Hacking the Rust object system at Mozilla

Lindsey Kuper
Grinnell College
April 5, 2012
Me and how I got here
Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
- Perl plumbing at a publishing company, 2006–2008
Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
- Perl plumbing at a publishing company, 2006–2008
  - but in 2007, I moved in with a couple of Haskell hackers...
Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
- Perl plumbing at a publishing company, 2006–2008
  - but in 2007, I moved in with a couple of Haskell hackers...
- Ph.D. student at Indiana studying PL since fall 2008
My field: programming language semantics

- Questions we might want to answer:

Source: Andrew Myers’ CS 611 course at Cornell
My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?

Source: Andrew Myers’ CS 611 course at Cornell
Questions we might want to answer:
- Is this program correct?
- Will this program encounter a run-time type error?

Source: Andrew Myers’ CS 611 course at Cornell
My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?

Source: Andrew Myers’ CS 611 course at Cornell
My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?
  - Is this optimization a safe program transformation?

Source: Andrew Myers’ CS 611 course at Cornell
My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?
  - Is this optimization a safe program transformation?
  - Can programs written in this language crash?

Source: Andrew Myers’ CS 611 course at Cornell
Questions we might want to answer:

- Is this program correct?
- Will this program encounter a run-time type error?
- Is one program indistinguishable from another?
- Is this optimization a safe program transformation?
- Can programs written in this language crash?
- Is this compiler translation correct?

Source: Andrew Myers’ CS 611 course at Cornell
My field: programming language semantics

Questions we might want to answer:

- Is this program correct?
- Will this program encounter a run-time type error?
- Is one program indistinguishable from another?
- Is this optimization a safe program transformation?
- Can programs written in this language crash?
- Is this compiler translation correct?
- Can source language A be translated into target language B?

Source: Andrew Myers’ CS 611 course at Cornell
Why Rust?

- What do we want in a programming language?

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code
  - Safe: type system provides guarantees that prevent certain bugs

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code
  - Safe: type system provides guarantees that prevent certain bugs
  - Concurrent: easy to build concurrent programs and to take advantage of parallelism

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code
  - Safe: type system provides guarantees that prevent certain bugs
  - Concurrent: easy to build concurrent programs and to take advantage of parallelism
  - “Systemsy”: fine-grained control, predictable performance characteristics

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we have now?

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe
  - Erlang is Safe and Concurrent

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe
  - Erlang is Safe and Concurrent
  - Haskell is (sometimes) Fast, (very) Safe, and Concurrent

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe
  - Erlang is Safe and Concurrent
  - Haskell is (sometimes) Fast, (very) Safe, and Concurrent
  - Java and C# are Fast and Safe

Source: Michael Sullivan, “Closures for Rust”
Why Rust?

A systems language pursuing the trifecta: fast, concurrent, safe
You worked on the *what* system?!
You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
- and was told, “None of that’s implemented yet; go for it!”
You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
  - and was told, “None of that’s implemented yet; go for it!”
- When I started: no object extension, method overriding, or self-dispatch
You worked on the **what** system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
  - and was told, “None of that’s implemented yet; go for it!”
- When I started: no object extension, method overriding, or self-dispatch
- During my internship, I implemented those things
You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
  - and was told, “None of that’s implemented yet; go for it!”
- When I started: no object extension, method overriding, or self-dispatch
- During my internship, I implemented those things
  - and learned that they interact with each other in interesting ways
Self-dispatch
Self-dispatch

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();
assert (shortcat.zzz() == "meow");
```
Self-dispatch + object extension

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");
```
Self-dispatch + object extension

```plaintext
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");

let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcut

    assert (longcat.zzz() == "meow");
```
A brainteaser...

```
obj cat() {
  fn ack() -> str {
    ret "ack";
  }
  fn meow() -> str {
    ret "meow";
  }
  fn zzz() -> str {
    ret self.meow();
  }
}

let longcat = obj() {
  fn lol() -> str {
    ret "lol";
  }
  fn nyan() -> str {
    ret "nyan";
  }
  with shortcut
    assert (longcat.zzz() == "meow");
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");
```

After my first implementation attempt, this assertion failed. Why?
A hint...

virtual table is virtual
A brainteaser...

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcut
    
    assert (longcat.zzz() == "meow");
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");
```

After my first implementation attempt, this assertion failed. Why?
A brainteaser...

```javascript
obj cat() {
  fn ack() -> str {
    ret "ack";
  }
  fn meow() -> str {
    ret "meow";
  }
  fn zzz() -> str {
    ret self.meow();
  }
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");
```

```javascript
let longcat = obj() {
  fn lol() -> str {
    ret "lol";
  }
  fn nyan() -> str {
    ret "nyan";
  }
  with shortcut
  ```
```
```
assert (longcat.zzz() == "meow");
```
```
```
```
```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>forward to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ack</td>
<td>shortcut.ack()</td>
</tr>
<tr>
<td>1</td>
<td>lol</td>
<td>&quot;lol&quot;</td>
</tr>
<tr>
<td>2</td>
<td>meow</td>
<td>shortcut.meow()</td>
</tr>
<tr>
<td>3</td>
<td>nyan</td>
<td>&quot;nyan&quot;</td>
</tr>
<tr>
<td>4</td>
<td>zzz</td>
<td>shortcut.zzz()</td>
</tr>
</tbody>
</table>
```
```
A brainteaser...

