Rust

I can't believe it's not functional

Who am I?

- Sean Griffin
- Developer at Shopify
- Rails Committer
- Creator of Diesel
- Cohost of The Bike Shed

What is Rust?

Rust is not a functional programming language

(but it plays one on TV)

- Strong type system
- Type inference
- Pattern matching
- Higher order functions
- Immutable by default
- Iterators
- Monadic error handling

3 string types

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

- runs blazingly fast
- prevents segfaults
- guarantees thread safety

- ~~runs blazingly fast~~
- prevents segfaults
- guarantees thread safety

- ~~runs blazingly fast~~
- ~~prevents segfaults~~
- guarantees thread safety

Rust is a pragmatic language, providing a strong static type system in a way that is easy to use in the "real world"

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

```haskell
instance (Eq a) => Eq [a] where
    xs == ys =
        length x == length y &&
        all (uncurry (==)) (zip xs ys)
```

```
trait Eq {
    fn eq(&self, other: &Self) -> bool;

    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
```

+, +=, &, &=, |, |=, ^, ^=, /, /=, *, *=,
-, !, %, %=, <<, <<=, >>, >>=, -, -=
function calls, subscript access,
dereferencing, dereferencing assignment

I don't know what ~>==<~ does and I can't google it

```
trait Eq {
    fn eq(&self, other: &Self) -> bool;

    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
```

```
impl<A: Eq> Eq for [A] {
    fn eq(&self, other: &Self) -> bool {
        self.len() == other.len() &&
            self.iter().zip(other).all(|(a, b)| a == b)
    }
}
```

```haskell
instance (Eq a) => Eq (Maybe a) where
    Just a == Just b = a == b
    Nothing == Nothing = true
    _ == _ = false
```

```
impl<A: Eq> Eq for Option<A> {
    fn eq(&self, other: &Self) -> bool {
        match (self, other) {
            (Some(x), Some(y)) => x == y,
            (None, None) => true
            _ => false
        }
    }
}
```

```haskell
data Maybe a = Just a | Nothing
    deriving (Eq, Ord, Show)
```

```rust
#[derive(PartialEq, Eq, PartialOrd, Ord, Debug)]
enum Option<A> {
    Some(A),
    None,
}
```

Caveat: No Higher Kinded Types

(So no `Functor`, `Monad`, etc)

Ownership

What does it mean to own an value?

What does it mean to destroy an value?

What's an example of a value that can no longer be used at some point?

When will you stop answering questions with more questions?

Example time!

```rust
fn do_stuff_with_file(f: File) {
    // ...
    // f will be destroyed when this function returns
}

fn main() {
    let file = File::open("hello.txt").unwrap();
    do_stuff_with_file(file);
    // ownership of file is moved into `do_stuff_with_file`

    do_more_stuff_with_file(file);
    // ERROR: Use of moved value `file`
}
```

```rust
fn do_stuff_with_file(f: &File) {
    //                        ^~~~ Note the ampersand.
    // ...
    // f is borrowed, and therefore not destroyed
}

fn main() {
    let file = File::open("hello.txt").unwrap();
    do_stuff_with_file(&file);
    // do_stuff_with_file borrows `file`, ownership is not moved.

    do_more_stuff_with_file(file);
    // This works fine now.
}
```

Rust uses these in novel ways

Thread Safety

```
pub fn spawn<F>(f: F) -> JoinHandle
    where F: FnOnce() + Send + 'static
```

'static

- Value must outlive the scope it was created in
- For closures, this means they have to own all data that they access
- No shared references

Send

- A value that can safely be sent from one thread to another
- Any data which is made only of `Send` values is automatically `Send`
- Types which can be used for sharing data are not `Send` (like pointers).

Sync

- A value that can safely be shared between threads.
- Not explicitly referenced by the Thread signature.

```
unsafe impl<T> Send for &T where T: Sync
```

What makes a `Fn()` be `Send`?

If you were to make a closure be a struct, it'd look something like this:

```
struct Closure {
  args: Args,
  env: Environment,
}

struct Args {
  # field for each argument type
}

struct Environment {
  # field for each value closed over
}
```

How do you share data in Rust anyway?

- Cannot share data by default in Rust, only borrow it.
- Borrows work well for straightforward lifetimes, does not work for complex
  ones

How do you share data in Rust anyway?

Enter `Rc` (short for "reference counted")

```
pub struct Rc<T: ?Sized> {
    _ptr: NonZero<*mut RcBox<T>>,
}
```

How do you share data in Rust anyway?

- `*mut` is a pointer
- Pointers are not `Send`
- `Rc` is not `Send`

Ok no seriously how do you share data across threads, though?

Enter `Arc` (short for atomic reference counted)

- Similar structure to `Rc`
- Implementation is atomic
- Still uses a pointer, so must be explicitly declared as `Send`

```
unsafe impl<T> for Arc<T> where T: Send + Sync
```

How does that prevent data races?

- `Arc` can only give you a shared (immutable) reference to it's data.
- The `Arc` owns the value, no way to reclaim ownership.
- Need a type that is `Sync` and also exposes a mutable reference to it's data

Enter `Mutex`

- This and other locks are the only types that give mutability and are `Sync`
- The only way to share data across threads (in safe Rust anyway)
- The primitives `Mutex` uses are still accessible but your type would not be
  `Sync` or `Send` by default

Rust's type system prevents writing incorrect programs

Performance

- No garbage collector
- Minimal runtime
- Stack allocated by default
- Zero cost abstractions

```
let versions = Version::belonging_to(krate)
    .select(id)
    .order(num.desc())
    .limit(5);
let downloads = try!(version_downloads
    .filter(date.gt(now - 90.days()))
    .filter(version_id.eq(any(versions)))
    .order(date)
    .load::<Download>(&conn));
```

```
SELECT version_downloads.*
  WHERE date > (NOW() - '90 days')
    AND version_id = ANY(
      SELECT id FROM versions
        WHERE crate_id = 1
        ORDER BY num DESC
        LIMIT 5
    )
  ORDER BY date
```

```
let versions = Version::belonging_to(krate)
    .select(id)
    .order(num.desc())
    .limit(5);
let downloads = try!(version_downloads
    .filter(date.gt(now - 90.days()))
    .filter(version_id.eq(any(versions)))
    .order(date)
    .load::<Download>(&conn));
```

Rust isn't functional, but you'll still feel right at home

Questions?

Thank you!

- Sean Griffin
- twitter: @sgrif
- github: sgrif
- email: sean@seantheprogrammer.com
- podcast: http://bikeshed.fm
- Diesel: http://diesel.rs