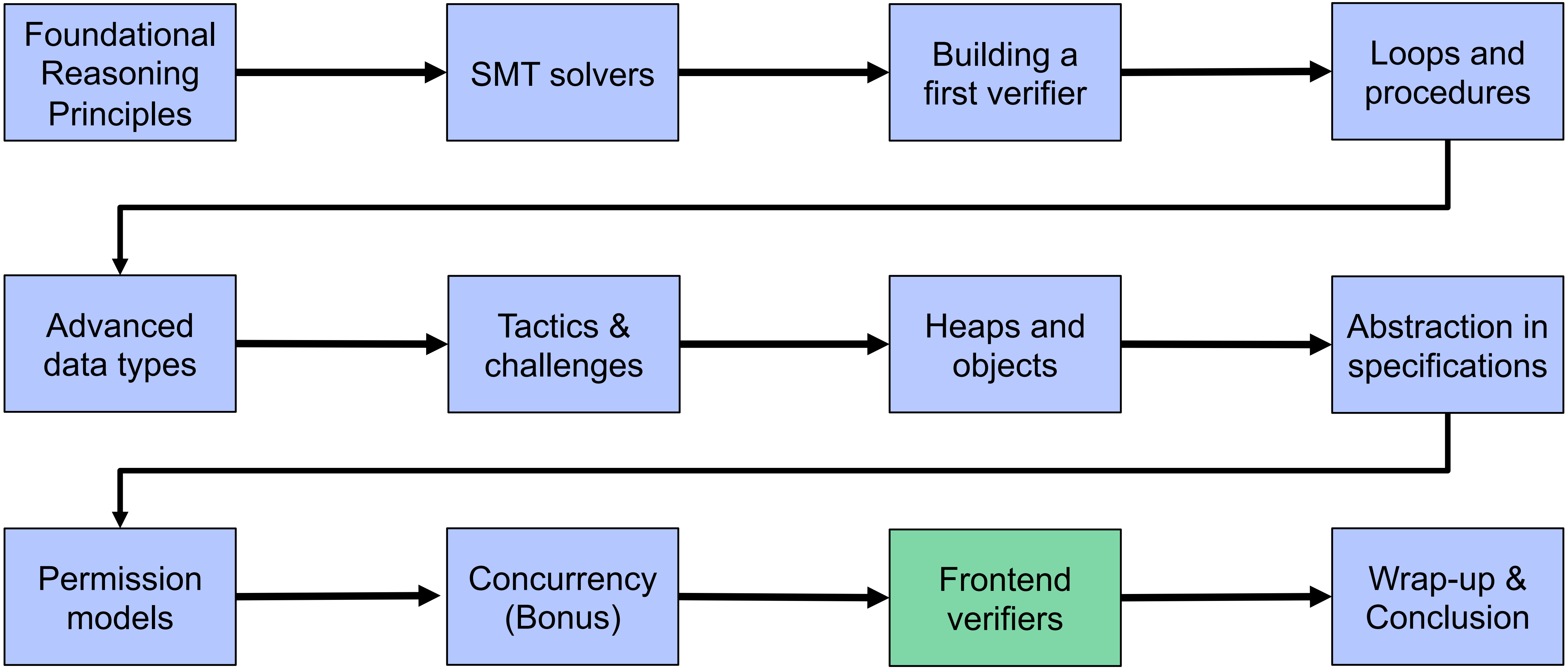
02245 – Module 11

VERIFIER FRONTENDS

Tentative course outline



Source code verifier frontend

Encoding of:

