GObject subclassing in Rust for extending GTK+ & GStreamer

Or: How to safely implement subclassing in Rust while making use of a C library

FOSDEM 2019

3 February 2019, Brussels

Sebastian 'slomo' Dröge
< sebastian@centricular.com >

Who?

What?

Subclassing or inheritance in Rust like in traditional OOP

But Rust does not support this!

But Object Oriented Programming sucks!

... or not?

So... why?

OOP is everywhere

Almost every major language is based on traditional OOP

It would be a shame to not be able to make use of all that existing code

We don't want to rewrite the whole world all at once!

So... why exactly?

- Interoperability with other platforms
  - E.g. GNOME/GStreamer or the HTML DOM
- Using existing OOP code/libraries
- Extending OOP libraries from Rust code
- Replacing existing libraries with Rust code
  - RIIR! 🦀

Rust is ideal for interoperability with other platforms

GObject

- C library for doing traditional OOP
  - Classes, interfaces, inheritance, virtual methods, RTTI, ...
  - Close to Objective-C type system
- Used by GNOME, GTK+, GStreamer and a lot of other code out there
- gobject-introspection! 
  - Automatic bindings for any* language
  - A **stable** OOP API/ABI

Using GObject from Rust

First some example code

```
let window = gtk::Window::new(gtk::window::Toplevel);
let button = gtk::Button::new();
button.set_label("test");
window.add(&button);
window.show_all();
```

How does it look under the hood?

Objects

- Conceptually like

```
struct Button(ptr::NonNull<gtk_ffi::GtkButton>)
```

- `Clone`/`Drop`: Reference counting
- Behaves like an `Rc<RefCell<_>>`
  - Interior mutability: This is OOP after all!
  - Includes weak references
- FFI ↔ Rust translation infrastructure

`impl` blocks

- For constructors and static functions only
  - Or `&self` methods for final types
- Directly calls into C functions

```
pub fn new() -> Button {
    unsafe { from_glib_none(ffi::gtk_button_new()) }
}
```

Ext **traits**

- Provide all `&self` methods
- Autogenerated
  - `ExtManual` traits are manual
- Implemented generically for all types that are
  - subclasses or interface implementors

```
impl<T: IsA<Container>> ContainerExt for T {
    fn add<W: IsA<Widget>>(&self, widget: &T) {
        unsafe {
            ffi::gtk_container_add(
                self.as_ref().to_glib_none().0,
                widget.as_ref().to_glib_none().0,
            );
        }
    }
}
```

`IsA<P>` marker trait

- Provides the subclass/implements interface relationship
  - `T: IsA<P>`
- Implies `T: AsRef<P>` and `T, P: ObjectType`
- Always use this for generic functions!

```
fn foo<T: IsA<P>>(f: &T) { ... }
```

`ObjectType` trait

- Implemented by all Object types
- Type-system mapping between Rust struct and FFI types
- Translation from/to raw pointer
- Access to `GObject` type ID via `StaticType` trait
- Requires all kinds of convenience traits

`Cast` trait

- Safe zero-cost upcasting, almost-free downcasting/dynamic casting
  - Safe: Runtime type checks if needed
  - Unsafe casts without checks
- Works via `mem::transmute()`
  - All Rust Object structs have the same memory representation

```
button
    .upcast::<gtk::Widget>()
    .downcast::<gtk::Button>()
    .expect("Not actually a button?");
```

Wrap-up

- `Ext` traits for methods, `impl` blocks for constructors
  - Mostly autogenerated
- All usage safe Rust
- Implicit upcasting, explicit downcasting
- Boilerplate autogenerated via `glib_wrapper!()` macro

```
glib_wrapper!(
    Object<Button, ffi::GtkButton,
        ffi::GtkButtonClass, ButtonClass>
    @extends Bin, Container, Widget, @implements Buildable
);
```

Code

If you want to look at the code yourself

- github.com/gtk-rs/gtk
  - `src/button.rs` (if manual code was necessary)
  - `src/auto/button.rs`
- github.com/gtk-rs/glib
  - `src/object.rs` for most of the infrastructure

