WebAssembly Components:
The Modular Polyglot Ecosystem We Need

Robin Brown (she/her)

WebAssembly (Wasm) is…

a platform-agnostic **“compile target”** or…

something you can compile programs to.

Host

- The thing that runs the Wasm
- Creates the sandbox

Guest

- The thing you compiled to Wasm
- Lives inside the sandbox

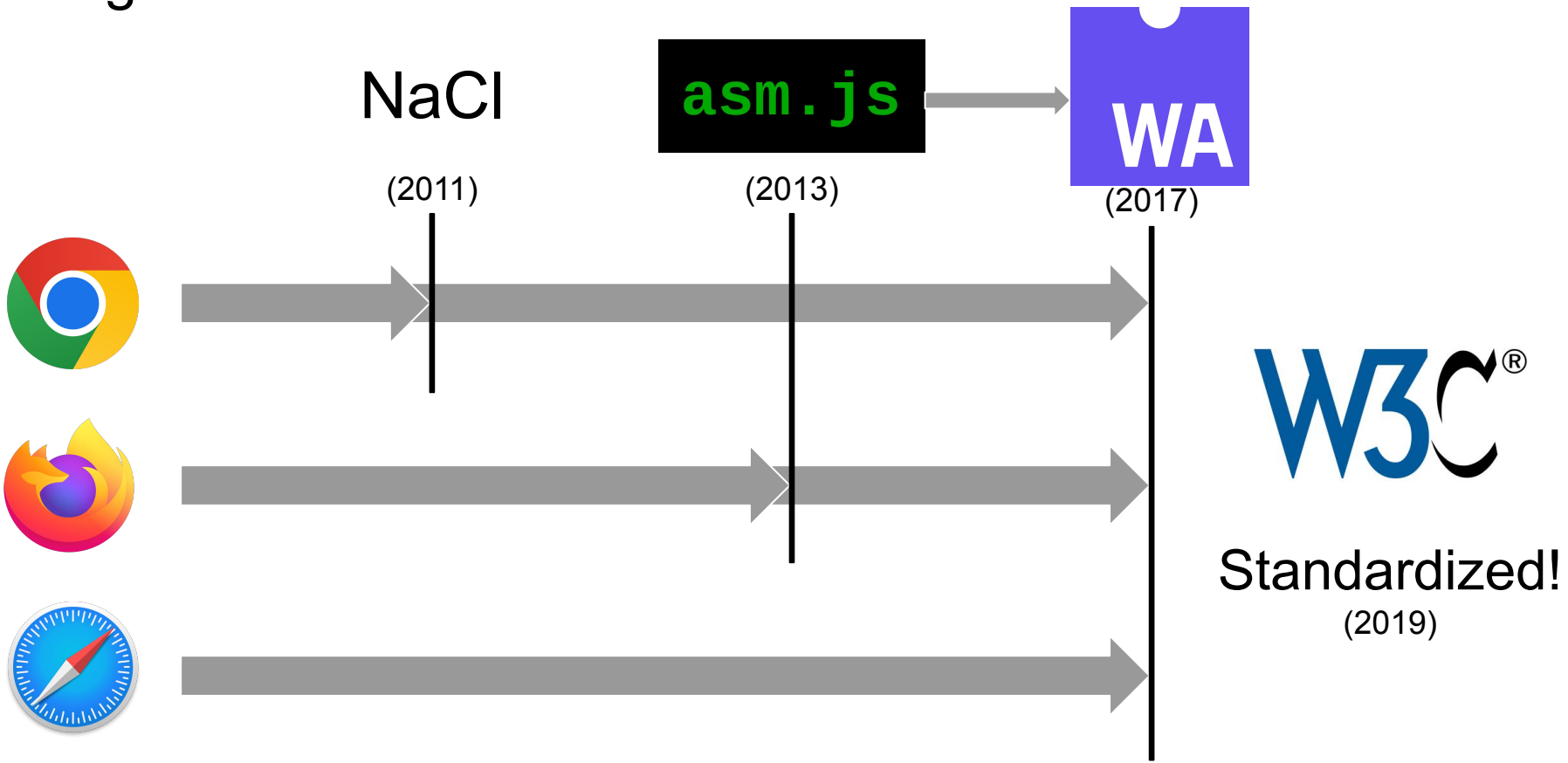
Host	Guest
Browsers (e.g. Firefox, Google Chrome)	Graphically-Intensive Apps (e.g. Photoshop)
Databases (e.g. SingleStore)	Extensions and User-Defined Functions
Web Servers (e.g. WasmCloud, Spin)	Distributed Apps, Serverless Functions, etc.

