Interface Types & WASI

WASI 2021-07-29

WASI & Interface Types today

- Not using interface types, but wants to!
- Defined with custom `*.witx` format and type grammar
- Inching towards IT over time
  - Use `(expected ..)` as return values
  - Start making `handle` types more first class
  - Initial effort towards a "v2 ABI" of WASI
- Moving to IT will change the ABI (signature) of existing functions
- Multiple code generators for IT (wiggle, Rust's `wasi` crate, wasi-libc's generator, …)

What are interface types?

*The proposal adds a new set of **interface types** to WebAssembly that describe high-level values. The proposal is semantically layered on top of the WebAssembly core spec and can be implemented in terms of an unmodified core wasm engine.*

github.com/WebAssembly/interface-types

What are interface types?

- Used to define interfaces
- Language agnostic
- Abstract representation
- Supports virtualization
- Seamless language support

Interface Types Overview

- Not formally specified (yet!)
  - Current Explainer.md is a bit dated
- Based on module linking
- Based on component model
- No binary format representation (yet!)
- No engine implementation (yet!)
- Type grammar specifics a bit in flux

Interface Types Type Grammar

```
intertype ::= f32 | f64 | s8 | u8 | s16 | u16 | s32 | u32 | s64 | u64
            | char
            | (list <intertype>)
            | (record (field <name> <id>? <intertype>)*)
            | (variant (case <name> <id>? <intertype>?)*)
            | (push-buffer <intertype>)
            | (pull-buffer <intertype>)
            | (handle <resource>)
```

Interface Types Type Grammar

```
                                string ≡ (list char)

                   (tuple <intertype>*) ≡ (record ("i" <intertype>)*) for i=0,1,...

                        (flags <name>*) ≡ (record (field <name> bool)*)

                                  bool ≡ (variant (case "false") (case "true"))

                         (enum <name>*) ≡ (variant (case <name>)*)

                  (option <intertype>) ≡ (variant (case "none") (case "some" <intertype>))

                   (union <intertype>*) ≡ (variant (case "i" <intertype>)*) for i=0,1,...

(expected <intertype>? (error <intertype>)?) ≡ (variant (case "ok" <intertype>?)
                                                         (case "error" <intertype>?))
```

Interface Types Type Grammar

- All values are always "valid"
  - records have all their fields
  - variants are always one of the cases
  - strings/chars are always valid USV (no replacement characters needed)
- Handles are un-forgeable references
  - When imported, refer to resources owned by a "someone else"
  - When exported, always receive a valid handle
  - Deterministically managed
- Push/pull buffers
  - Most likely to change in future updates
  - Like handles, always valid
  - Intended for `fd_{read,write}`

Interface Types - witx

```
(typename $clockid
  (enum (@witx tag u32)
    $realtime
    ;; ...
  )
)
(module $wasi_snapshot_preview1
  (@interface func (export "clock_time_get")
    (param $id $clockid)
    (param $precision $timestamp)
    (result $error (expected $timestamp (error $errno)))
  )
)
```

Interface Types - witx

```
enum clockid {
    realtime,
    // …
}
enum errno { /* … */ }
type timestamp = u64

clock_time_get: function(id: clockid, precision: timestamp)
  -> expected<timestamp, errno>
```

Interface Types - witx

- New syntax not s-expression based
- Does not 100% correspond to an adapter module
- Describes something to import, or export, not both
- Optimized for readability
    - no s-expressions
    - Comments
    - Name resolution
    - Import types/resources between files
    - "source of truth" for definition and code generators
    - "compiles to" interface types

Interface types & Canonical ABI

- Values need to be represented somehow in each language
- Adapter functions seen as "gonna take awhile to stabilize"
- Temporary stop-gap, a "canonical ABI" all languages use
  - In the future each language can customize representation
  - Mostly matches the wasm C ABI
  - Provides a way to use interface types *today* with engines

Interface types & Canonical ABI

- Scalars/Records/Variants - "do what C does"
- Strings - specialized from `list<char>` to be utf-8 or utf-16
- Lists - all interface types have an in-memory representation, lists are pointer/length to contiguous in-memory representations
- Handles/{push,pull}-buffer - use imported intrinsics to implement

```
(module
 (import "canonical_abi" "resource_new_$resource" (func (param i32)  (result i32)))
 (import "canonical_abi" "resource_get_$resource" (func (param i32)  (result i32)))
 (import "canonical_abi" "resource_clone_$resource" (func (param i32)  (result i32)))
 (import "canonical_abi" "resource_drop_$resource" (func (param i32)))

 (import "canonical_abi" "push_buffer_len" (func (param i32) (result i32)))
 (import "canonical_abi" "push_buffer_push" (func (param i32 i32 i32) (result i32)))
 (import "canonical_abi" "pull_buffer_len" (func (param i32) (result i32)))
 (import "canonical_abi" "pull_buffer_pull" (func (param i32 i32 i32) (result i32)))
```

Interface types & Canonical ABI

- Memory management done through malloc/free:

```
(module

  (func (export "canonical_abi_realloc") (param i32 i32 i32 i32) (result
i32))

  (func (export "canonical_abi_free") (param i32 i32 i32))

)
```

- Arguments to imports need no memory management

Interface types & Canonical ABI

- Adapter "glue" between modules performs validation
  - strings are valid utf-8 (or utf-16)
  - variants match one case
  - u8/u16 values smaller than i32 are in-bounds (no extra bits set)
  - handles are valid - the module providing the handle actually owns said handle
- Glue is "trusted code"
  - Either performed by the host for host <-> wasm communication
  - Or synthesized by a "wasm linker" for wasm <-> wasm communication

witx-bindgen

- Initial implementation of interface types using the canonical ABI
- github.com/bytecodealliance/witx-bindgen
- Supported languages
  - wasm - Rust
  - wasm - C
  - host - Rust (wasmtime)
  - host - JS (browser, node, …)
  - host - Python (wasmtime)
  - misc - Markdown (doc generator)
- Self-hosting demo of generated code
  - bytecodealliance.github.io/witx-bindgen

witx-bindgen

```
witx_bindgen_rust::import!("wasi_snapshot_preview1.witx");
use wasi_snapshot_preview1::*;


fn main() {
    let code: Exitcode = 2;
    proc_exit(code);
}
```

witx-bindgen

```
$ witx-bindgen js --import wasi_snapshot_preview1.witx
Generating "bindings.js"
Generating "bindings.d.ts"
```

```
// my_lib.js
import { addWasiNextToImports } from "./bindings.js"
const myWasi = {
     procExit(code) { throw new WasiExit(code); }
     // …
};
const myImportObject = {};   // filled in with any other host imports
addWasiNextToImports(myImportObject, myWasi, export_name => instance.exports[export_name]);
await WebAssembly.instantiateStreaming(fetch('./my_file.wasm'), myImportObject);
```

witx-bindgen

- Generate idiomatic code in each language
- Don't require raw memory manipulation
- Each bindings mode responsible for upholding IT invariants/guarantees
- Details of the canonical ABI hidden and not something you worry about
- Relatively easily extensible to new languages

What's next?

- Wasm CG agreement on the canonical ABI
- Binary format for interface types
    - based on module linking
    - … which is re-envisioned as "adapter modules"
- Define next WASI snapshot in terms of interface types
    - The function signatures of the next snapshot determined by the canonical ABI
    - Meaning of all types (e.g. "what is a string?") defined by interface types
    - New snapshot written with new `*.witx` syntax
    - Functions using `(@witx pointer T)` migrated to `{push,pull}-buffer` as appropriate
    - Tooling likely to be shuffled around and/or refactored as appropriate