# Some tidbits about functional interfaces

Matty Weatherley

# "Functional interface"

What do I mean by "functional interface"?

- *Functional* in the sense of functional programming — i.e., composing and applying functions explicitly. (In Rust world, this generally means things which take closures as inputs.)

# “Functional interface”

What do I mean by “functional interface”?

- *Functional* in the sense of functional programming — i.e., composing and applying functions explicitly. (In Rust world, this generally means things which take closures as inputs.)
- *Interface* here being a **trait**-interface which abstracts over data implementations to provide some shared functionality through methods.

# “Functional interface”

What do I mean by “functional interface”?

- *Functional* in the sense of functional programming — i.e., composing and applying functions explicitly. (In Rust world, this generally means things which take closures as inputs.)
- *Interface* here being a **trait**-interface which abstracts over data implementations to provide some shared functionality through methods.

The key example is `Iterator::map`.

**Iterator::map**

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    Self: Sized,
    F: FnMut(Self::Item) -> B,
{ //... }
```

# Outline

1. Prototypical implementation
2. Specializations
3. Ownership and borrowing
4. Object safety and type erasure

# Toy example: Curves

Let's use something simpler than `Iterator` for the sake of illustration.
Here is a trait that describes one-parameter families of values of type `T`:

```
pub trait Curve<S> {
    fn sample(&self, t: f64) -> S;
}
```

# Toy example: Curves

Let's use something simpler than `Iterator` for the sake of illustration. Here is a trait that describes one-parameter families of values of type `T`:

```
pub trait Curve<S> {
    fn sample(&self, t: f64) -> S;
}
```

Implementors might include curves interpolated from a set of samples, curves defined by functions or equations, and so on.

# Toy example: Curves

Let's use something simpler than `Iterator` for the sake of illustration. Here is a trait that describes one-parameter families of values of type `T`:

```
pub trait Curve<S> {
    fn sample(&self, t: f64) -> S;
}
```

Implementors might include curves interpolated from a set of samples, curves defined by functions or equations, and so on.

The only required method is `sample`, but we might also expose something that transforms `Curve<S>` into `Curve<T>` given a function `S -> T`:

**Curve::map**

```
fn map<T>(self, f: impl Fn(S) -> T) -> impl Curve<T> {
    //...
}
```

# Curve::map

## Concrete implementation

```rust
struct MapCurve<S, T, C, F>
where
    C: Curve<S>,
    F: Fn(S) -> T,
{
    inner: C,
    function: F,
    _phantom: PhantomData<(S, T)>,
}

impl<S, T, C, F> Curve<T> for MapCurve<S, T, C, F>
where //... same as above
{
    fn sample(&self, t: f64) -> T {
        (self.function)(self.inner.sample(t))
    }
}
```

# Curve::map

MapCurve provides a basis for our functional interface method:

## Curve::map

```
fn map<T>(self, f: impl Fn(S) -> T) -> impl Curve<T>
where Self: Sized {
    MapCurve {
        inner: self,
        function: f,
        _phantom: PhantomData,
    }
}
```

# Curve::map

`MapCurve` provides a basis for our functional interface method:

**Curve::map**

```
fn map<T>(self, f: impl Fn(S) -> T) -> impl Curve<T>
where Self: Sized {
    MapCurve {
        inner: self,
        function: f,
        _phantom: PhantomData,
    }
}
```

A few things to notice about this construction:

- It's lazy — i.e. `f` is never invoked until `sample` is called on its output.

# Curve::map

`MapCurve` provides a basis for our functional interface method:

**Curve::map**

```
fn map<T>(self, f: impl Fn(S) -> T) -> impl Curve<T>
where Self: Sized {
    MapCurve {
        inner: self,
        function: f,
        _phantom: PhantomData,
    }
}
```

A few things to notice about this construction:

- It's lazy — i.e. `f` is never invoked until `sample` is called on its output.
- It moves `self` into its output.

# Curve::map

`MapCurve` provides a basis for our functional interface method:

**Curve::map**

