Leveraging Rust Types for Modular Specification and Verification

Vytautas Astrauskas Peter Müller Federico Poli Alexander J. Summers

ETH zürich Department of Computer Science

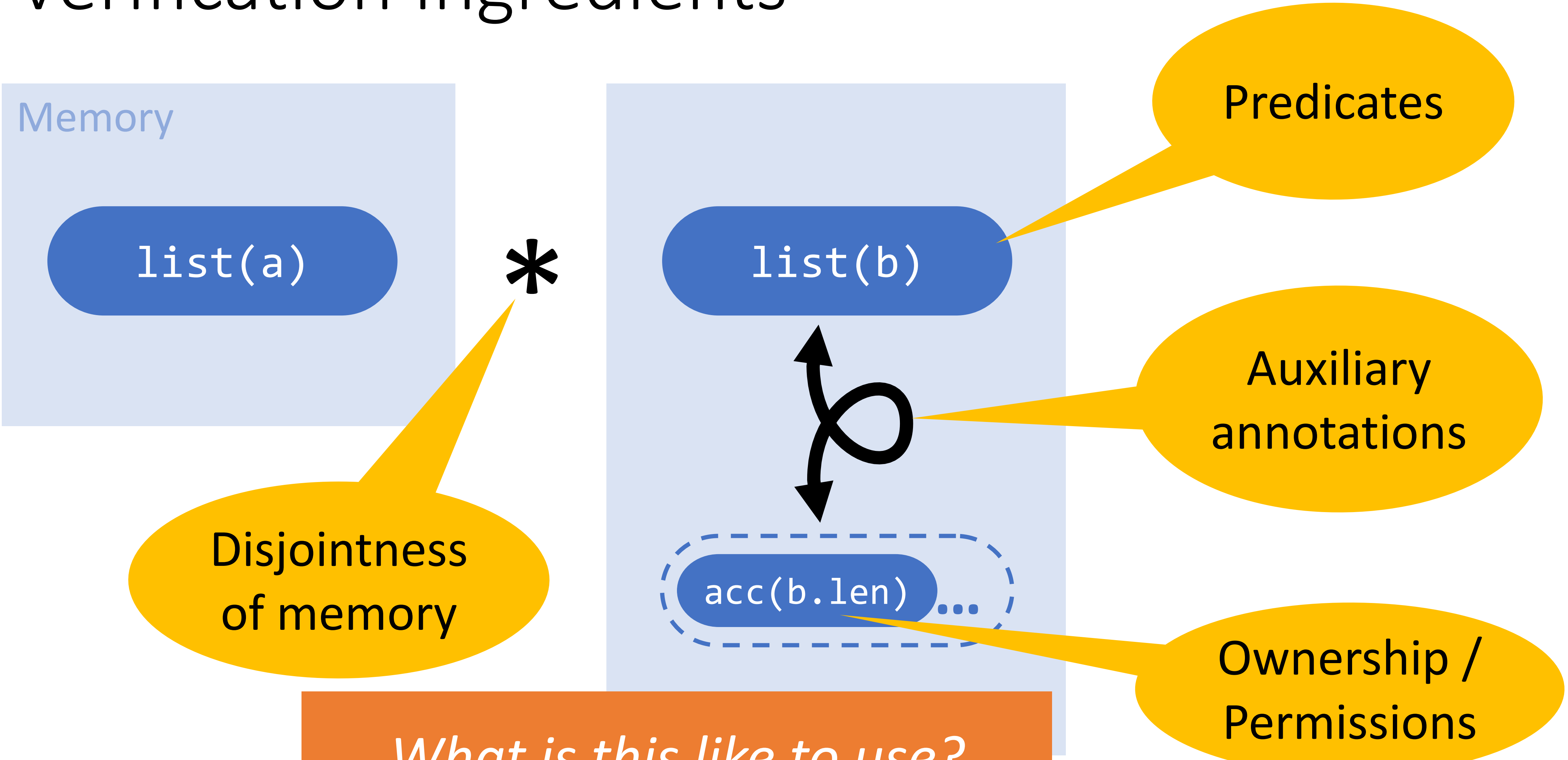
Analogy with C verification

```c
void client(list *a, list *b)
{
  int old_len = b->len;
  append(a, 100);
  assert(b->len == old_len);
}
```