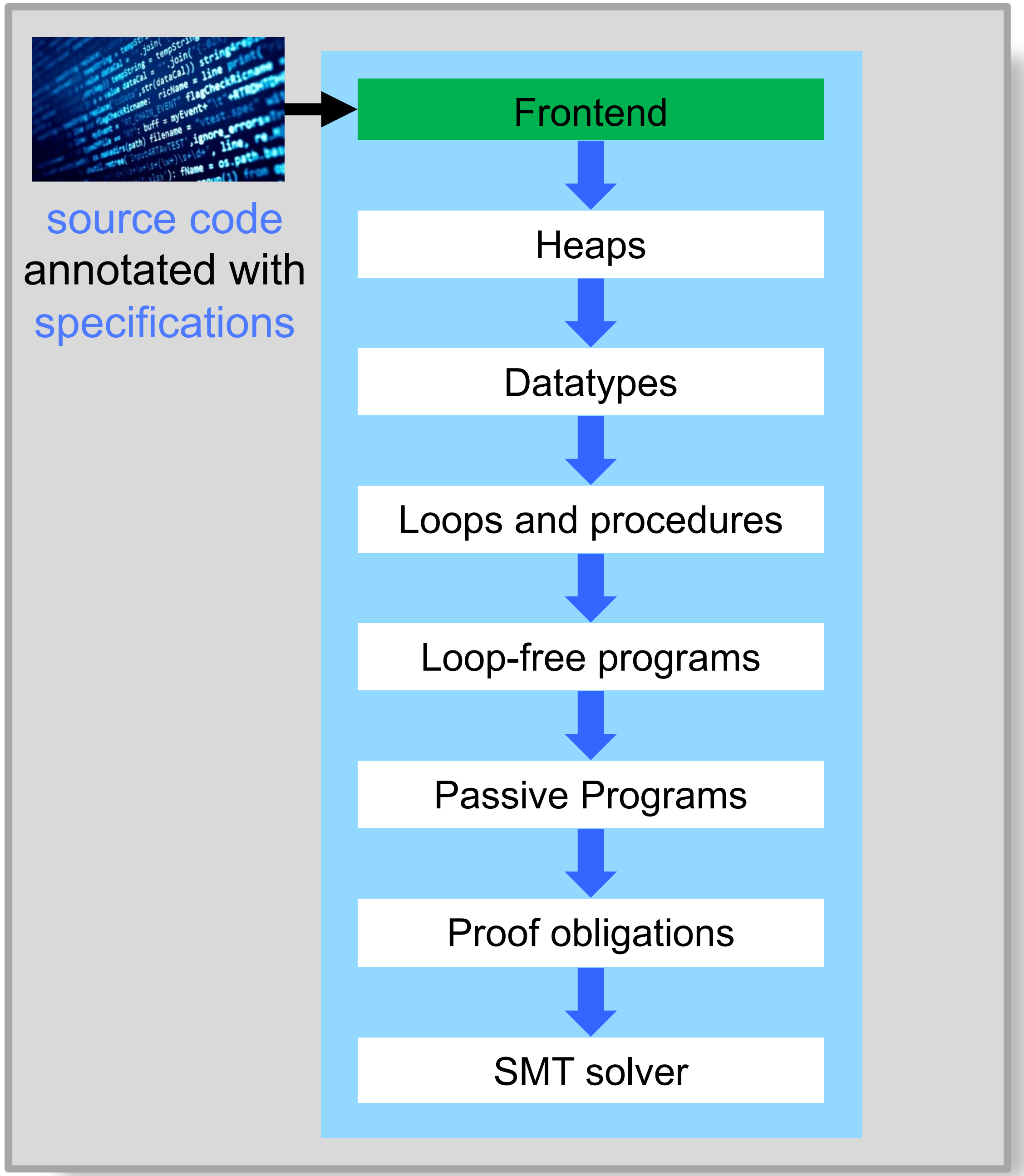
- Program semantics
  - Type system
  - State model
  - Language features (concurrency, etc.)

- Proof obligations that are not checked by default
  - Overflows, termination, well-formedness, etc.

- Specifications and annotations

- Verification logic and proof rules

- Unverified and unverifiable code



Example: Go verification in Gobra

```
requires acc(x) && acc(y)
ensures  acc(x) && acc(y)
ensures  *x == old(*y)
ensures  *y == old(*x)
func swap(x *int, y *int) {
    tmp := *x
    *x = *y
    *y = tmp
}
```

- Go supports pointers to integers
- Parameters can be assigned to
- Locals get initialized by default

```
field val: Int

method swap(x: Ref, y: Ref)
  requires acc(x.val) && acc(y.val)
  ensures  acc(x.val) && acc(y.val)
  ensures  x.val == old(y.val)
  ensures  y.val == old(x.val)
{
  var yLocal: Ref  // declare locals
  var xLocal: Ref

  xLocal := x      // copy parameters
  yLocal := y

  var tmp: Int     // declare tmp
  inhale tmp == 0

  tmp := xLocal.val             // tmp = *x
  xLocal.val := yLocal.val      // *x = *y
  yLocal.val := tmp             // *y = tmp
}
```

Exposing the verification logic

- Gobra's specification and verification technique is very similar to Viper's

- Developers need to use permissions, declare predicates, use unfold and fold statements, etc.

```
requires acc(x) && acc(y)
ensures  acc(x) && acc(y)
ensures  *x == old(*y)
ensures  *y == old(*x)
func swap(x *int, y *int) {
    tmp := *x
    *x = *y
    *y = tmp
}
```