Let `longcat = obj()` {
  fn `lol()` -> str {
    ret "lol";
  }
  fn `nyan()` -> str {
    ret "nyan";
  }
  with shortcut

  assert (longcat.zzz() == "meow");
}

Let `shortcut = cat()`;

assert (shortcut.zzz() == "meow");

<table>
<thead>
<tr>
<th>longcat’s vtable</th>
<th>0</th>
<th>ack</th>
<th>forward to shortcut.ack()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lol</td>
<td>ret “lol”</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>meow</td>
<td>forward to shortcut.meow()</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>nyan</td>
<td>ret “nyan”</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>zzz</td>
<td>forward to shortcut.zzz()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>shortcut’s vtable</th>
<th>0</th>
<th>ack</th>
<th>ret “ack”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>meow</td>
<td>ret “meow”</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td>ret self.meow()</td>
<td></td>
</tr>
</tbody>
</table>
How to fix it

Obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();
assert (shortcat.zzz() == "meow");

let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat

    assert (longcat.zzz() == "meow");

    obj shortcat = cat();
    assert (shortcat.zzz() == "meow");

<table>
<thead>
<tr>
<th>0</th>
<th>ack</th>
<th>forward to shortcat.ack()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lol</td>
<td>ret &quot;lol&quot;</td>
</tr>
<tr>
<td>2</td>
<td>meow</td>
<td>forward to shortcat.meow()</td>
</tr>
<tr>
<td>3</td>
<td>nyan</td>
<td>ret &quot;nyan&quot;</td>
</tr>
<tr>
<td>4</td>
<td>zzz</td>
<td>forward to shortcat.zzz()</td>
</tr>
</tbody>
</table>
How to fix it

```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}
```

```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
    assert (longcat.zzz() == "meow");
}
```

---

<table>
<thead>
<tr>
<th>shortcat's vtable</th>
<th>longcat's vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  ack</td>
<td>forward to shortcat.ack()</td>
</tr>
<tr>
<td>1  lol</td>
<td>ret &quot;lol&quot;</td>
</tr>
<tr>
<td>2  meow</td>
<td>forward to shortcat.meow()</td>
</tr>
<tr>
<td>3  nyan</td>
<td>ret &quot;nyan&quot;</td>
</tr>
<tr>
<td>4  zzz</td>
<td>forward to shortcat.zzz()</td>
</tr>
</tbody>
</table>

---

shortcat's backwarding vtable

| 0  ack | backward to longcat.ack() |
| 1  meow | backward to longcat.meow() |
| 2  zzz | backward to longcat.zzz() |
How to fix it

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}
```

```javascript
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
}

assert (longcat.zzz() == "meow");
```

**shortcat's backwarding vtable**

<table>
<thead>
<tr>
<th></th>
<th>Item</th>
<th>Backward To</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ack</td>
<td>longcat.ack()</td>
</tr>
<tr>
<td>1</td>
<td>meow</td>
<td>longcat.meow()</td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td>longcat.zzz()</td>
</tr>
</tbody>
</table>

**longcat's vtable**

<table>
<thead>
<tr>
<th></th>
<th>Item</th>
<th>Forward To</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ack</td>
<td>shortcat.ack()</td>
</tr>
<tr>
<td>1</td>
<td>lol</td>
<td>&quot;lol&quot;</td>
</tr>
<tr>
<td>2</td>
<td>meow</td>
<td>shortcat.meow()</td>
</tr>
<tr>
<td>3</td>
<td>nyan</td>
<td>&quot;nyan&quot;</td>
</tr>
<tr>
<td>4</td>
<td>zzz</td>
<td>shortcat.zzz()</td>
</tr>
</tbody>
</table>
Self-dispatch + object extension + overriding

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();
assert (shortcat.zzz() == "meow");
```
Self-dispatch + object extension + overriding

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();
assert (shortcat.zzz() == "meow");
```
Self-dispatch + object extension + overriding

```plaintext
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcut
    assert (longercat.zzz() == "zzz");
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");
```
Self-dispatch + object extension + overriding

Caveat: Some disagreement on whether it should work this way (see: Aldrich, “Selective Open Recursion”)

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcut
    
    assert (longercat.zzz() == "zzz");
}

let shortcut = cat();

assert (shortcut.zzz() == "meow");
```
Self-dispatch + object extension + overriding

