Corroded Pipelines

Writing GStreamer Elements in Rust for Safety and Fun

GStreamer Conference 2016, Berlin

11 October 2016

Sebastian Dröge <sebastian@centricular.com>

Introduction

Who?

- Long-term GStreamer core developer and maintainer since 2006
- Did the last few GStreamer releases and probably touched every piece of code by now

- One of the founders of Centricular Ltd
  - Consultancy offering services around GStreamer, graphics and multimedia related software

What?

- What is Rust?
- Why should we care?
- Writing GStreamer elements in Rust

Rust

What is Rust?

- Type-safe, compiled, systems programming language
- Unique* safety guarantees
- Many modern language features usually seen in functional and scripting languages
- Zero-cost abstractions & no manual mem management but no GC
- Type system with inference, traits, generics and algebraic data types

- Targeting the same audience as C/C++

Rust is what C++ should have been*

* and what C++ tries to become, see C++ 17

Why?

- Writing safe code in C/C++ is hard
- Too many footguns to keep in mind
  - Memory management
  - Raw pointers, no type-safety
  - Undefined behaviour
- Let the compiler **help** us writing code instead of doing the job of an accountant
  - Strict ownership & mutability model
  - Compiler assisted, explicit error handling
  - But not a proof assistant!

Why? (2)

- Writing code in C/C++ is missing out on lots of ergonomic features of higher-level languages
  - Generalized “switch” – pattern matching / destructuring
  - (safe) closures & simple usage of higher-order functions
  - Algebraic Data Types – type-safe unions/enums on steroids
  - …
- Need a full featured standard library
- Also: GObject boilerplate!

Why? (3)

- Writing low-level, low-overhead code in high-level languages is hard
  - GC vs. (even soft) real-time requirements
  - VM overhead
  - Heavy runtime system
- Need full control over memory if needed
- Need full, cheap and easy access to hardware and existing C code if needed

Why? (4)

- More here in this free O'Reilly book

  http://www.oreilly.com/programming/free/why-rust.csp

- Also check the Rust website for all kind documentation, tutorials and other useful documents
  - https://www.rust-lang.org

Why should **we** care?

Handling Untrusted Data

- Parsing complicated data structures from untrusted sources
- Need to be more confident about handling broken data
  - Prevent buffer overflow
  - Memory bugs in error handling
- Majority of CVEs in multimedia related software are exactly that

- Bounds checking, automatic memory management, compiler assisted error handling, higher-level parsing abstractions, …

Multithreading

- GStreamer is heavily multi-threaded
- Getting multi-threaded code right in C is hard
  - Deadlocks, race conditions, concurrent data modifications, …
- Explicit mutability, mutexes protecting data instead of code, scope based unlocking, simpler to use threading primitives, move semantics, higher-level thread-safe data structures, ...

Ownership Model & Mutability

- Maps 1:1 to our memory, buffer, mini-object mutability model
  - One writable reference or multiple readable references
  - Copy-on-write
  - Built-in language and standard library constructs
- Can move runtime checks to the compiler
- Ownership transfer, etc expressed in the language instead of comments in the documentation

Convenience

- No GObject anymore – Number 1 complaint about GStreamer
- Batteries-included standard library
- More expressive language
  - More concise code
  - More explicit expression of intent in the code instead of documentation
  - No manual refcounting / memory management

But not a magic bullet!

- Your code will not be magically bug-free
  - Logic mistakes, (opt-in) unsafe code, …
- But a big category of common bugs can be prevented
- Need bindings for many languages via C
  - There are bindings generators

Writing GStreamer Elements

First the code

- Can be found here
  - https://github.com/sdroege/rsplugin/
- Currently contains
  - a file source/sink
  - HTTP source
  - Start of a demuxer
  - Some infrastructure

How does it work?

- GstElement subclass in C
- Registers the element factory
- Implements all virtual methods, pad event/chain/etc functions
- Each function just directly calls into Rust code
  - No actual logic
  - No translations, Rust FFI is using C ABI

How does it work? (2)

- Translation layer in Rust
- Handles GStreamer multi-threading
- Creates Rust wrapper objects
- Translates GStreamer concepts to a much more simplified API
  - Wrapper around a use-case specific Rust trait (“interface”)
  - Calls back into GStreamer as needed

How does it work? (3)

- Use-case specific Rust trait and implementations
  - Source, Sink, Demuxer
- Single-threaded (external) interface
- Implementation focussed on the use case and not obstructed by GStreamer internals
  - Think of GstVideoDecoder vs. FFMPEG's AVCodecContext

How does it work? (4)

- Wrappers for Buffer, Error messages, etc.
- Reimplementation of GstAdapter
  - So much simpler!

- Usage of Rust libraries for everything possible
  - URI handling, HTTP, parsers, ...

The Code

The Future?

Future? We will see!

- Looks very promising
- Let’s start writing elements in Rust now
  - Wrap GStreamer APIs on the way
  - Reuse Rust implementations as much as possible
- Let's experiment with nicer APIs for GStreamer elements
  - You should also watch Wim's talk before lunch!

- Maybe start rewriting GStreamer API in Rust
- Maybe write GStreamer 2.0 in Rust?

Thanks!

Any questions?