- The overhead for programmers is substantial (both amount and complexity of annotations)

- Many existing verifiers take this approach because it enables modular verification of programs in mainstream languages, including concurrent and heap-manipulating programs

Source code verifiers – design questions

- How to model program semantics in a sound way?
- What is the adequate abstraction level?
  - How much verification logic is exposed? What is checked?
  - What is the required expertise?
  - Trade-off: automation vs. completeness
- How to deal with code at the verification boundary?
  - Libraries, external code
  - Code with unsupported features

gobra : expert verification tool that exposes most capabilities of Viper

Ownership types in Rust

```
fn swap(x: &mut i32, y: &mut i32) {
  let tmp = *x;
  *x = *y;
  *y = tmp;
}

fn client()
{
  let mut a = 17;
  swap(&mut a, &mut a);
}
```

mutable reference (borrow)

```
error[E0499]: cannot borrow `a` as
mutable more than once at a time
  --> .\swap.rs:11:26
   |
11 |     swap(&mut a, &mut a);
   |     ---- ------  ^^^^^^
      second mutable borrow occurs here

error: aborting due to previous error
```

- Rust's type system tracks ownership of memory locations ➔ memory safety
  - Ownership ≈ write permission
  - Moving & borrowing ≈ transfer of fractional permissions
  - Borrow checker ≈ bookkeeping for references with fractional permissions
- Can we leverage this guarantee to simplify verification?

Example: Rust verification in Prusti

```
#[ensures(*x == old(*y) )]
#[ensures(*y == old(*x) )]
fn swap(x: &mut i32, y: &mut i32) {
  let tmp = *x;
  *x = *y;
  *y = tmp;
}
```

P∗rust→∗i

- The overhead for programmers is substantially reduced
  - less complex annotations
  - less annotations overall

- Prusti hooks into the Rust compiler to generate a Viper program
- Prusti generates a "core memory safety proof" completely automatically using the compiler's type information
  - Permissions & predicates
  - Fold / unfold statements
- Users can add functional correctness specifications, by using a slight extension of Rust expressions

Comparison of annotation overhead: zip lists

```
#![feature(box_patterns)]

use prusti_contracts::*;

struct Node {
  elem: i32,
  next: List,
}

enum List {
  Empty,
  More(Box<Node>),
}

impl List {
  #[pure]
  #[ensures(result >= 0)]
  fn len(&self) -> usize {
    match self {
      List::Empty => 0,
      List::More(box node) =>
        1 + node.next.len(),
    }
  }

  #[ensures(result.len() ==
             self.len() + that.len())]
  pub fn zip(&self, that: &List) -> List {
    match self {
      List::Empty => that.cloneList(),
      List::More(box node) => {
        let new_node = Box::new(Node {
          elem: node.elem,
          next: that.zip(&node.next),
        });
        List::More(new_node)
      }
    }
  }
```

```
  #[ensures(result.len() == self.len())]
  pub fn cloneList(& self) -> List {
    match self {
      List::Empty => List::Empty,
      List::More(box node) => {
        let new_node = Box::new(Node {
          elem: node.elem,
          next: node.next.cloneList(),
        });
        List::More(new_node)
      }
    }
  }
}
```

P∗rust→∗i

```
field next: Ref
field elem: Int

predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==> list(this.next))
}

function len(this: Ref): Int
  requires acc(list(this), wildcard)
{
  unfolding acc(list(this), wildcard) in
(this.next == null ? 0 : len(this.next) + 1)
}

method zip(this: Ref, that: Ref)
                             returns (res: Ref)
  requires acc(list(this), 1/2) &&
           acc(list(that), 1/2)
  ensures  acc(list(this), 1/2) &&
           acc(list(that), 1/2)
  ensures  list(res)
  ensures  res != null
  ensures  len(res) == len(this) + len(that)
{
  unfold acc(list(this), 1/2)
  if(this.next == null) {
    res := cloneList(that)
  } else {
    res := new(*)
    res.elem := this.elem
    var rest: Ref
    rest := zip(that, this.next)
    res.next := rest
    fold list(res)
  }
  fold acc(list(this), 1/2)
}
```

```
method cloneList(this: Ref) returns (res: Ref)
  requires acc(list(this), 1/2)
  ensures  acc(list(this), 1/2) && list(res)
  ensures  res != null
  ensures  len(res) == len(this)
{
  res := new(*)
  unfold acc(list(this), 1/2)
  if(this.next == null) {
    res.next := null
  } else {
    var tmp: Ref
    tmp := cloneList(this.next)
    res.elem := this.elem
    res.next := tmp
  }
  fold acc(list(this), 1/2)
  fold list(res)
}
```