```
fn map<T>(self, f: impl Fn(S) -> T) -> impl Curve<T>
where Self: Sized {
    MapCurve {
        inner: self,
        function: f,
        _phantom: PhantomData,
    }
}
```

A few things to notice about this construction:

- It's lazy — i.e. `f` is never invoked until `sample` is called on its output.
- It moves `self` into its output.
- It requires `Self: Sized`.

# Laziness

Some problems with laziness:

**Loss of concreteness**

It might be hard, for example, to serialize/deserialize a `MapCurve` even if the input curve can easily be serialized/deserialized, since it holds an arbitrary `Fn` closure (and we only have an `impl Curve<T>` too).

# Laziness

Some problems with laziness:

**Loss of concreteness**

It might be hard, for example, to serialize/deserialize a `MapCurve` even if the input curve can easily be serialized/deserialized, since it holds an arbitrary `Fn` closure (and we only have an `impl Curve<T>` too).

**Unnecessary complexity**

Storing the mapping function may be more complicated or memory-intensive than just transforming some underlying owned data.

# Laziness

Some problems with laziness:

**Loss of concreteness**

It might be hard, for example, to serialize/deserialize a `MapCurve` even if the input curve can easily be serialized/deserialized, since it holds an arbitrary `Fn` closure (and we only have an `impl Curve<T>` too).

**Unnecessary complexity**

Storing the mapping function may be more complicated or memory-intensive than just transforming some underlying owned data.

**Struct nesting**

Since we're embedding the entire input into the `MapCurve` output, repeated calls will continue to increase the depth of struct nesting.

# Some specialized implementations

**Example:** `SampleCurve`

Underlying data implementation of `map` need not actually be lazy:

```
struct SampleCurve<S> {
    samples: Vec<S>,
}

impl<S> Curve<S> for SampleCurve<S> {
    fn sample(&self, t: f64) -> S {
        //... interpolated from self.samples
    }

    fn map<T>(self, f: impl Fn(S) -> T) -> impl Curve<T> {
        let mapped_samples: Vec<T> = self.samples.into_iter()
            .map(f)
            .collect();
        SampleCurve { samples: mapped_samples }
    }
}
```

# Some specialized implementations

One can imagine making a subtrait for types where some functional operations preserve the relevant form of concreteness, as in the preceding example; e.g.:

```
pub trait ConcreteCurve<S>: Curve<S> + Serialize + //...
{
    fn map_concrete<T>(self, f: Fn(S) -> T)
        -> impl ConcreteCurve<T>;
}
```

# Some specialized implementations

**Example: `MapCurve` itself**

Repeated calls to `Curve::map` can reuse the same inner data:

```
impl<S, T, C, F> Curve<T> for MapCurve<S, T, C, F> {
    //... sample implementation...
    fn map<U>(self, f: Fn(T) -> U) -> impl Curve<U> {
        let new_func = move |x| f((self.function)(x));
        MapCurve {
            inner: self.inner,
            function: new_func,
            _phantom: PhantomData,
        }
    }
}
```

# Another idea: GATs

Another approach is to specify the output type in the trait:

**Example: concrete map output type**

```
pub trait Curve<S> {
    //... sample method etc.
    type MapOutput<T, F: Fn(S) -> T>: Curve<T>;

    fn map<T, F: impl Fn(S) -> T>(self, f: F)
        -> Self::MapOutput<T, F>
    where Self: Sized;
}
```

# Another idea: GATs

Another approach is to specify the output type in the trait:

**Example: concrete map output type**

```
pub trait Curve<S> {
    //... sample method etc.
    type MapOutput<T, F: Fn(S) -> T>: Curve<T>;

    fn map<T, F: impl Fn(S) -> T>(self, f: F)
        -> Self::MapOutput<T, F>
    where Self: Sized;
}
```

Caveats:

- `Curve<S>` is no longer object-safe.

# Another idea: GATs

Another approach is to specify the output type in the trait:

**Example: concrete map output type**

```
pub trait Curve<S> {
    //... sample method etc.
    type MapOutput<T, F: Fn(S) -> T>: Curve<T>;

    fn map<T, F: impl Fn(S) -> T>(self, f: F)
        -> Self::MapOutput<T, F>
    where Self: Sized;
}
```

Caveats:

- `Curve<S>` is no longer object-safe.
- Associated type defaults are unstable.

# Another idea: GATs

Another approach is to specify the output type in the trait:

**Example: concrete map output type**

```rust
pub trait Curve<S> {
    //... sample method etc.
    type MapOutput<T, F: Fn(S) -> T>: Curve<T>;

    fn map<T, F: impl Fn(S) -> T>(self, f: F)
        -> Self::MapOutput<T, F>
    where Self: Sized;
}
```

Caveats:

- `Curve<S>` is no longer object-safe.
- Associated type defaults are unstable.
- Implementation of `map` cannot be defaulted.

# Ownership and borrowing

Onto the next problem: `Curve::map` takes ownership of `self`. This can be bad for ergonomics; for example, we might imagine that there is some other `Curve<S>` method which only needs to immutably borrow `self`:

```
fn extract_data(&self, impl IntoIterator<Item = f64>)
    -> impl Iterator<Item = S> { //... }
```

# Ownership and borrowing

Onto the next problem: `Curve::map` takes ownership of `self`. This can be bad for ergonomics; for example, we might imagine that there is some other `Curve<S>` method which only needs to immutably borrow `self`:

```
fn extract_data(&self, impl IntoIterator<Item = f64>)
    -> impl Iterator<Item = S> { //... }
```

Then we run into issues with patterns like this:

```
// my_curve is something which implements Curve<f64>
let my_curve = function_curve(|t| t * t + 2.0);
let some_data = my_curve
    .map(|x| x * x)
    .extract_data(some_iterator);
// Here, we can no longer use my_curve, even though
// we did no mutation
```

# Ownership and borrowing

You can try to circumvent this by throwing lifetimes and borrowing into `MapCurve` itself, but it's arguably better to go the opposite way:

# Ownership and borrowing

You can try to circumvent this by throwing lifetimes and borrowing into `MapCurve` itself, but it's arguably better to go the opposite way:

```
impl<S, C> Curve<S> for &C
where C: Curve<S> {
    fn sample(&self, t: f64) -> S {
        <C as Curve<S>>::sample(self, t)
    }
}
```

# Ownership and borrowing

You can try to circumvent this by throwing lifetimes and borrowing into `MapCurve` itself, but it's arguably better to go the opposite way:

```
impl<S, C> Curve<S> for &C
where C: Curve<S> {
    fn sample(&self, t: f64) -> S {
        <C as Curve<S>>::sample(self, t)
    }
}
```

With this in hand, the problem is easily avoided:

```
let my_curve = function_curve(|t| t * t + 2.0);
let some_data = (&my_curve)
    .map(|x| x * x)
    .extract_data(some_iterator);
// Now we can still use my_curve after the borrow
```

# Ownership and borrowing

In fact, `Iterator` does something exactly like this:

**Example: Iterator**

```
impl<I: Iterator + ?Sized> Iterator for &mut I {
    type Item = I::Item;
    //... other methods
}
```

# Ownership and borrowing

In fact, `Iterator` does something exactly like this:

**Example: Iterator**

```
impl<I: Iterator + ?Sized> Iterator for &mut I {
    type Item = I::Item;
    //... other methods
}
```

It also provides the convenience method `by_ref`, which makes syntax for this situation more convenient:

**Example: Iterator::by_ref**

```
fn by_ref(&mut self) -> &mut Self
where Self: Sized {
    self
}
```

This lets you write things like `iter.by_ref()` instead of `(&mut iter)`.

# Object safety and type erasure

Finally, `Curve::map` requires a `Sized` bound on `Self` since it must generally embed the `Curve` implementor into a field of `MapCurve`.

# Object safety and type erasure

Finally, `Curve::map` requires a `Sized` bound on `Self` since it must generally embed the `Curve` implementor into a field of `MapCurve`.

The primary consequence is:

**Exclusion from vtables**

A trait method requiring `Self` to be `Sized` is considered *explicitly non-dispatchable*, so it is excluded from the trait object completely.

**Unavailability of method on unsized types**

(Obviously.) This most prominently applies to `dyn Curve<S>` (see above). Note that this means that code like the following simply doesn't work, since the `Box` gets dereferenced:

```
let boxed_curve: Box<dyn Curve<f64>> = Box::new(my_curve);
// The following will fail to compile:
let mapped_curve = boxed_curve.map(|x| x * 2.0);
```

# Object safety and type erasure

Solutions for this kind of situation give implementations of `Curve<S>` for things like `Box<dyn Curve<S>>` explicitly:

```
impl<S, C, D> Curve<S> for D
where
    C: Curve<S> + ?Sized,
    D: Deref<Target = C>,
{
    fn sample(&self, t: f64) -> S {
        <C as Curve<S>>::sample(self, t)
    }
}
```

The practical implication is that calling `Curve::map` on a `Box<dyn Curve<S>>` uses the generic implementation of `map` based on the implementation of `sample` which is dispatched dynamically.

# Object safety and type erasure

**&T**

Note that `&T` implements `Deref` with a target of `T`, so this is actually subsumes the previously discussed blanket implementation.

# Object safety and type erasure

### &T

Note that `&T` implements `Deref` with a target of `T`, so this is actually subsumes the previously discussed blanket implementation.

### Caveat emptor

There is something sneakily problematic here: a specialized implementation of `map` can be passed through to a `&T`, but in the `Deref` version that is impossible because of the required `?Sized` bound which must therefore exclude `map`.

# Object safety and type erasure

**&T**

Note that `&T` implements `Deref` with a target of `T`, so this is actually subsumes the previously discussed blanket implementation.

**Caveat emptor**

There is something sneakily problematic here: a specialized implementation of `map` can be passed through to a `&T`, but in the `Deref` version that is impossible because of the required `?Sized` bound which must therefore exclude `map`.

(Always be suspicious of trait-based blanket implementations.)

# That's all!

Thanks for listening! This talk should be available on my GitHub profile if you want to see any of the slides again:

`https://github.com/mweatherley`