C

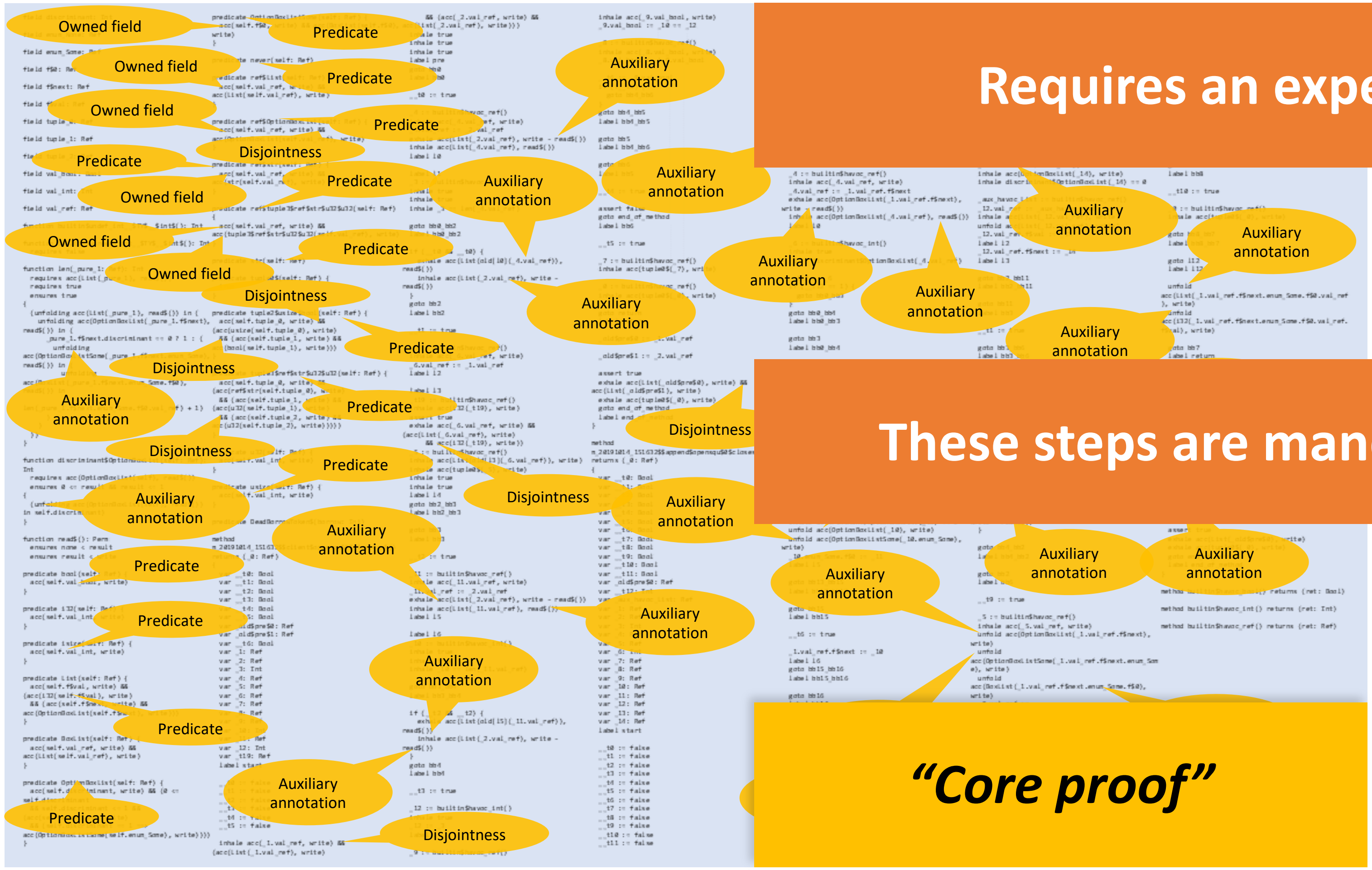
- Functional properties
- Memory Errors
- Aliasing
- Data Races

Verification Ingredients



What is this like to use?

Verification Ingredients at Scale



Requires an expert

These steps are mandatory

"Core proof"

Rust, and its type system

```rust
fn client(a: &mut List, b: &mut List)
{
  let old_len = b.len();
  append(a, 100);
  assert!(b.len() == old_len);
}
```