This is idealized code; it is **not** the generated code

VIPER

Rust’s ownership system

Ownership rules:

1. Every memory location is owned by a unique variable.
2. A location is disposed of once its owner goes out of scope.
3. Ownership can be moved to another variable if the original owner is not used afterward.

ownership ≈ write permission

moves ≈ permission transfer

```
fn main() {
                                  x: ?       y: ?
  let mut x = Box::new(17);
                                  x: 17      y: ?
  let mut y = x;
                                  x: moved   y: 17
  *y = 42;
                                  x: moved   y: 42
  assert!(*y == 42);
}
                                  x: moved   y: 42
```

Rust's ownership system (II)

Ownership rules:

1. Every memory location is owned by a unique variable.
2. A location is disposed of once its owner runs out of scope.
3. Ownership can be moved to another variable if the original owner is not used afterward.

```
fn main() {
  let mut x = Box::new(17);
  let mut y = x;
  *x = 42;
  assert!(*y == 42);
}
```

```
error[E0382]: use of moved value: `x`
3 |     let mut y = x;
  |                 - value moved here
4 |     *x = 42;
  |     ^^^^^^^ value used here after move
```

Rust's ownership system (III)

Ownership rules:

1. Every memory location is owned by a unique variable.
2. A location is disposed of once its owner runs out of scope.
3. Ownership can be moved to another variable if the original owner is not used afterward.

```
fn create() -> Box<i32> { Box::new(-42) }
```

```
fn foo(x: Box<i32>) -> Box<i32> {
  if *x == i32::MIN {
    x
  } else {
    Box::new(-1 * (*x))
  }
}
```

```
fn bar(x: Box<i32>) { /*...*/ }
```

```
fn main() {
  let mut x = create();
  x = foo(x);
  bar(x);
  assert!(*x == 42); // FAILS
}
```

Viper encoding (simplified sketch)

```
struct S {
  val: i32
}
```

→

```
predicate S(this: Ref) {
  acc(this.val) && i32(this.val)
}
```

```
#[pure]
fn g(a: &i32) -> i32 {
  *a + 17
}
```

→

```
function g(a: Ref): Int
  requires i32(a)
{
  unfolding i32(a) in a.val + 17
}
```

```
fn f(mut x: S, mut y: S) {
  x.val = y.val;

  let z = g(&x.val);

  assert!(z + x.val > 20);
}
```

→

```
method f(x: Ref, y: Ref)
  requires S(x) && S(y)
{
  unfold S(x)
  // ...
  fold S(x)
}
```

computed by simulating Rust's type system

Mutable borrows

→ `02-mut-borrow.rs`
→ `03-mut-borrow.vpr`

Borrowing rules:

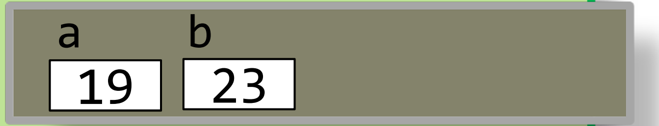
1. Ownership can be temporarily borrowed using references:
   - unique mutable borrow

2. Owned locations cannot be disposed of or mutated while they are borrowed.

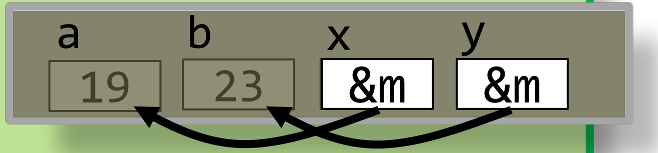
```
fn swap(x: &mut i32, y: &mut i32) {
  let tmp = *x;
  *x = *y;
  *y = tmp;
}
```

```
method swap(x: Ref, y: Ref)
  requires acc(x.val) && acc(y.val)
  ensures acc(x.val) && acc(y.val)
```

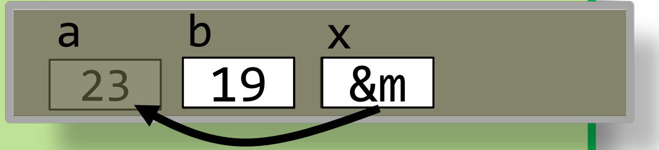
```
fn main() {
  let mut a = 19;
  let mut b = 23;
```




```
  let x = &mut a;
  let y = &mut b;
```




```
  swap(x, y);
```




```
  a = 42; // FAILS

  assert!(*x == 23 && b == 19);

  swap(&mut a, &mut a); // FAILS
}
```

Shared borrows

→ `04-shared-borrow.rs`
→ `05-shared-borrow.vpr`

Borrowing rules:

1. Ownership can be temporarily borrowed using references:
   - unique mutable borrow, xor
   - multiple read-only shared borrows
2. Owned locations cannot be disposed of or mutated while they are borrowed.