Overview

- History
- Core WebAssembly
- Component Model
- WASI
- Language Support
- Composition
- Ecosystem
- Demo

History

Origins in the Browser



Wasm’s Broadly-Useful Properties

Portability	Speed	Security
• Platform agnostic • Not specialized to one Operating System	• Low startup latency • Near-native performance	• Capability safety • Sandboxing & memory isolation

Wasm leaves the browser

CLOUD AND SYSTEMS

How Prime Video updates its app for more than 8,000 device types

The switch to WebAssembly increases stability, speed.

By Alexandru Ene

January 27, 2022

Share

WebAssembly on Cloudflare Workers

10/01/2018

How Shopify Uses WebAssembly Outside of the Browser

by Duncan Uszkay • Development

Dec 18, 2020 • 8 minute read

July 21, 2022

[r]evolution Summer 2022: Bring Application Logic to Your Data With SingleStoreDB Code Engine for Wasm

HarfBuzz 8.0 Released - Introduces Shaper For WebAssembly Within Font Files

Written by Michael Larabel in Desktop on 9 July 2023 at 05:57 AM EDT. 28 Comments

Core Wasm

Core Wasm

	Defines	Abstraction Level
Core WebAssembly	Modules	Numbers, Memory, …

Types

- Numbers
  - `i32`
  - `i64`
  - `f32`
  - `f64`
- References
  - `funcref`
  - `externref`
- Functions
  - `vec(valtype) → vec(valtype)`

Module Binary Format

- **Header**
  - magic number
  - version
- **Non-Custom Sections**
  - fixed order
  - each is optional
- **Custom Sections**
  - allowed before, after, and between normal sections
  - any bytes can go here!
  - toolchains may ignore or remove these!

https://webassembly.github.io/spec/core/binary/modules.html

Sections: Types & Imports

Section	Description
Type	The defined types (e.g. function types)
Import	use funcs, tables, memories, and globals from outside

Sections: Declarations

Section	Description
Function	The defined functions
Table	Tables of functions for dynamic dispatch
Memory	The defined linear memory spaces
Global	The defined global values (e.g. u32, i64, f32)

Sections: Entry Points

Section	Description
Export	Binds external names to funcs, tables, memories, and globals
Start	Indicates a function to run to initialize state