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcut
        assert (longercat.zzz() == "zzz");
}

let shortcut = cat();
assert (shortcut.zzz() == "meow");
```
Self-dispatch + object extension + overriding

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack"
    }
    fn meow() -> str {
        ret "meow"
    }
    fn zzz() -> str {
        ret self.meow()
    }
}

let longercat = obj() {
    fn meow() -> str {
        ret "zzz"
    }
    with shortcutcat
    assert (longercat.zzz() == "zzz")
}

let shortcutcat = cat();
assert (shortcutcat.zzz() == "meow")
```

**longercat's vtable**

<table>
<thead>
<tr>
<th>0</th>
<th>ack</th>
<th>forward to shortcut.ack()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>meow</td>
<td>ret &quot;zzz&quot;</td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td>forward to shortcut.zzz()</td>
</tr>
</tbody>
</table>
Self-dispatch + object extension + overriding

```rust
let longercat = obj() {  
    fn meow() -> str {  
        ret "meow";
    }  
    with shortcut
    assert (longercat.zzz() == "zzz");
}

let shortcat = cat();
assert (shortcat.zzz() == "meow");

obj cat() {  
    fn ack() -> str {  
        ret "ack";
    }  
    fn meow() -> str {  
        ret "meow";
    }  
    fn zzz() -> str {  
        ret self.meow();
    }
}
```

<table>
<thead>
<tr>
<th>shortcat's vtable</th>
<th>longercat's vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ack</td>
</tr>
<tr>
<td>1</td>
<td>meow</td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
</tr>
</tbody>
</table>
Self-dispatch + object extension + overriding

```rust
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}
```

```rust
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcutcat
}
assert (longercat.zzz() == "zzz");
```

**shortcat's vtable**

<table>
<thead>
<tr>
<th>0</th>
<th>ack</th>
<th>backward to longercat.ack()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>meow</td>
<td>backward to longercat.meow()</td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td>backward to longercat.zzz()</td>
</tr>
</tbody>
</table>

**shortcat's vtable**

<table>
<thead>
<tr>
<th>0</th>
<th>ack</th>
<th>ret &quot;ack&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>meow</td>
<td>ret &quot;meow&quot;</td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td>ret self.meow()</td>
</tr>
</tbody>
</table>

**longercat's vtable**

<table>
<thead>
<tr>
<th>0</th>
<th>ack</th>
<th>forward to shortcut.ack()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>meow</td>
<td>ret &quot;zzz&quot;</td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td>forward to shortcut.zzz()</td>
</tr>
</tbody>
</table>
Self-dispatch + object extension + overriding

```rust
fn ack() -> str {
    ret "ack";
}
fn meow() -> str {
    ret "meow";
}
fn zzz() -> str {
    ret self.meow();
}
```

```rust
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcutcat
    assert (longercat.zzz() == "zzz");
}
```

**shortcutcat’s vtable**

<table>
<thead>
<tr>
<th></th>
<th>ack</th>
<th>meow</th>
<th>zzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>meow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>zzz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**longercat’s vtable**

<table>
<thead>
<tr>
<th></th>
<th>ack</th>
<th>meow</th>
<th>zzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**shortcutcat’s backwarding vtable**

<table>
<thead>
<tr>
<th></th>
<th>ack</th>
<th>meow</th>
<th>zzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Self-dispatch + object extension + overriding to arbitrary depth

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortercat = cat();
assert (shortercat.fff() == "zzz");

let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortercat
    {
        assert (longercat.zzz() == "zzz");
    }
}
```

15
Self-dispatch + object extension + overriding to arbitrary depth

```javascript
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();
assert (shortcat.ddd() == "ddd");

let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
    assert (longercat.zzz() == "zzz");
```
Self-dispatch + object extension + overriding to arbitrary depth

```plaintext
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcut
        assert (longercat.zzz() == "zzz");
};

let shortcutcat = cat();
assert (shortcutcat.zzz() == "zzz");

let evenlongercat = obj() {
    fn meow() -> str {
        ret "zLLLLL";
    }
    with longercat
        assert (evenlongercat.zzz() == "zzzzzzz");
};
```
Self-dispatch + object extension + overriding to arbitrary depth
We need a way to temporarily pretend that self is the inner object, while still keeping track of what the extended self is.
We need a way to temporarily pretend that self is the inner object, while still keeping track of what the extended self is.

Solution: create a stack of “fake selves” threaded through the run-time stack.
Self-dispatch + object extension + overriding to arbitrary depth

- We need a way to temporarily pretend that self is the inner object, while still keeping track of what the extended self is.

- Solution: create a stack of “fake selves” threaded through the run-time stack.

- Every **forwarding** function allocates space in its frame for a “fake self” comprising a backwarding vtable and an inner object body.
Go try it out!

rust-lang.org
Questions?

Thanks to:
Graydon Hoare and everyone on the Rust team
Dave Herman and everyone at Mozilla Research

Photo by jamesrbowe on Flickr. Thanks!