

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

- non-linear arithmetic
- bit-vector reasoning

shared-memory concurrency support

- by encoding many proof obligations in the Rust type system

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

non-linear arithmetic

bit-vector reasoning

shared-memory concurrency support

by encoding many proof obligations in the Rust type system

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

```
proof fn div_is_smaller(x: nat, y: nat)
    by (nonlinear_arith)
    requires
        y != 0
    ensures
        divide(x, y) <= x,
{
}
```

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

```
proof fn div_is_smaller(x: nat, y: nat)
    by (nonlinear_arith)
    requires
        y != 0
    ensures
        divide(x, y) <= x,
{
}
```

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

```
proof fn div_is_smaller(x: nat, y: nat)
    by (nonlinear_arith)
    requires
        y != 0
    ensures
        divide(x, y) <= x,
{
}

fn mod8_bw(x: u32) -> (ret: u32)
    ensures
        ret == x % 8,
{
    assert(x & 7 == x % 8) by (bit_vector);
    x & 7
}
```

Verus as a pragmatic “toolbox” for verification

efficient encoding to SMT queries ▶ verification performance

custom solvers for sytems problems

```
proof fn div_is_smaller(x: nat, y: nat)
    by (nonlinear_arith)
    requires
        y != 0
    ensures
        divide(x, y) <= x,
{
}

fn mod8_bw(x: u32) -> (ret: u32)
    ensures
        ret == x % 8,
{
    assert(x & 7 == x % 8) by (bit_vector);
    x & 7
}
```

Verus as a pragmatic “toolbox” for verification

Verus as a pragmatic “toolbox” for verification

scale up verification to real-world systems

Verus as a pragmatic “toolbox” for verification

scale up verification to real-world systems

“proving” Verus with systems projects

Verus as a pragmatic “toolbox” for verification

scale up verification to real-world systems

“proving” Verus with systems projects

- Kubernetes controllers [UIUC, VMware]

Verus as a pragmatic “toolbox” for verification

scale up verification to real-world systems

“proving” Verus with systems projects

- Kubernetes controllers [UIUC, VMware]

- kernel page-table code / OS verification [ETH, VMware, UBC]
  [Matthias Brun et al.; hotOS’23]
  https://andrea.lattuada.me/assets/hotos23-beyond-isolation.pdf

Verus as a pragmatic “toolbox” for verification

scale up verification to real-world systems

“proving” Verus with systems projects

- Kubernetes controllers [UIUC, VMware]

- kernel page-table code / OS verification [ETH, VMware, UBC]
  [Matthias Brun et al.; hotOS’23]
  https://andrea.lattuada.me/assets/hotos23-beyond-isolation.pdf