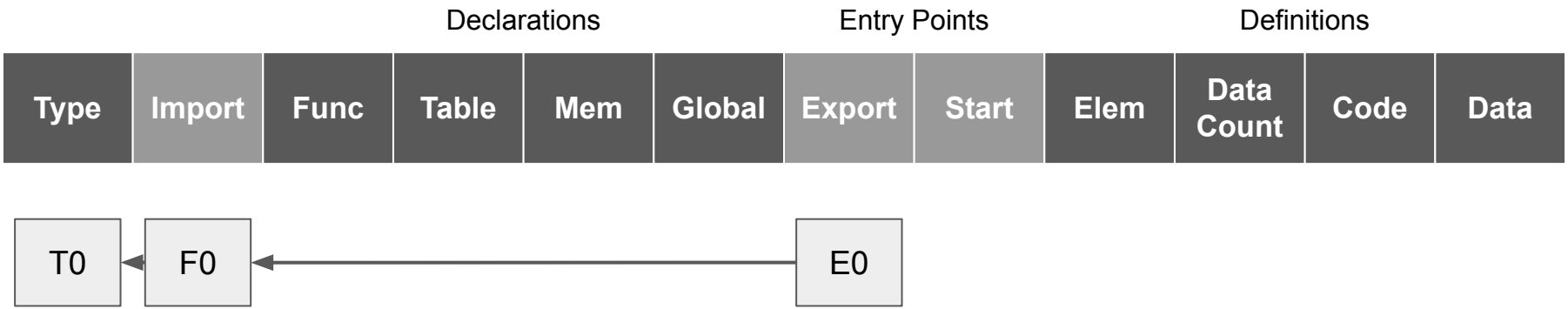
Sections: Definitions

Section	Description
Element	The function elements to populates tables with
Data Count	The number of data segments
Code	The instruction sequences that implement the functions
Data	The byte sequences to populate memories with

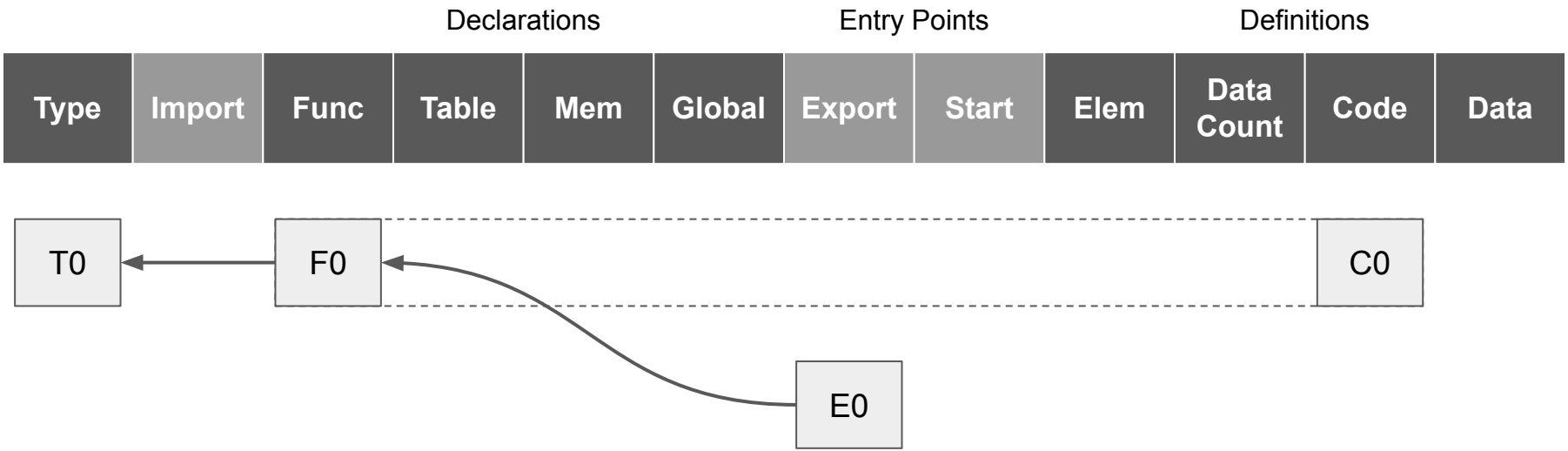
Sections

			Declarations				Entry Points		Definitions		
Type	**Import**	**Func**	**Table**	**Mem**	**Global**	**Export**	**Start**	**Elem**	**Data Count**	**Code**	**Data**