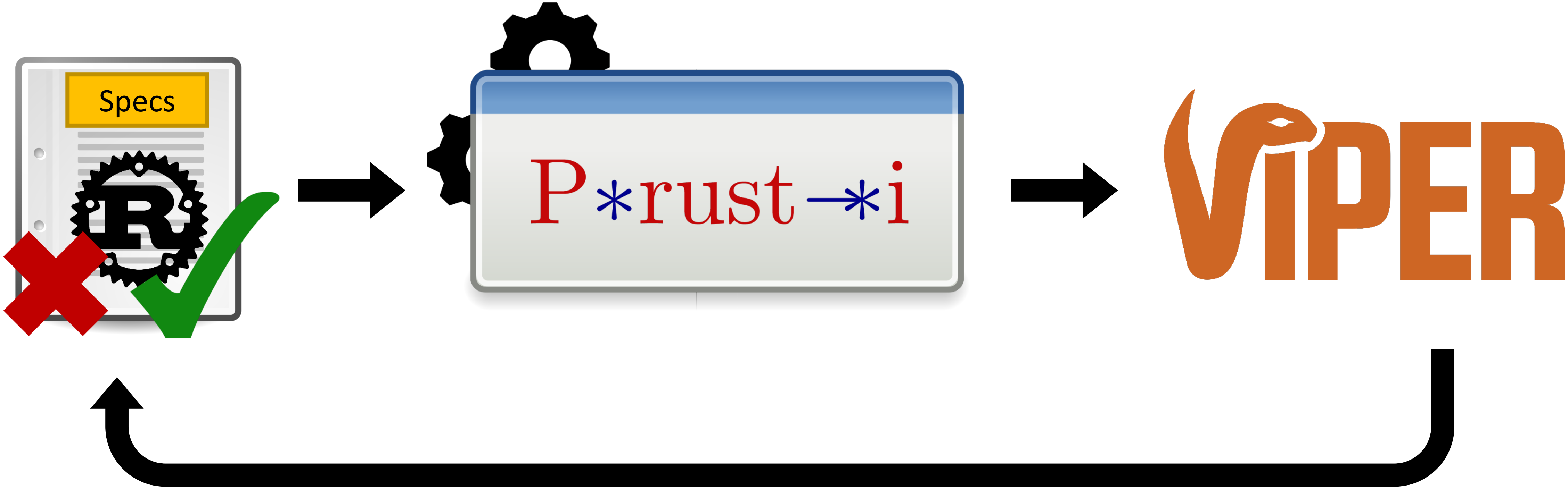
Rust

Can we exploit this type system for *verification*?

- Functional properties
- No Memory Errors
- Controlled Aliasing
- No Data Races

What would we like?

Prusti: An Overview



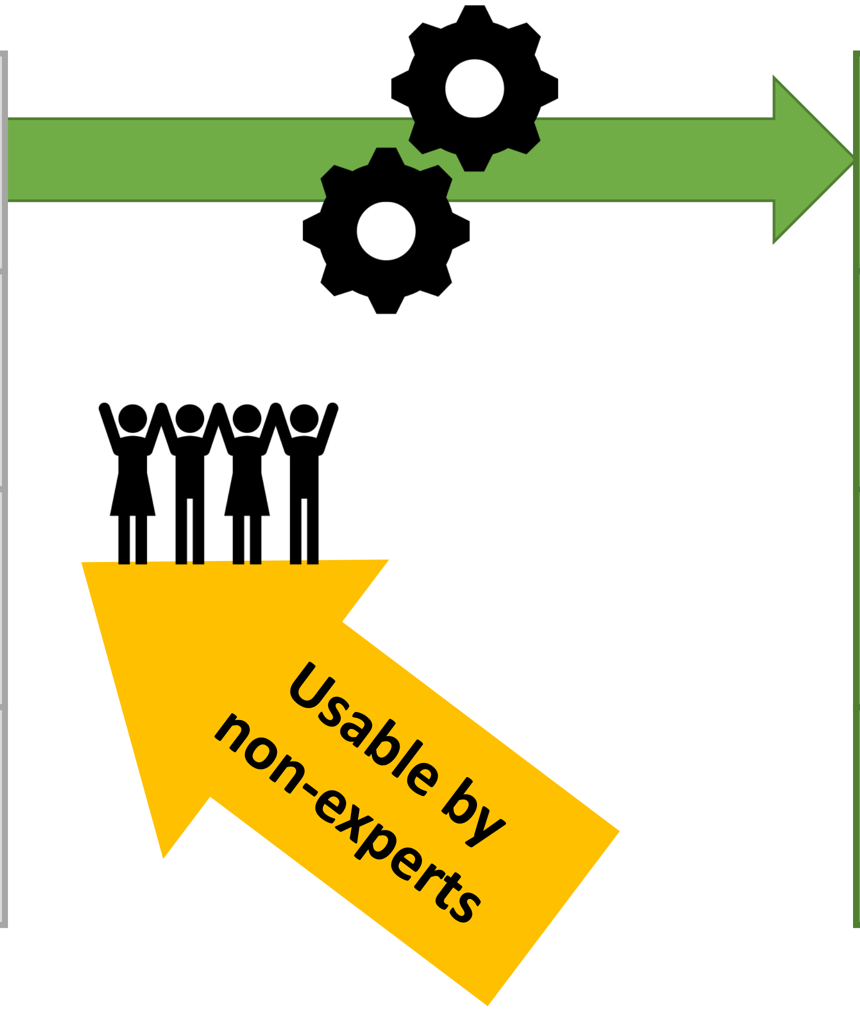
Leveraging Rust Types for Modular Specification and Verification