Creating `GObject` subclasses from Rust

Generally

- In the `subclass` module of `glib`/etc crate
- Compared to C
  - Less boilerplate, but still quite some
  - Safer due to stronger type-system
  - Equally low overhead
- Lots of traits and generic functions again
- Might require `unsafe` code ☢

`ObjectSubclass` **trait**

- Mirror of `ObjectType` trait
- Type-mapping for FFI structs, type name
- Registration, class and instance initialization
- Translation from instance to impl type
  - Public `gtk::Button` vs. private `Button` impl
  - The trait is implemented on private impl

Example

```
impl ObjectSubclass for MyObject {
    const NAME: &'static str = "MyObject";
    type ParentType = glib::Object;

    type Instance = subclass::simple::InstanceStruct<Self>;
    type Class = subclass::simple::ClassStruct<Self>;

    glib_object_subclass!();

    fn class_init(klass: &mut Self::Class) { }
    fn new() -> Self {
        Self { ... }
    }
}
```

Instance and Class structs

- Has the parent type's as the first field
- Instance has public instance fields
- Class is basically the vtable
  - Function pointers for virtual methods
  - Defining new virtual methods requires `unsafe` ☢
- Empty ones available generically
  - See previous slide

`IsClassFor` & `IsSubclassable` traits

- Mapping from instance to class type (vtable!)
- `IsSubclassable` overrides virtual methods
  - C/Rust translation functions for each virtual method
  - Happens during class initialization automatically
  - Map to functions on the `Impl` trait

Example virtual method C/Rust translation function

```
unsafe extern "C" fn constructed<T: ObjectSubclass + ObjectImpl>
(
    obj: *mut gobject_ffi::GObject
) {
    let instance = &*(obj as *mut T::Instance);
    let imp = instance.get_impl();

    imp.constructed(&from_glib_borrow(obj));
}
```

Class-specific `Impl` traits

- Provide impls for virtual methods
  - Default impls for functions if optional
  - Functions to call into the parent impl
- `FooImpl` requires `BarImpl` for enforcing subclass relationship
- This is where everything interesting happens
  - You want to impl a `Button`? `ButtonImpl`!

Example `Impl` trait

```
pub trait ObjectImpl: 'static {
    fn constructed(&self, obj: &Object) {
        self.parent_constructed(obj);
    }

    fn parent_constructed(&self, obj: &Object) {
        unsafe {
            let data = self.get_type_data();
            let parent_class = data.as_ref().get_parent_class()
                as *mut gobject_ffi::GObjectClass;

            (*parent_class).constructed.as_ref().map(|func| {
                func(obj.to_glib_none().0)
            })
        }
    }
```

Remarks about memory layout

- Instance struct has parent first
  - Pointer can be casted
- Rust type (`glib::Object`, etc) uses this
- Impl struct is stored right before it
  - Same allocation
  - First base types, before that sub type
- No boxing or dynamic dispatch on the Rust side

Type registration and instance creation

- `glib::Object::new(T::get_type(), &[])`
  - `get_type()` registers type
- Use `glib_wrapper!` around this if needed

Code

If you want to look at some code yourself

What else is possible?

- `GObject` properties and signals are supported
- Virtual method definitions
- Class methods
- Interface impls and definitions
- Boxed types

The Future

More autogeneration of the C/Rust translation code

- A lot exists already
- But not for subclassing yet
- And not for various special cases

Support for more classes

- Usage-wise almost all covered
- Subclassing: only GStreamer and very basic otherwise

`gobject-class` procedural macro

- Allows writing a C#/Rust-style language for creating `GObject` subclasses
  - Not ready yet but slowly getting there
  - More convenient and removing more usage of `unsafe` code ☢

Making use of all this to write more things in Rust

Your chance to get involved!

- librsvg
- GStreamer plugins
- ...
- Your own ideas!

Thanks! Questions?

sebastian@centricular.com