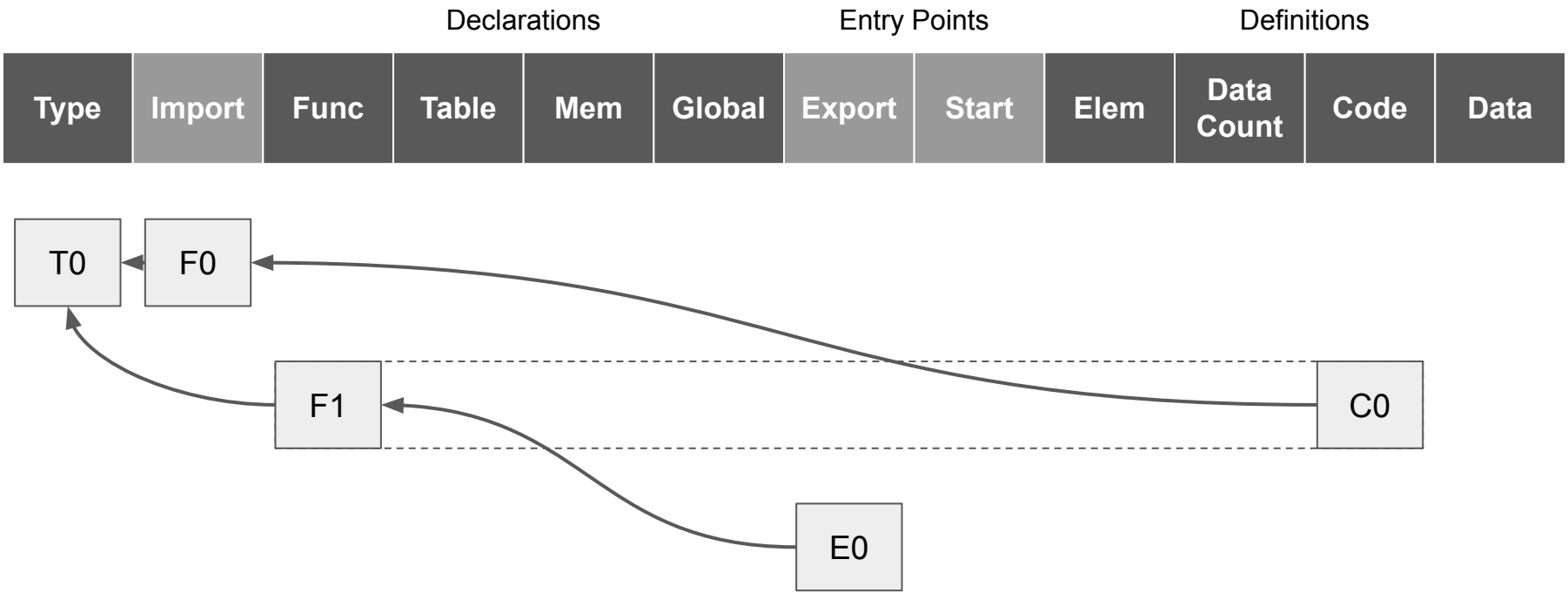
Exporting an Imported Function



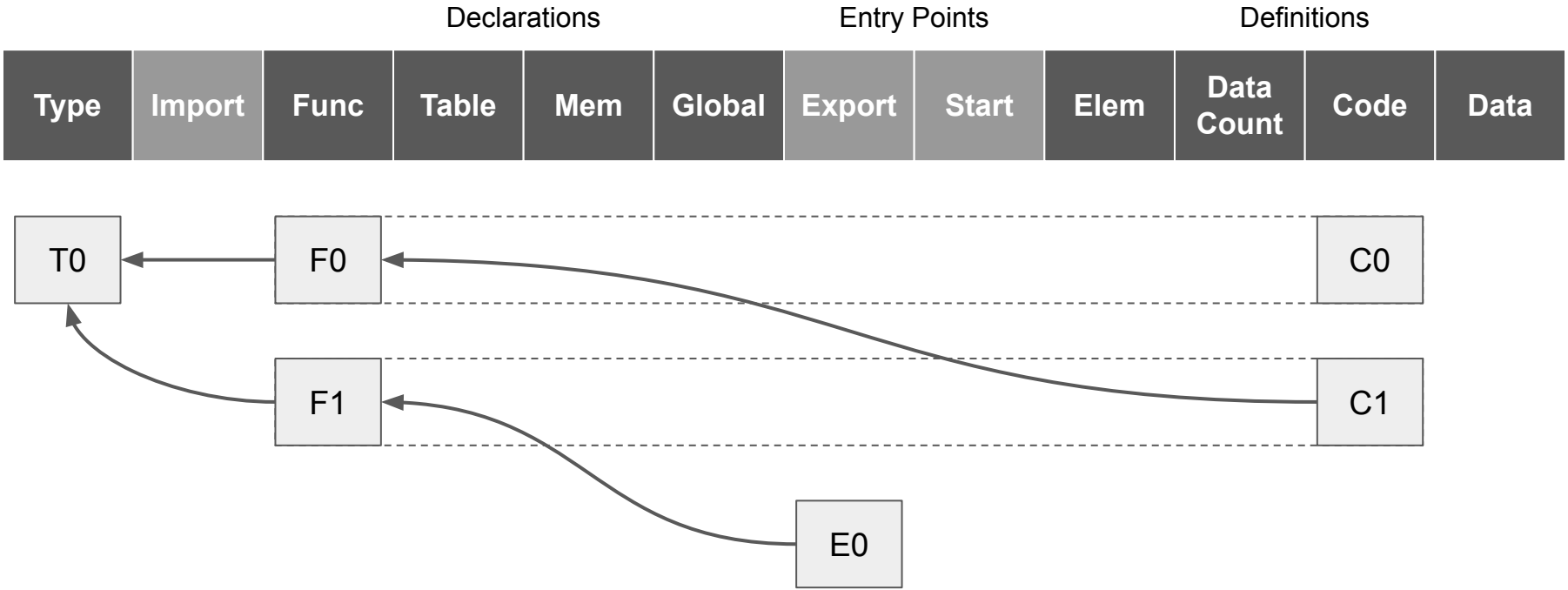