To appear at OOPSLA 2019, Athens, Greece (next week)

The Prusti Approach

Rust

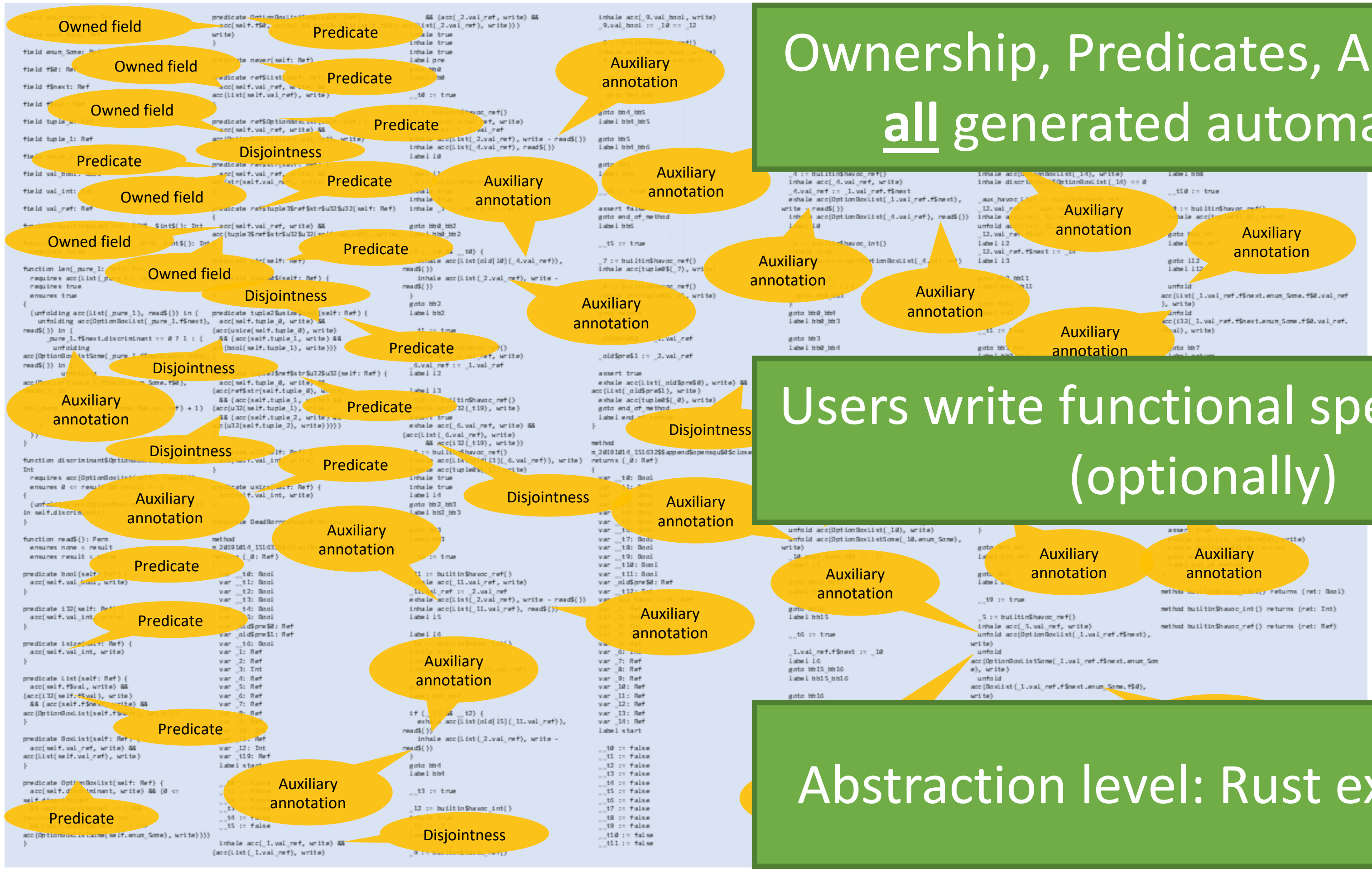
- Types
- Signature
- Compiler analyses (e.g. borrow checker)
- User specifications (optional)



