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);
}
```

- Functional properties
- Memory Errors
- Aliasing
- Data Races
Verification Ingredients

Memory

list(a)

list(b)

acc(b.len)

What is this like to use?

Predicates

Ownership / Permissions

Auxiliary annotations

Disjointness of memory
Verification Ingredients at Scale

**Requires an expert**

**These steps are mandatory**

“Core proof”
Rust, and its type system

Can we exploit this type system for **verification**?

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

- 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

<table>
<thead>
<tr>
<th>Rust</th>
<th>Verification Ingredients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Predicates and Ownership</td>
</tr>
<tr>
<td>Signature</td>
<td>Pre/postconditions</td>
</tr>
<tr>
<td>Compiler analyses (e.g. borrow checker)</td>
<td>Auxiliary annotations</td>
</tr>
<tr>
<td>User specifications (optional)</td>
<td>Functional specification</td>
</tr>
</tbody>
</table>

**Usable by non-experts**
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

Rust

```rust
fn client(a: &mut List, b: &mut List) {
    // Implementation...
}
```

Viper

```viper
method client(a: Ref, b: Ref) {
    requires List(a) * List(b) && a.sorted() && ...
    ensures List(a) * List(b) && a.sorted()
}
```
Reborrowing Challenges

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

For the caller:
Reborrowing Challenges

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

Permissions? Combined effect?
Reborrowing Challenges

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

Permissions: *magic wand*

Novel specification: *pledges*

(see OOPSLA paper for details...)
Evaluation (no specifications)

- **500** most downloaded packages (crates)
- **11’791** (21%) supported functions
- **100%** of functions: core proof verifies
- **Total: ~40K loc**
- **100%** of functions: core proof verifies
- **Total: 1M lines of Viper**
- **Auxiliary annotations**: **100K**

No specification
Evaluation with specifications

<table>
<thead>
<tr>
<th>Example</th>
<th>LOC</th>
<th>#Fns</th>
<th>Spec. LOC</th>
<th>Time (s)</th>
<th>No Panic</th>
<th>No Overflow</th>
<th>Verified Additional Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 doors</td>
<td>19</td>
<td>2</td>
<td></td>
<td>7</td>
<td>10.9</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binary Search (generic)</td>
<td>16</td>
<td>1</td>
<td></td>
<td>2</td>
<td>16.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heapsort</td>
<td>39</td>
<td>3</td>
<td></td>
<td>18</td>
<td>30.6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Knight’s tour</td>
<td>89</td>
<td>6</td>
<td></td>
<td>71</td>
<td>127.6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Knuth Shuffle</td>
<td>16</td>
<td>2</td>
<td></td>
<td>3</td>
<td>9.5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Langton’s Ant</td>
<td>58</td>
<td>4</td>
<td></td>
<td>22</td>
<td>16.7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Selection Sort (generic)</td>
<td>20</td>
<td>2</td>
<td></td>
<td>8</td>
<td>19.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ackermann Func.</td>
<td>16</td>
<td>2</td>
<td>17</td>
<td>7.4</td>
<td>4.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Binary Search (monomorphic)</td>
<td>16</td>
<td>1</td>
<td>29</td>
<td>9.1</td>
<td>5.7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fibonacci Seq.</td>
<td>46</td>
<td>6</td>
<td>26</td>
<td>9.1</td>
<td>5.7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Knapsack Problem/0-1</td>
<td>27</td>
<td>1</td>
<td>86</td>
<td>139.4</td>
<td>131.6</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Linked List Stack</td>
<td>59</td>
<td>5</td>
<td>60</td>
<td>21.4</td>
<td>16.9</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Selection Sort (monomorphic)</td>
<td>20</td>
<td>2</td>
<td>34</td>
<td>29.6</td>
<td>24.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Towers of Hanoi</td>
<td>10</td>
<td>2</td>
<td>5</td>
<td>5.9</td>
<td>3.2</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Borrow First</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>6.6</td>
<td>3.6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Message</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>7.2</td>
<td>4.2</td>
<td>×</td>
<td>-</td>
</tr>
</tbody>
</table>

+ Specification

rosetacode.org
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

Dramatically simplifies Rust verification

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

Enables verification by developers

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

prusti.ethz.ch