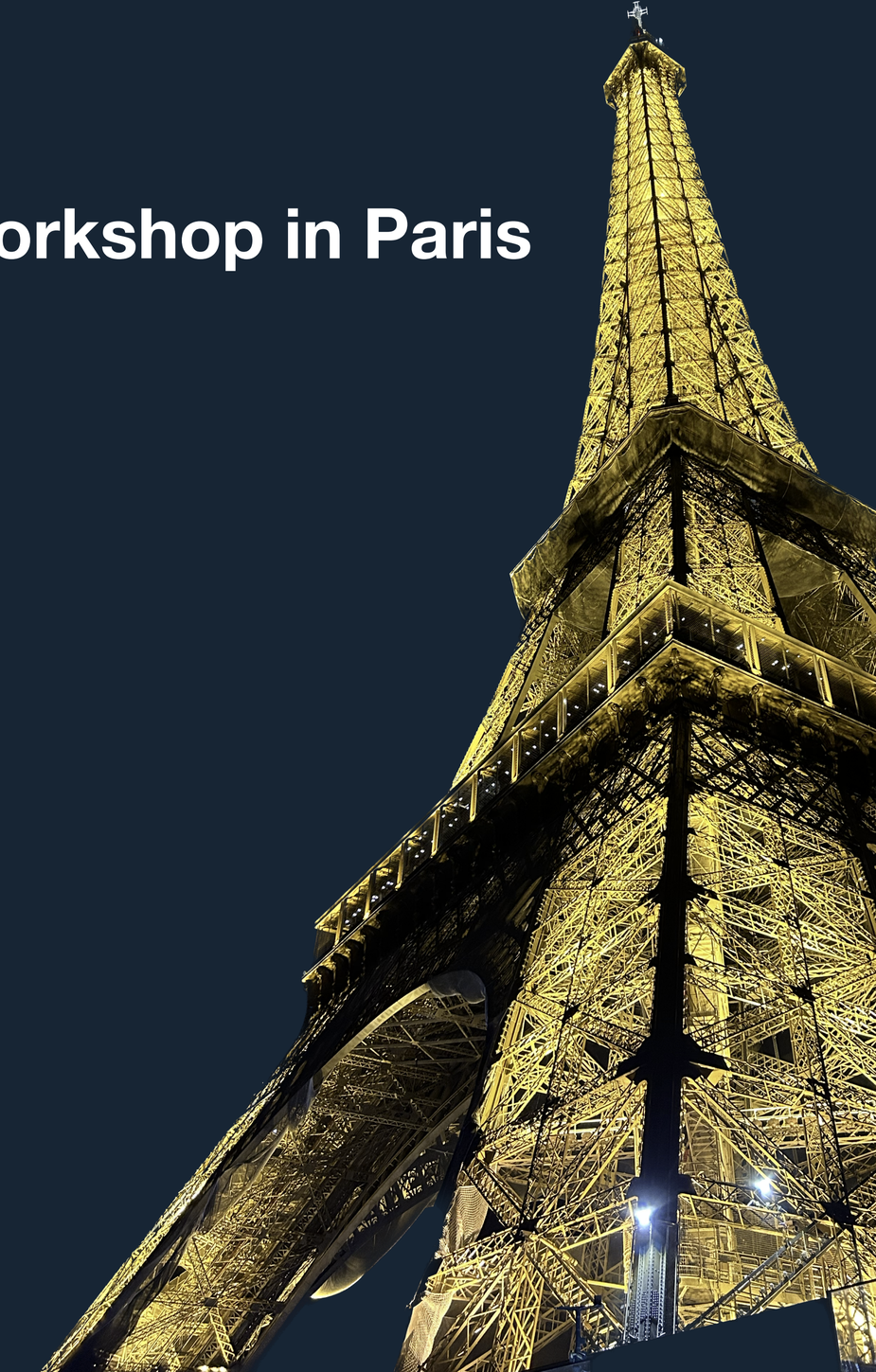
- (more to come) [VMware, MSR, CMU, ETH, …]

Ghost code in rustc

from discussions at the Rust Verification Workshop in Paris

- Verus ported to unmodified rust
- Compiler doesn't understand **ghost code**

▶ Collaborative effort to add ghost code
to the rust compiler
(with Xavier Denis, and many others)

