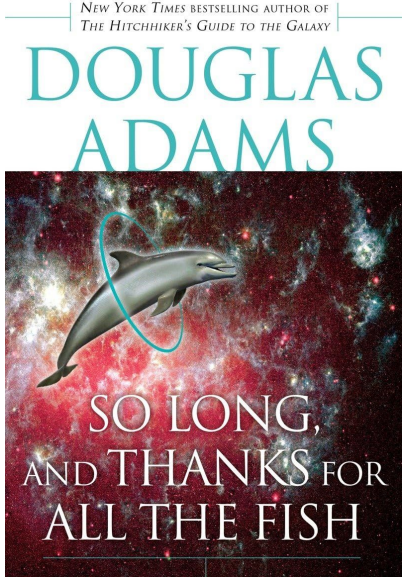
Hello, and thanks for all the feature requests

Tyler Mandry

Hello, and thanks for all the feature requests

Tyler Mandry

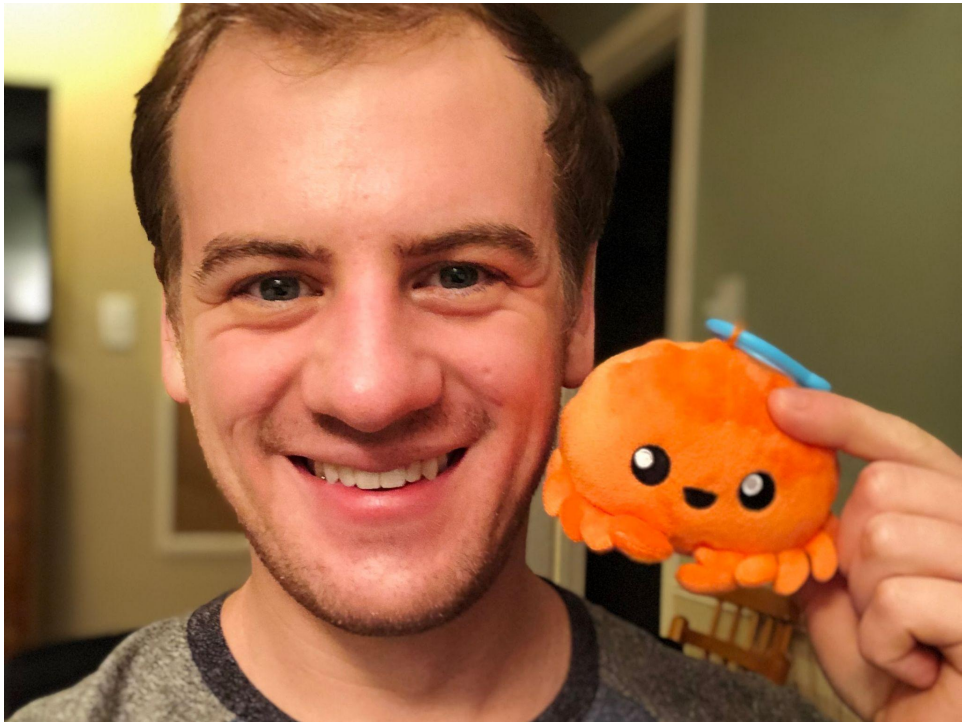


Hello, I’m Tyler

- Co-lead Rust language design team
- Full-time @ Google

Longtime OS dev

Longtime OS dev



How can we make designing and shipping Rust language features enjoyable?

Alignment in attention

- Regular meetings
- Flagship goals
- Longer commitment cycles
- Champions

Alignment in design values

- Proactive champions
- Regular design meetings on active goals
- ⅔ to resolve concerns
- Design or "motivation-only" RFCs
- Design leads

Limited bandwidth and distractibility

- Prioritize alignment between goals
- “North star” strategy RFCs

Language features

Custom Reference Types

Projection

```
foo: &Cell<Struct>
foo.@field: &Cell<Field>
```

Reborrowing

```
pin.as_mut().method()
=> pin.method()
```

Autoref

```
pin.as_ref().method()
=> pin.method()
```

In-place initialization

```
fn make_x_in_place() -> Result<init MyStruct, Error> {
    let out: MyStruct;
    out.another_field = 20;
    if foo() {
        // another_field is implicitly dropped
        return Err(err);
    }
    out.field = [0; 1024];
    Ok(out)
}
```

Externally-sized types

```
// Only usable behind a pointer.

extern type Extern;
```

Partial initialization

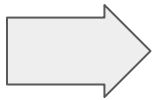
Ergonomic ref counting

```
let foo: Arc<Struct>;

// ...

let foo_clone = Arc::clone(&foo);
let closure = move || {
    foo_clone.clone().bar();
};

closure();
closure();
```



```
let foo: Arc<Struct>;

// ...


let closure = move || {
    foo.clone().bar();
};

closure();
closure();
```

Marker effects

```
const trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }
}

const fn foo<T: ~const PartialEq>(x: &T, y: &T) -> bool {
    x.eq(y)
}
```

Existential lifetimes

```
let value = GhostToken::new(|mut token| {
    let cell = GhostCell::new(State::new());

    cell.borrow_mut(&mut token).modify();
    cell.borrow(&token).get()
});

impl GhostToken {
  fn new<R>(f: impl FnOnce(GhostToken<'_>) -> R) -> R { .. }
}
```

Existential lifetimes

```
fn token<exists 'a>() -> GhostToken<'a> { ... }

let token = token();
let cell = GhostCell::new(State::new());

cell.borrow_mut(&mut token).modify();
let value = cell.borrow(&token).get();
```