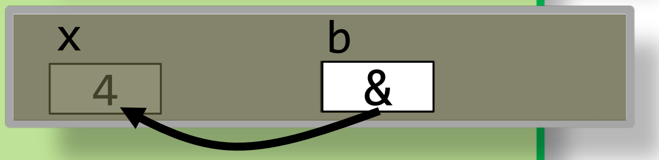
```
fn sum(p: &i32, q: &i32) -> i32 { p+q }
```

shared reference

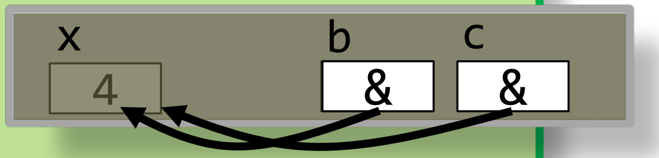
```
method sum(p:Ref, q:Ref) returns (r:Ref)
  requires acc(p.val, 1/2)
             && acc(q.val, 1/2)
  ensures acc(p.val, 1/2)
             && acc(q.val, 1/2)
             && acc(res.val)
```

```
fn main() {
  let mut x = 4;
```


```
  let b = &x;
```




```
  let c = &x;
```




```
  // x = 7 // FAILS

  x = sum(b, c);
```


```
}
```

Many more encoding tasks (omitted)

- Copy types
- Generating fold and unfold statements for calls and loops
- Generics and lifetimes
- Reference-typed fields
- Unsafe Rust code
- …