Verification Ingredients

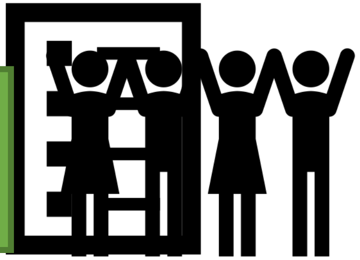
- Predicates and Ownership
- Pre/postconditions
- Auxiliary annotations
- Functional specification

Core Proofs: Behind the Scenes



Ownership, Predicates, Annotations **all** generated automatically

Users write functional specifications (optionally)

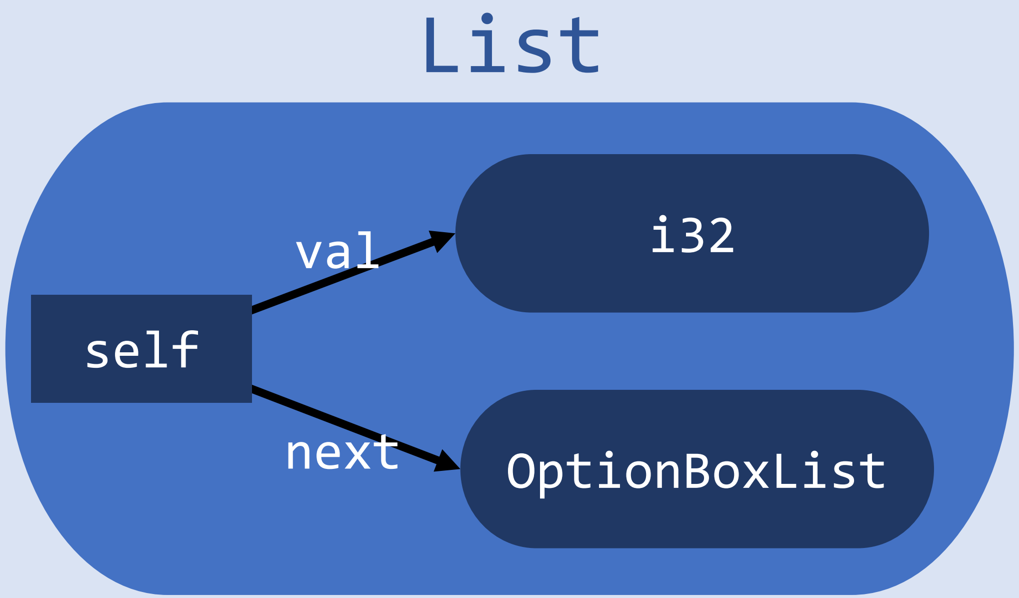


Abstraction level: Rust expressions

Type Encoding

```
struct List { val: i32, next: Option<Box<List>> }
```

Rust



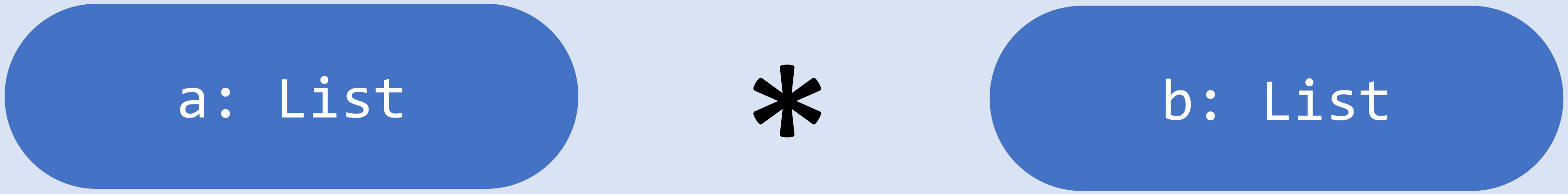

```
predicate List(self: Ref)
{
  acc(self.val) *
  acc(self.next) *
  i32(self.val) *
  OptionBoxList(self.next)
}
```

Viper

Signature Encoding

```
fn client(a: &mut List, b: &mut List)
```

Rust




```
method client(a: Ref, b: Ref)
  requires List(a) * List(b) && a.sorted() && …
  ensures List(a) * List(b) && a.sorted()
```

