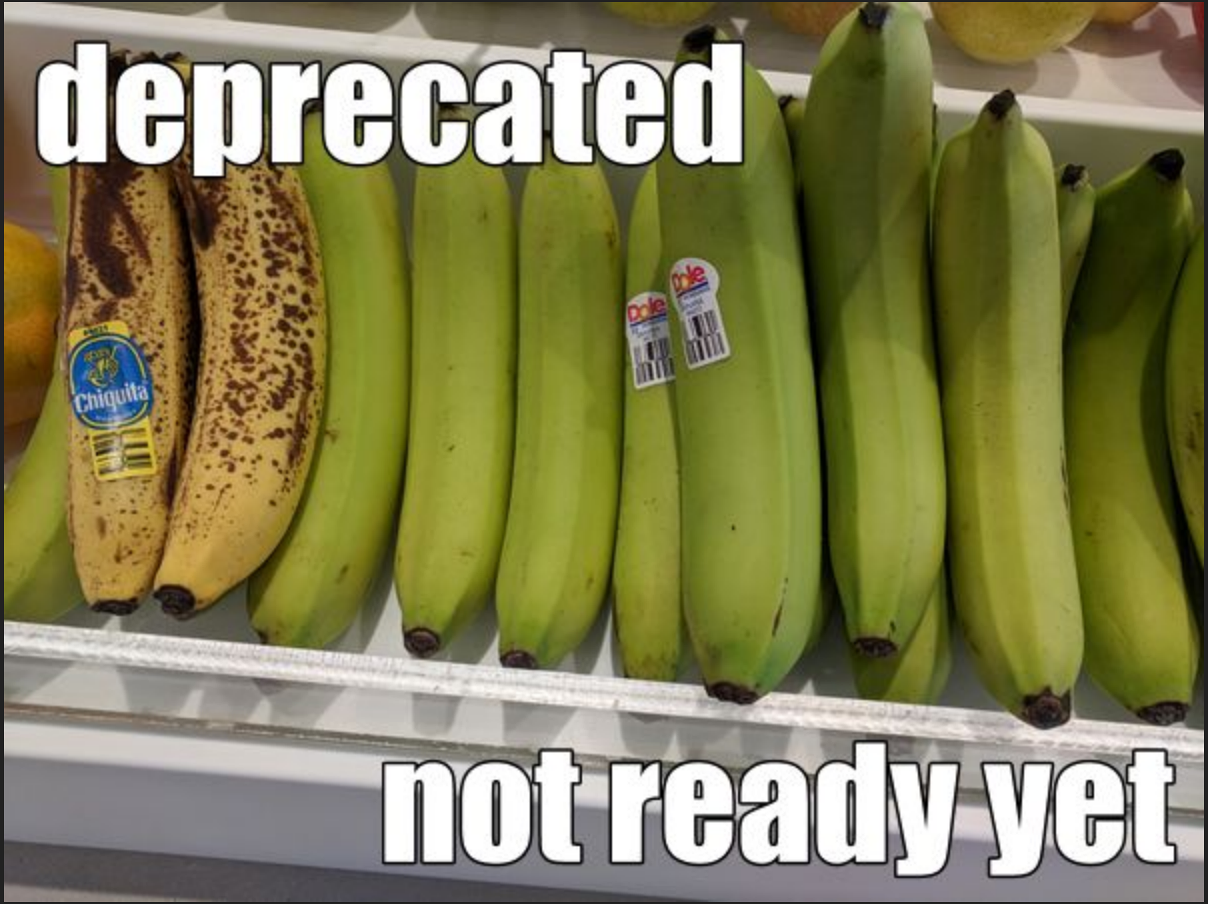
🦀 Rust + WASM 🦀

state of the ecosystem

J Pratt (jopra@)
With support from Michael Martin (mykmartin@)
With **a lot** of ideas from Cameron Silvestrini (csilvestrini@)

Tried using Mozilla's new lang again
But I'm a bit rusty

Spoilers


deprecated
Chiquita
Dole
Dole
not ready yet

The plan

- Build two wasm modules
- Have them communicate via a runtime ‘host’
- Have this perform some task
- Set up that communication via shared memory (with zero copy!)
- Profile +/ assess different communication strategies
    - Hand rolled communication over byte array/vector
    - Rust ‘native’ Serialization/Deserialization library (I chose Serde)
    - Flatbuffers
    - Cap’n protos

How it went

- Build two wasm modules
- Have them communicate via a runtime ‘host’
- Have this perform some task
- Set up that communication via shared memory (with zero copy!)
- Profile +/ assess different communication strategies
    - Hand rolled communication over byte array/vector
    - Rust ‘native’ Serialization/Deserialization library (I chose Serde)
    - Flatbuffers
    - Cap’n protos

What did I get working?

(and why did it take a week?)

Do we have time for a demo?