Prusti Example: Zip Lists

Annotated Rust Code

➔ `06-zip-lists.rs`

`(ca. 75 lines)`

Handwritten Viper Model

➔ `07-zip-lists.vpr`

`(ca 55 lines)`

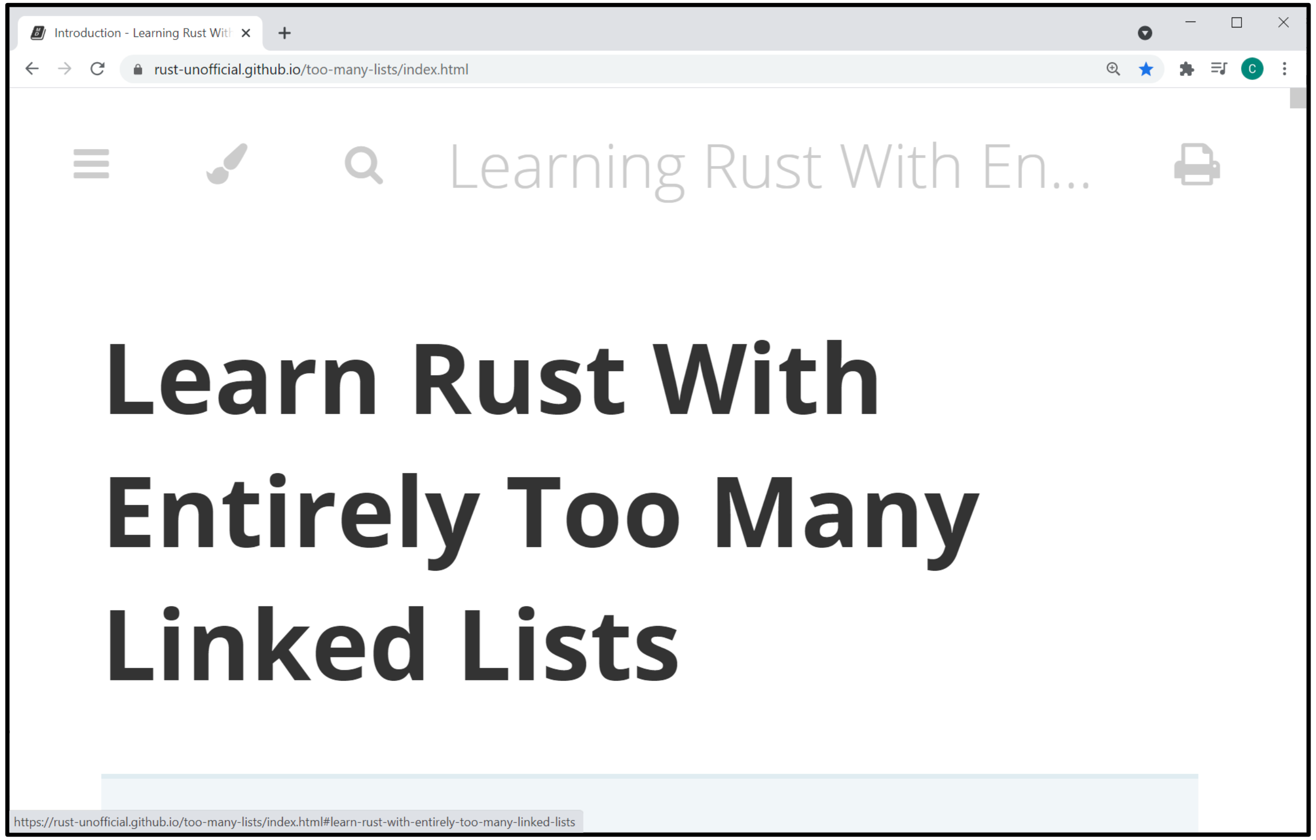
Automated Encoding

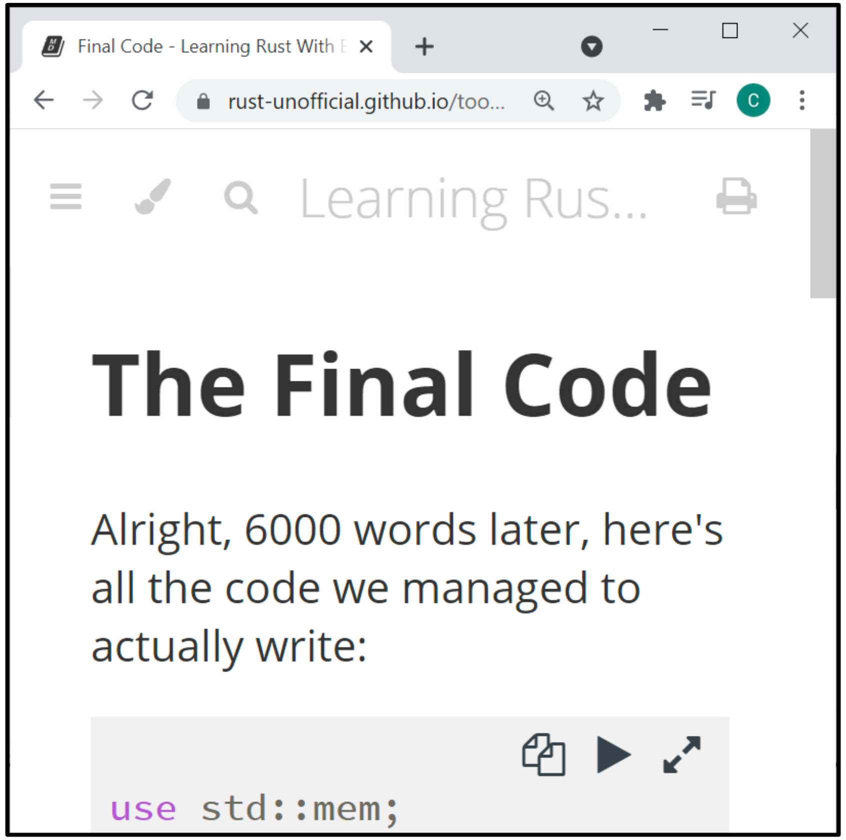
➔ `08-gen-XYZ.vpr`

`(ca. 1’403 lines)`

➔ Z3 applies `ca. 915’469` proof steps in total for verification

Prusti Example: Verified Stack





➔ `09-stack.rs`

Source code verifiers – design questions

- How to model program semantics in a sound way?
- What is the adequate abstraction level?
  - How much verification logic is exposed? What is checked?
  - What is the required expertise?
  - Trade-off: automation vs. completeness
- How to deal with code at the verification boundary?
  - Libraries, external code
  - Code with unsupported features

P∗rust→∗i: lightweight verification tool targeting everyday programmers

gobra: expert verification tool that exposes most capabilities of Viper