A brief overview of reflection in Rust

Matty Weatherley

Reflection

What is reflection?

- Reflection refers to the ability for a program and/or its components to introspect on their own structure.

Reflection

What is reflection?

- Reflection refers to the ability for a program and/or its components to introspect on their own structure.
- Often, what we want to do with reflection is to breaking down the compile-time/runtime distinction — for example, types aren't "real" at runtime, but we might still want to know about them.

Reflection

What is reflection?

- Reflection refers to the ability for a program and/or its components to introspect on their own structure.
- Often, what we want to do with reflection is to breaking down the compile-time/runtime distinction — for example, types aren't "real" at runtime, but we might still want to know about them.

For example, maybe we want to know how types are shaped at runtime:

Introspection on a struct

```
struct Foo {
    bar: u16,
    baz: String,
}

// We want something like:
let fields: StructFields = <Foo as Reflect>::fields()?;
```

Reflection

How do we do reflection in Rust?

- Store extra information in the vtable part of trait objects.

Reflection

How do we do reflection in Rust?

- Store extra information in the vtable part of trait objects.
- This generally means we end up with a trait `Reflect` that keeps track of the additional information.

Reflection

How do we do reflection in Rust?

- Store extra information in the vtable part of trait objects.
- This generally means we end up with a trait `Reflect` that keeps track of the additional information.
- This means that casting to `dyn Reflect` is what gets the information actually inserted at runtime.

Reflection

How do we do reflection in Rust?

- Store extra information in the vtable part of trait objects.
- This generally means we end up with a trait `Reflect` that keeps track of the additional information.
- This means that casting to `dyn Reflect` is what gets the information actually inserted at runtime.

Here is what we get just from Rust:

Any

```
pub trait Any: 'static {
    fn type_id(&self) -> TypeId;
}
```

Identifiers and registries

The `Any` trait can be made more useful by combining it with a type registry, which uses type identifiers as keys for retrieving type data at runtime.

Identifiers and registries

The `Any` trait can be made more useful by combining it with a type registry, which uses type identifiers as keys for retrieving type data at runtime.

For example:

bevy_reflect::TypeRegistry

```
pub struct TypeRegistry {
    registrations: TypeIdMap<TypeRegistration>,
    short_path_to_id: HashMap<&'static str, TypeId>,
    type_path_to_id: HashMap<&'static str, TypeId>,
    ambiguous_names: HashSet<&'static str>,
}
```

(`TypeIdMap<TypeRegistration>` ≈ `HashMap<TypeId, TypeRegistration>`)

Identifiers and registries

What kinds of things might go in a type registry?

- Just more information about the type — e.g., struct fields, enum variants, the type's path, etc. Basically, anything known about the type at compile time fits into this paradigm.

Identifiers and registries

What kinds of things might go in a type registry?

- Just more information about the type — e.g., struct fields, enum variants, the type's path, etc. Basically, anything known about the type at compile time fits into this paradigm.
- Auxiliary data for working with the type dynamically. This is much more subtle and often involves function pointers.

Identifiers and registries

What kinds of things might go in a type registry?

- Just more information about the type — e.g., struct fields, enum variants, the type's path, etc. Basically, anything known about the type at compile time fits into this paradigm.
- Auxiliary data for working with the type dynamically. This is much more subtle and often involves function pointers.
- Whatever you want!

Identifiers and registries

What kinds of things might go in a type registry?

- Just more information about the type — e.g., struct fields, enum variants, the type's path, etc. Basically, anything known about the type at compile time fits into this paradigm.
- Auxiliary data for working with the type dynamically. This is much more subtle and often involves function pointers.
- Whatever you want!

The main thing to keep in mind is that type registries are not magical and they are generally constructed at runtime (maybe one day the "life before main" story will change that somewhat).

Identifiers and registries: TypeRegistration

For example, here is bevy_reflect's type registration, which includes both shape/identity information (`TypeInfo`) and essentially arbitrary additional data which must be type-unique per type (`TypeData`):

bevy_reflect::TypeRegistration

```
pub struct TypeRegistration {
    data: TypeIdMap<Box<dyn TypeData>>,
    type_info: &'static TypeInfo,
}
```

Identifiers and registries: ReflectDefault

Here is a basic example of auxiliary data being used to implement a reflected version of `Default`:

bevy_reflect::ReflectDefault

```
pub struct ReflectDefault {
    default: fn() -> Box<dyn Reflect>,
}
impl ReflectDefault {
    pub fn default(&self) -> Box<dyn Reflect> {
        (self.default)()
    }
}
impl<T: Reflect + Default> FromType<T> for ReflectDefault {
    fn from_type() -> Self {
        ReflectDefault {
            default: || Box::<T>::default(),
        }
    }
}
```

Identifiers and registries: ReflectDefault

Here is how you would use such a thing (stolen from bevy_reflect docs):

bevy_reflect::ReflectDefault in practice

```rust
#[derive(Reflect, Default)]
struct MyStruct {
  foo: i32
}
let mut registry = TypeRegistry::empty();
registry.register::<MyStruct>();
registry.register_type_data::<MyStruct, ReflectDefault>();

let registration =
    registry.get(TypeId::of::<MyStruct>()).unwrap();
let reflect_default =
    registration.data::<ReflectDefault>().unwrap();

let new_value: Box<dyn Reflect> = reflect_default.default();
assert!(new_value.is::<MyStruct>());
```

Identifiers and registries

Using similar tricks (that is, storing function pointers in type-associated data), you can do things like casting from `Box<dyn Reflect>` to `Box<dyn Trait>` for types that implement `Trait`.

(And in bevy_reflect, this machinery can be derived for you for object-safe traits.)

Identifiers and registries

Using similar tricks (that is, storing function pointers in type-associated data), you can do things like casting from `Box<dyn Reflect>` to `Box<dyn Trait>` for types that implement `Trait`.

(And in bevy_reflect, this machinery can be derived for you for object-safe traits.)

Indirectly, by checking the type registry for this additional machinery, we can also check whether types implement a trait at runtime.

Limitations

This stuff is pretty nifty, but there are some shortcomings and limitations to keep in mind:

- Working with generics can be very thorny and generally leaves you with the choice of cherrypicking monomorphized types at compile time or trying to implement blanket solutions which treat the generic parameter opaquely (e.g. using `dyn Reflect`).

Limitations

This stuff is pretty nifty, but there are some shortcomings and limitations to keep in mind:

- Working with generics can be very thorny and generally leaves you with the choice of cherrypicking monomorphized types at compile time or trying to implement blanket solutions which treat the generic parameter opaquely (e.g. using `dyn Reflect`).
- The compiler doesn't know a lot of things that you know, so the code often involves both verbosity and a bunch of conversions, downcasting, and unwrapping. Basically, this isn't very Rusty.

Limitations

This stuff is pretty nifty, but there are some shortcomings and limitations to keep in mind:

- Working with generics can be very thorny and generally leaves you with the choice of cherrypicking monomorphized types at compile time or trying to implement blanket solutions which treat the generic parameter opaquely (e.g. using `dyn Reflect`).
- The compiler doesn't know a lot of things that you know, so the code often involves both verbosity and a bunch of conversions, downcasting, and unwrapping. Basically, this isn't very Rusty.
- Because of the orphan rule, you cannot implement reflection on foreign types unless you also own the reflection trait you're implementing, which can make interoperation painful or impossible.
- All of this has performance overhead.

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/talks`

And bevy_reflect is here:

`https://crates.io/crates/bevy_reflect`

Next time: `TypeInfo`, dynamic types, and dynamic data access.