Exported Function



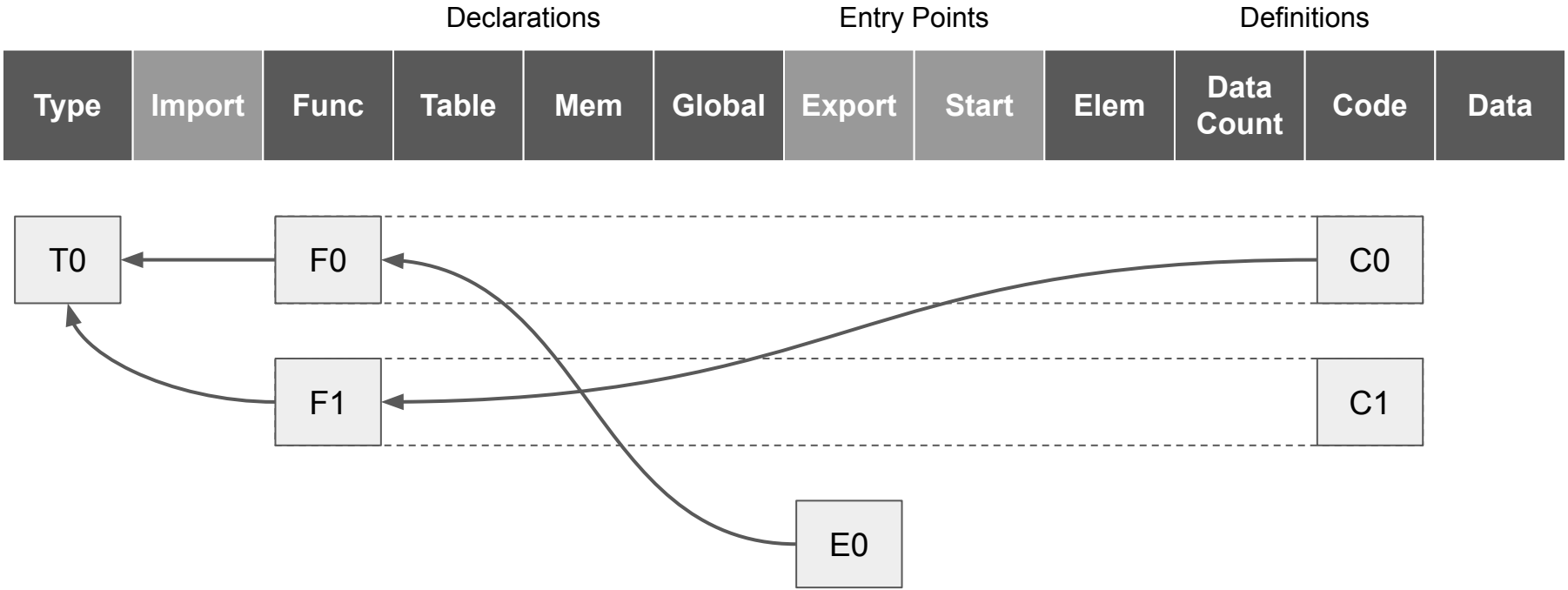
Exported Function Calling Imported Function