Building Wasm

- `wasm-pack build --release`
- `wasm-pack build --target bundler --release`
- `wasm-pack build --target no-modules --release`
- `wasm-pack build --target web --release`
- `wasm-pack build --target nodejs --release`
- `cargo build --release --target wasm32-unknown-unknown`
- `cargo build --release --target wasm32-wasi`
- `cargo wasi build --release`
- WASMer (not to be confused with Wasmr)
    - 3 options for compilers
        - Wasmer-compiler-singlepass
        - Wasmer-compiler-cranelift
        - Wasmer-compiler-llvm
    - 2 options of engines
        - Wasmer-engine-jit
        - Wasmer-engine-native
    - 2 environments
        - Wasm via wasmer-wasi
        - Emscripten via Wasmer-emscripten
    - Lots to learn about to work effectively in this space
- Cranelift? (alternative to LLVM)

What tools are available?

Web

```
wasm-pack build --release
```

This assumes using

- web-pack
- wasm-bindgen
- JS to do marshalling / orchestration

Native

```
cargo wasi build --release
```

This assumes using

- wasi
    - which doesn’t support wasm-bindgen*

What tools are available?

Web

```
wasm-pack build --target web --release
```

This assumes+allows using

- wasm-bindgen
- JS to do marshalling / orchestration

JS code generated is now more accessible & it’s possible to replace the initialisation code

...but it still doesn’t give us the ability to set up memory the way we want.

Native

```
cargo wasi build --release
```

This assumes using

- wasi
    - which doesn’t support wasm-bindgen*

Gotchas (web)

Using wasm-pack generates JS glue code with a bunch of drawbacks

- With --target bundler
  - Initialization code is built into the web-pack-dev-server (it's not accessible or modifiable)
  - Initialization code is tightly coupled with the runtime code (i.e. you can't modify much, it's not really configurable)
    - This means the shared memory story is basically non-existent
- With --target web
  - Initialization code is still tightly coupled, but is at least observable
  - Init & runtime code uses side effects and global variables
    - A set of caches that provide 'views' into the Wasm linear memory
    - WASM_VECTOR_LEN
      - set when returning a ptr to a slice of memory
      - i.e.

        ```
        WASM_VECTOR_LEN = len;
        return ptr;
        ```
      - Could just be

        ```
        return  [ptr, len];
        ```

Gotchas (web)

Using wasm-pack generates JS glue code with a bunch of drawbacks

- Additionally, the pairing of JS and Rust led to memory issues
  - Memory could be cleared when sending data from Wasm to JS via a reference (breaking Rust memory safety)
  - Wasm-pack & wasm-bindgen generate unsafe code for encoding +/ decoding but don't mark it as unsafe
    - Deserialization can break without being wrapped in unsafe or Result yielding surprising panics
- Still that's probably still better than writing unsafe code by hand...

Gotchas (native)

Using wasm-bindgen with Wasi

Source: Running wasm-bindgen - The cargo-wasi Subcommand

> “**Note**: Usage of `wasm-bindgen` and WebAssembly Interface Types is highly experimental,
>
> it's recommended that you expect breakage and/or surprises if you're using this.”

Gotchas (native)

Using wasm-bindgen with Wasi

Source: [Running wasm-bindgen - The cargo-wasi Subcommand]

> “**Note**: When building your crate with WebAssembly Interface Types enabled via `wasm-bindgen`, due to a bug in `wasm-bindgen`, it is currently necessary to build in release mode, i.e., `cargo wasi build --release`.”

Gotchas (native)

Using wasm-bindgen with Wasi

Source: [Running wasm-bindgen - The cargo-wasi Subcommand]()

“The `wasm-bindgen` project is primarily targeted at JavaScript and the web, but is also becoming the primary experiment grounds of WebAssembly Interface Types for Rust.

If you're not using interface types you probably don't need `wasm-bindgen`, ...”

Old one is deprecated,
new one is not ready yet.
Always two there are. No more. No less.

Gotchas (native + wasi) See https://wasi.dev/

Requires some libraries that aren't mentioned in tutorials

```
[dependencies]
# Lets us run wasm from Rust
# Needed for Wasmtime's error messages
anyhow = "1.0.40"
wasmtime-wasi = "0.26.0"
# This is the surprising one
wasi-cap-std-sync = "0.26.0"
```

Boilerplate needed for:

- set up the host
- allow panic to have stack traces
- turning on logging

Gotchas (native + wasi) See https://wasi.dev/

Requires writing unsafe code by hand
(or finding a library to do that for you)

One example: https://github.com/tearust/binio

[I’m still looking into using the techniques in this library +/ demo]

Optimisations!

Mostly it’s simple: use the defaults, add `--release` when you’re not debugging

As for tools:

`cargo wasi build --release` seems to give the best optimisation passes

`cargo build --release --target wasm32-unknown-unknown` gives the smallest output

You can combine these!

```
cargo wasi build -Zmultitarget --release --target wasm32-unknown-unknown
```

(‘obviously’ /s)

Optimisations via cargo config files!