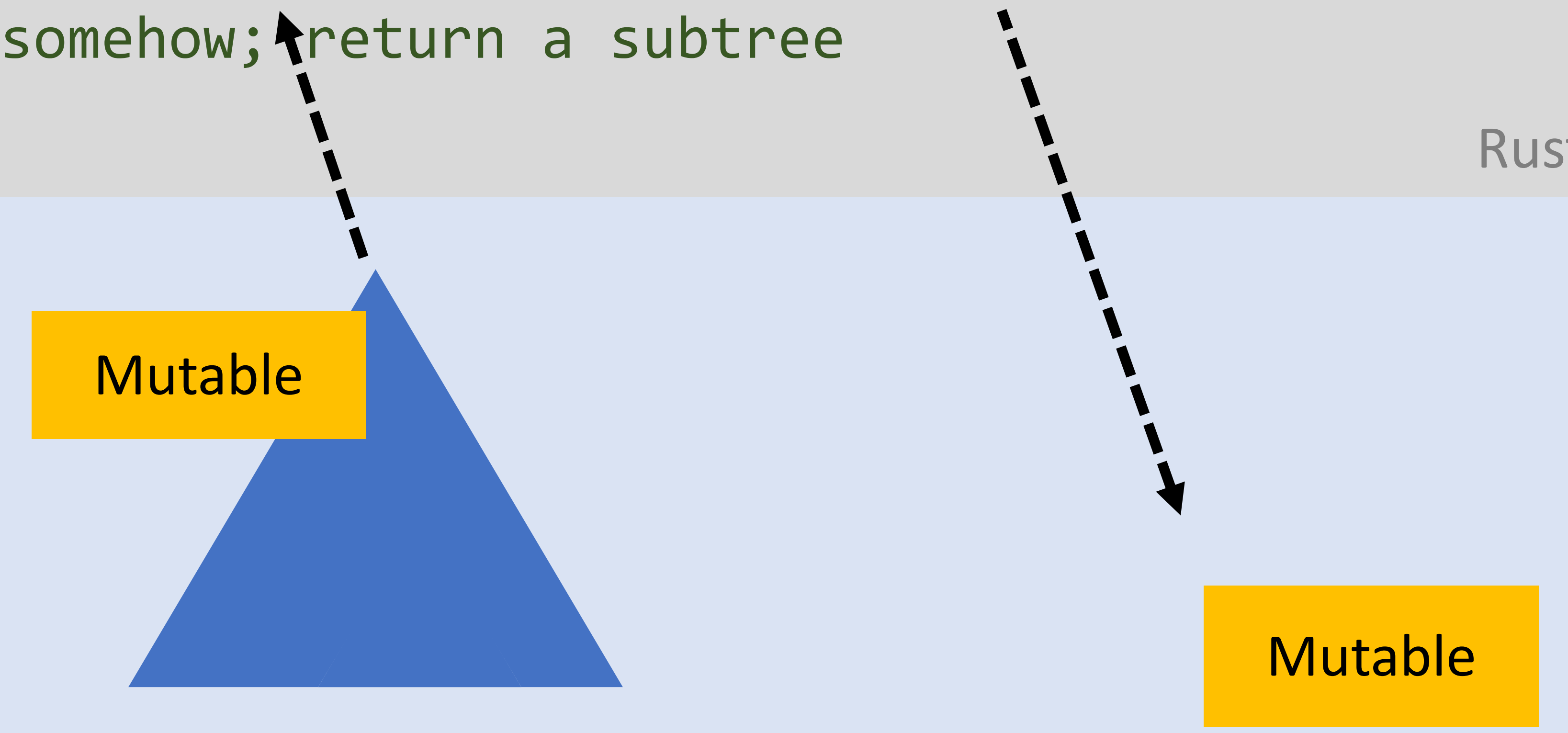
Viper

Reborrowing Challenges

```
fn get(t: &mut BinaryTree) -> &mut BinaryTree {
  // traverse somehow; return a subtree
}
```

Rust

For the caller:



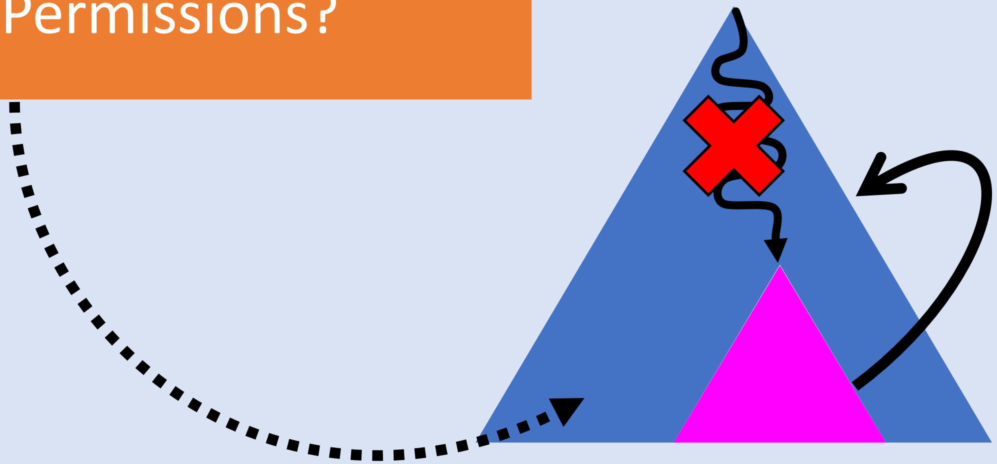
Reborrowing Challenges

```rust
fn get(t: &mut BinaryTree) -> &mut BinaryTree {
  // traverse somehow; return a subtree
}
```

Rust

Permissions?

Combined effect?



Reborrowing Challenges

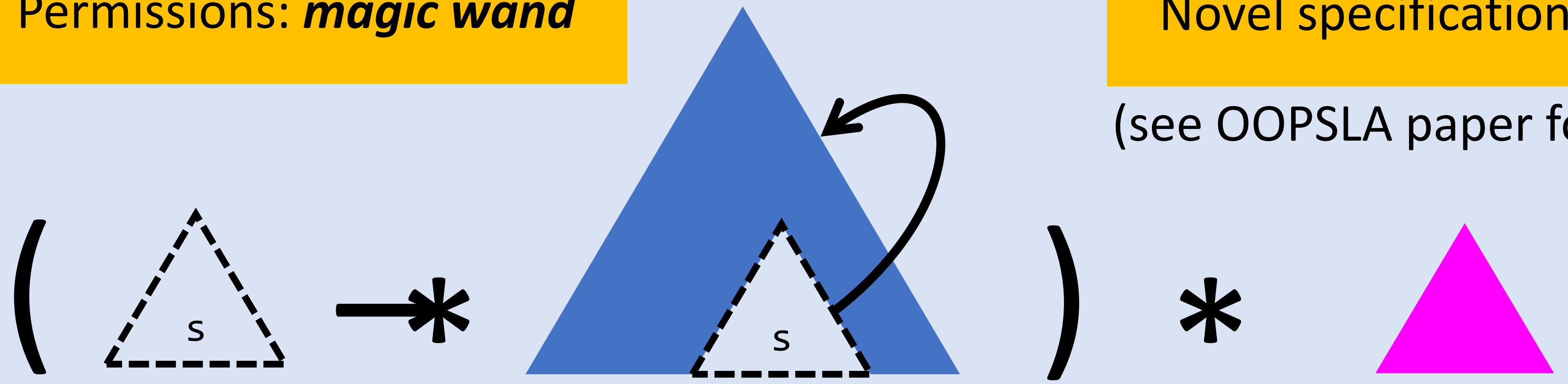
```rust
fn get(t: &mut BinaryTree) -> &mut BinaryTree {
  // traverse somehow; return a subtree
}
```

Rust

Permissions: ***magic wand***

Novel specification: ***pledges***

(see OOPSLA paper for details…)



Evaluation (no specifications)

500 most downloaded packages (crates)

