Introduction

What is `main`?

`main` is a function like this:

```
int main(int argc, char *argv[])
```

It takes a variety of types of data:

- Integers, booleans, actual strings, etc.
- Enums, subcommands, etc.
- Files, network connections, services, etc.

However, these are all communicated as strings.

Filenames

- User has a file: 😸
- They type in the name: `"cat.jpg"`
- Program gets the name: `"cat.jpg"`
- Program opens the file: 😸

What problems does parsing cause?

Commmand-line parsing is inconsistent and error-prone.

Is `-ab` the same as `-a -b`?

Is `-a -b` the same as `-b -a`?

Also, *security* and *virtualization*:

- Child process needs same filesystem view as parent
- Child process needs permissions of parent
- Child process hard-codes how to turn strings into streams

We can sandbox child processes, but we often don't.

What's different about WebAssembly?

Is WebAssembly Assembly?

Wasm is like an ISA, but different.

General-purpose CPUs have converged in many areas:

- 8-bit bytes
- Two's complement
- IEEE 754 floating-point

But also:

- Memory is a big virtual address space of bytes
- Calls are just jumps (-and-link) with register and memory conventions
- Syscalls are mode switches and jumps

In WebAssembly, the address space isn't everything.

Calls arguments are part of the call instruction.

WebAssembly has a static type system

It has an up-front validation step, rather than just SIGILL on thy fly.

Two perspectives

Minimize the differences to maximize compatibility?

Or take advantages of the differences to do new things?

Types

MVP types: `i32`, `i64`, `f32`, `f64`

WASI is about interfaces, so we also look forward to Interface Types:

- signed and unsigned integers
- `bool`
- lists
- variants
- records
- strings, aka lists of characters
- handles

Handles

The way we represent handles in wasm will likely evolve over time.

- Integer indices into tables

- `externref`
- Typed imports

At the witx level, we can just use the `handle` type.

Signatures

Functions have signatures.

Programs are functions.

What is the signature of a native program?

- The binary doesn't say.
- The OS doesn't know.
- The shell doesn't know (in general).

Unix imposes a single effective signature on all programs.

Typed Main lets programs declare their signatures.

Command-line usage

Command-line parsing for Typed Main programs happens in the Wasm engine.

Example: an `f32` argument

The user might type `"6.283185"`

In some locales, the user might type `"6,283185"`.

Or "0x1.921fb5p+2"`.

Parsing in the Wasm engine means that all programs have a consistent interface.

Example: a handle argument

Many programs read files, but they don't literally need *files*.

An "input stream", that supports `read` would often be enough.

Programs written this way:

- Do one thing and do it well (waves to Unix)
- Don't depend on a particular filesystem view
- Don't depend on filesystem privileges
- Don't depend on a filesystem at all!

Bonus:

- No implied string comparisons! No:
  - Non-Unicode filenames
  - Unicode normalization
  - Case sensitivity
  - Windows special-case path parsing
  - Filename length limits
  - ...

Putting it all together

Typed Main programs are just programs.

With signatures, they're also just functions.

We can use this to compose multiple programs together.

Compatibility with existing code

We have three options.

Option A: Out of the box

For porting an existing application with no changes, things work like they do in WASI today:

- Everything is strings
- User needs to use `--dir` preopens

This uses Typed Main, but with a fixed signature.

- List-of-strings for the args
- List-of-(handle,string) for the preopens

Option B: Provide a witx description

In this option, the Developer writes a witx file to describe their application.

This option involves no changes to the program itself.

The main program will still take strings and use preopens, but it can be wrapped in a wasm interface generated form witx.

Example:

```
;; Typed main example: a simple grep

(module $grep
  ;;; Main entrypoint for grep.
  (@interface func (export "main")
    ;;; The string to search for.
    (param $pattern string)

    ;;; The output to write to.
    (param $output $output_byte_stream)

    ;;; The inputs to search for it in.
    (param $inputs (list $input_byte_stream))

    ;;; The result: Just indicate if any I/O failed.
    (result $error (expected unit (error $input_byte_stream_error)))
  )
)
```

Option C: Typed Main in the source language

What if programming languages let you just write a `main` function which took arbitrary types?

Nameless

Nameless is a Rust crate prototyping Option C, for native code:

https://github.com/sunfishcode/nameless

https://crates.io/crates/nameless

Nameless today works for native code by doing everything itself. But once we hook it up to Typed Main in WASI, it'll be a very thin API.

```
/// # Arguments
///
/// * `pattern` - The regex to search for
/// * `inputs` - Input sources
#[kommand::main]
fn main(
    pattern: Regex,
    output: LazyOutput<OutputTextStream>,
    inputs: Vec<InputTextStream>,
) -> anyhow::Result<()> {
    let mut output = output.materialize(Type::text())?;

    let print_inputs = inputs.len() > 1;

    for input in inputs {
        let pseudonym = input.pseudonym();
        for line in BufReader::new(input).lines() {
            let line = line?;
            if pattern.is_match(&line) {
                if print_inputs {
                    output.write_pseudonym(&pseudonym)?;
                    write!(output, ":")?;
                }
                writeln!(output, "{}", line)?;
            }
        }
    }

    Ok(())
}
```

Wrap up

Current Status

The **witx** syntax shown here is supported by the witx parser in the WASI tree.

New-style commands, a toolchain feature that can be one of the building blocks:

- https://reviews.llvm.org/D81689
- https://github.com/WebAssembly/wasi-libc/pull/203
- https://github.com/rust-lang/rust/pull/82924

Interface types are in phase 1, and there is prototyping underway at a few different levels.

Typed Main uses in WASI

wasi-clocks

Clocks should be capabilities, rather than ambient authorities.

wasi-random

Entropy sources should be capabilities, rather than ambient authorities.

New WASI proposals

- Serial ports
- Audio devices
- Database connections
- etc.

A big question for all capability systems is how a program obtains the first capability. Typed Main is a way to do this.