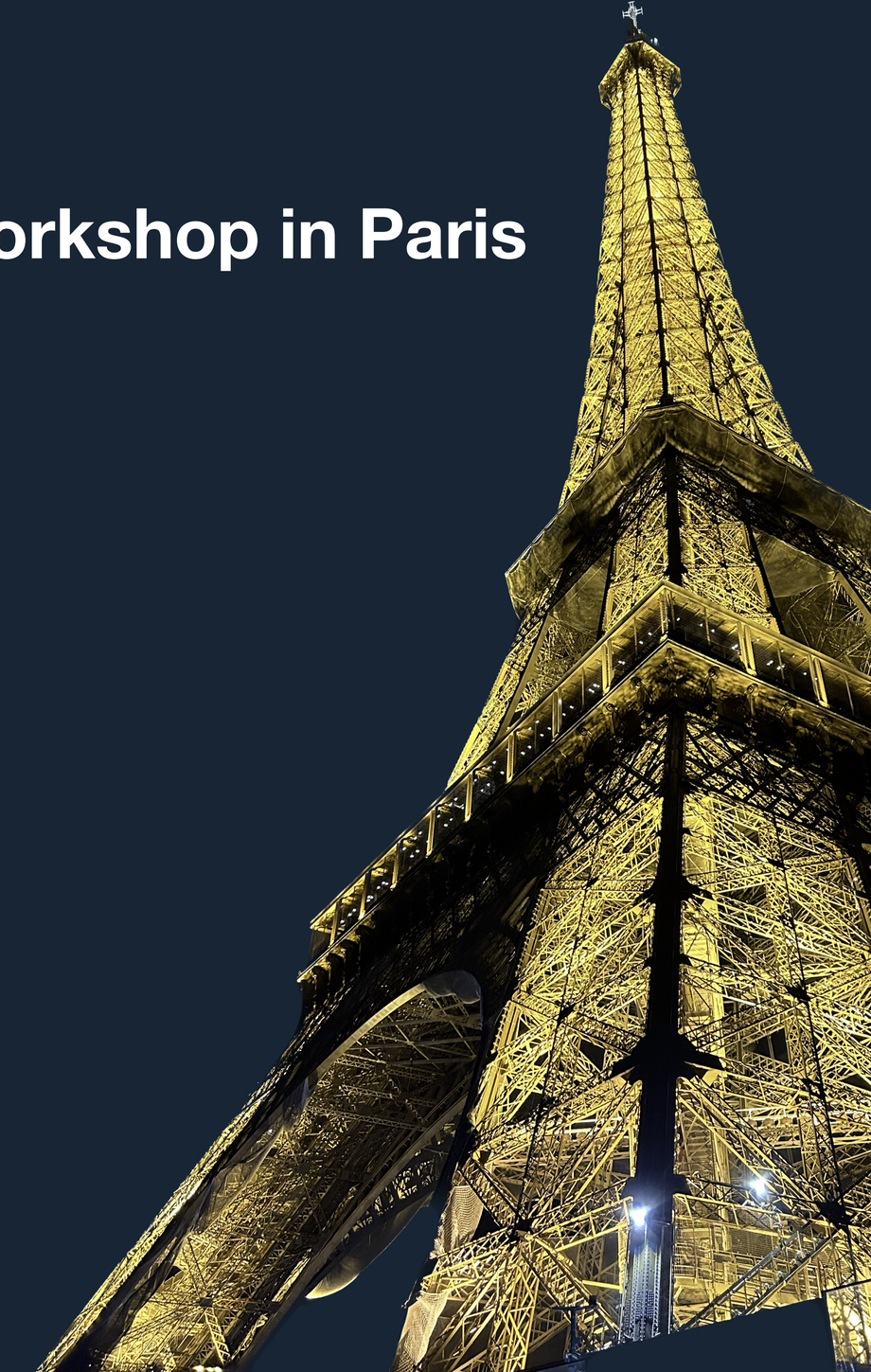


👻 Ghost code in rustc

from discussions at the Rust Verification Workshop in Paris

- Verus ported to unmodified rust
- Compiler doesn't understand **ghost code**

▶ Collaborative effort to add ghost code
to the rust compiler
(with Xavier Denis, and many others)



Verus

github.com/verus-lang/verus

a toolbox for systems verification

Andrea Lattuada, Jon Howell (VMware Research), Tej Chajed (VMware/U-Wisconsin)
Chris Hawblitzel, Ziqiao Zhou, Jay Lorch, Weiteng Chen (Microsoft Research)
Travis Hance, Bryan Parno, Chanhee Cho, Jay Bosamiya, Yi Zhou (Carnegie Mellon University)
Reto Achermann (University of British Columbia)
Isitha Subasinghe, Matthias Brun (ETH Zurich)
Tenzin Low (University of Washington), Tony Zhang (U-Michigan), Xudong Sun (UIUC)

Try the guide!

Try Verus at https://play.verus-lang.org/ [experimental]

Read the paper: https://doi.org/10.1145/3586037

Get in touch! lattuada@vmware.com / andrea@lattuada.me

Andrea Lattuada — Researcher, vmware®