No specification

11’791 (21%) supported functions

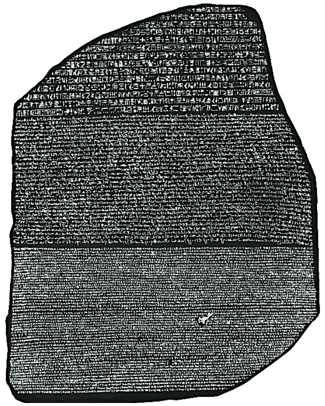
Total: **~40K** loc

100% of functions: core proof verifies

Total: **1M** lines of Viper

Auxiliary annotations: **100K**

Evaluation with specifications



rosettacode.org

+ Specification

Example	LOC	#Fns	Spec. LOC	Time (s) All	Time (s) Viper	No Panic	No Overflow	Verified Additional Properties
100 doors	19	2	7	10.9	7.4	✓	✓	
Binary Search (generic)	16	1	2	16.2	12.9	✓	✓	
Heapsort	39	3	18	30.6	26.2	✓	✓	
Knight's tour	89	6	71	127.6	120.2	✓	✓	
Knuth Shuffle	16	2	3	9.5	6.2	✓	✓	
Langton's Ant	58	4	22	16.7	11.8	✓	✓	
Selection Sort (generic)	20	2	8	19.2	15.2	✓	✓	
Ackermann Func.	16	2	17	7.4	4.4	-	×	Correct result
Binary Search (monomorphic)	16	1	29	25.5	21.4	✓	✓	Correct result
Fibonacci Seq.	46	6	26	9.1	5.7	-	-	Correct result
Knapsack Problem/0-1	27	1	86	139.4	131.6	✓	×	Correct computation
Linked List Stack	59	5	60	21.4	16.9	✓	-	Correct behaviour
Selection Sort (monomorphic)	20	2	34	29.6	24.2	✓	✓	Sorted result
Towers of Hanoi	10	2	5	5.9	3.2	-	✓	Correct param. range
Borrow First	7	1	1	6.6	3.6	✓	✓	
Message	13	1	0	7.2	4.2	×	-	

What else is in the paper?

VIPER ENCODING

AUTOMATION

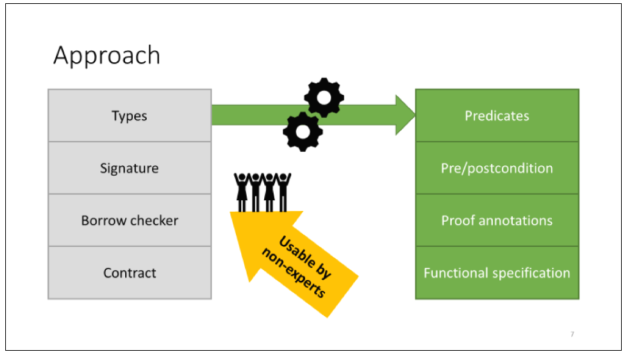
PLEDGES

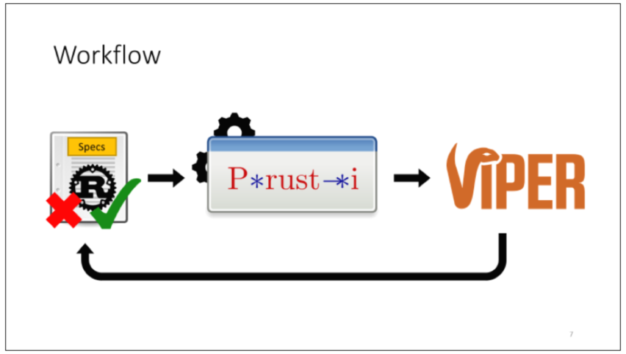
RUST SUBSET

Leveraging Rust Types for Modular Specification and Verification

To appear at OOPSLA 2019, Athens, Greece (next week)

Conclusion

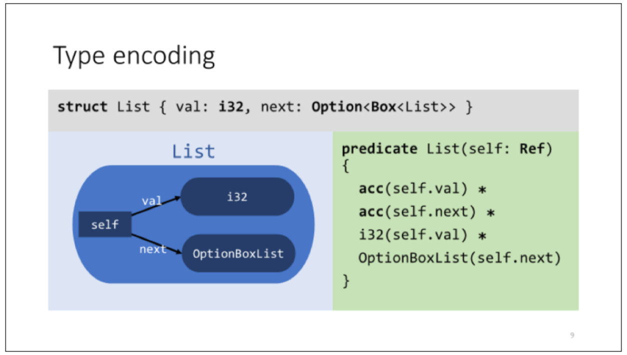


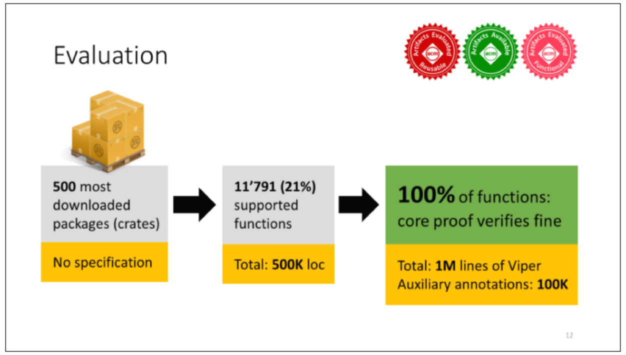


Plenty more to work on!
e.g. closures, unsafe code, reference counting, standard libraries, …

Dramatically simplifies Rust verification

Enables verification by developers





On the lookout for Master's (ETH/UBC) and PhD students (UBC)
- get in touch!

prusti.ethz.ch