In wasm module Cargo.toml

```
wee_alloc = { version = "0.4.5", optional = true }

[profile.release]
# opt-level="s" # optimises for small code but in practice O3 gives the same size and is 30x faster
panic = 'abort' # lose the code for stack unwinding
lto=true # slow, but can ~half code size & give a nice speed up
```

🎉 Plug for sccache (a build cache tool, really handy, especially when building multiple crates)

Takeaways 🥡

Wasm is:

- Ready for 1 * JS app containing n * Wasm module(s)
- Ready for 1 * JS app containing n * Wasm module(s) that inter-communicate
    - You have to provide the wrappers for this, but it’s possible [very Arcs-y]
- Ready for 1 * Rust app containing n * Wasm module(s)

- Capable of small n-copy (n<4) without serialize-deserialize with a shared representation
- FAST (ish)!
    - Capable of call times of:
        - under 2 microseconds doing simple string manipulation and a little memory movement
        - under 40 microseconds when also doing some logging
        - ~100 millisecond set up time [there may be ways to improve this]

Takeaways 🥡

Wasm is not:

- Ready for 1 * Rust app containing n * Wasm module(s) that inter-communicate
    - This is getting close [there’s work on linking between modules etc]
    - You can provide wrappers like in JS but there’s limitations due to memory safety (JS doesn’t fix this, it just leaves making it safe as an exercise for the programmer)
- Ready for zero copy +/ setting up shared memory in Rust
    - This may be possible in C/C++
- Memory safe (the foreign function interface is [currently] inherently unsafe)

Takeaways 🥡

Wasm needs:

- Support for **non-web Wasm** usage:
  - The Isolation is great
  - ...but the web is the main client
  - ...and Mozilla is the main driver for:
    - Wasm
    - Cranelift
    - Rust
- True **Wasm type & memory safety**
  - interface types (proposal)
  - memory safety across boundaries

Wasm would benefit from:

- **Memory + initialization configuration**
  - Performance from **shared memory**
  - Isolation via restricted access to memory
  - See the ‘**multi-memory**’ feature
- More **hygienic code generation**
  - **Generate separable [sub] modules**
  - Generated code should be easy to use
  - Shouldn’t rely on web-pack

Shout out to erickt for their profiling work!

https://github.com/erickt/rust-serialization-benchmarks

```
test goser::bench_clone                         ... bench:         333 ns/iter (+/- 129) = 1645 MB/s
test goser::bincode::bench_decoder              ... bench:       1,399 ns/iter (+/- 571) = 285 MB/s
test goser::bincode::bench_encoder              ... bench:         135 ns/iter (+/- 43) = 2962 MB/s
test goser::bincode::bench_populate             ... bench:         878 ns/iter (+/- 116)
test goser::bincode_serde::bench_deserialize    ... bench:       1,188 ns/iter (+/- 460) = 301 MB/s
test goser::bincode_serde::bench_populate       ... bench:         900 ns/iter (+/- 312)
test goser::bincode_serde::bench_serialize      ... bench:         170 ns/iter (+/- 46) = 2105 MB/s
test goser::capnp::bench_deserialize            ... bench:         344 ns/iter (+/- 56) = 1302 MB/s
test goser::capnp::bench_deserialize_packed     ... bench:         812 ns/iter (+/- 360) = 415 MB/s
test goser::capnp::bench_populate               ... bench:         644 ns/iter (+/- 344)
test goser::capnp::bench_serialize              ... bench:          32 ns/iter (+/- 19) = 14000 MB/s
test goser::capnp::bench_serialize_packed       ... bench:         564 ns/iter (+/- 307) = 597 MB/s
test goser::msgpack::bench_decoder              ... bench:       2,234 ns/iter (+/- 831) = 128 MB/s
test goser::msgpack::bench_deserializer         ... bench:       2,686 ns/iter (+/- 1,117) = 106 MB/s
test goser::msgpack::bench_encoder              ... bench:         784 ns/iter (+/- 471) = 366 MB/s
test goser::msgpack::bench_populate             ... bench:       1,063 ns/iter (+/- 471)
test goser::msgpack::bench_serializer           ... bench:         922 ns/iter (+/- 183) = 311 MB/s
test goser::protobuf::bench_decoder             ... bench:       2,016 ns/iter (+/- 554) = 141 MB/s
test goser::protobuf::bench_encoder             ... bench:         779 ns/iter (+/- 444) = 367 MB/s
test goser::protobuf::bench_populate            ... bench:         908 ns/iter (+/- 264)
test goser::rustc_serialize_json::bench_decoder  ... bench:      30,541 ns/iter (+/- 7,753) = 19 MB/s
test goser::rustc_serialize_json::bench_encoder  ... bench:       3,469 ns/iter (+/- 1,583) = 174 MB/s
test goser::rustc_serialize_json::bench_populate ... bench:       1,010 ns/iter (+/- 400)
test goser::serde_json::bench_deserializer      ... bench:       4,726 ns/iter (+/- 2,393) = 128 MB/s
test goser::serde_json::bench_populate          ... bench:         949 ns/iter (+/- 216)
test goser::serde_json::bench_serializer        ... bench:       1,966 ns/iter (+/- 692) = 307 MB/s
```

Thanks 🎉

More info at go/arcs-rust+wasm-experiment

For anyone interested in working on shared memory:
Here’s the wasmtime Memory types we’d need to construct:
VMMemoryDefinition
ExportMemory