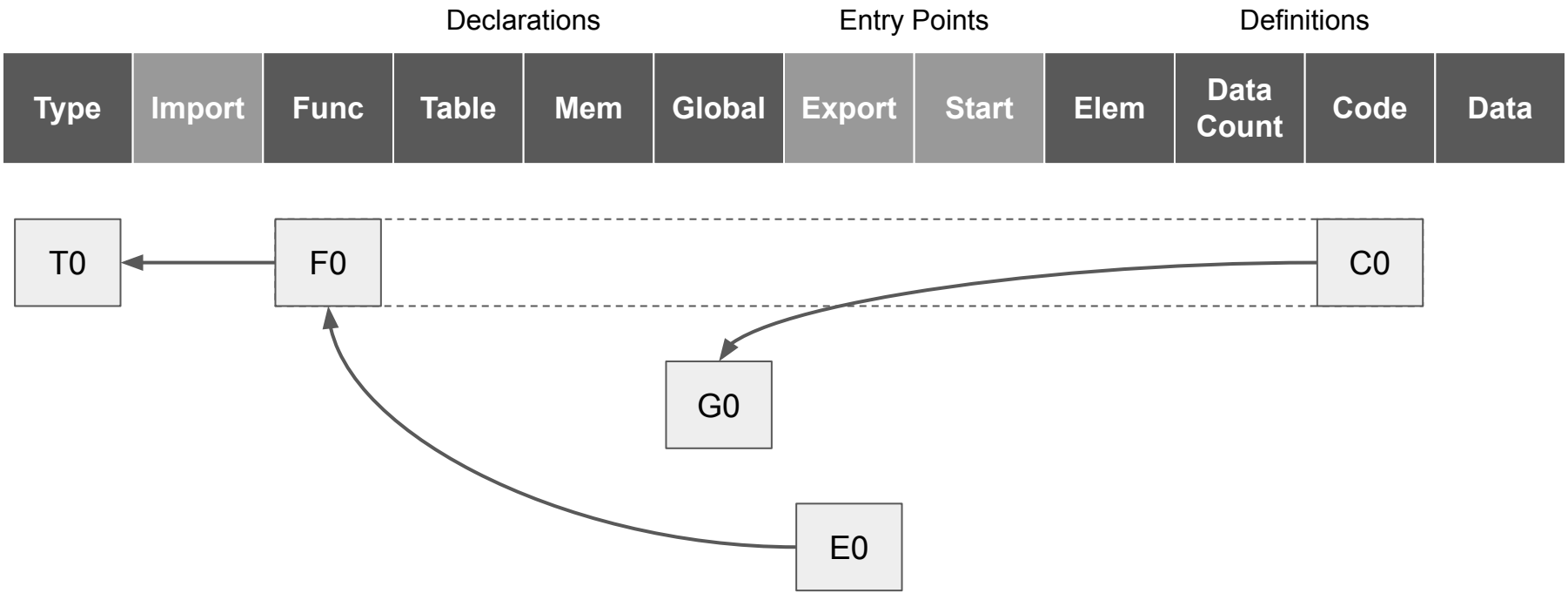
Exported Function Calling an Earlier Function



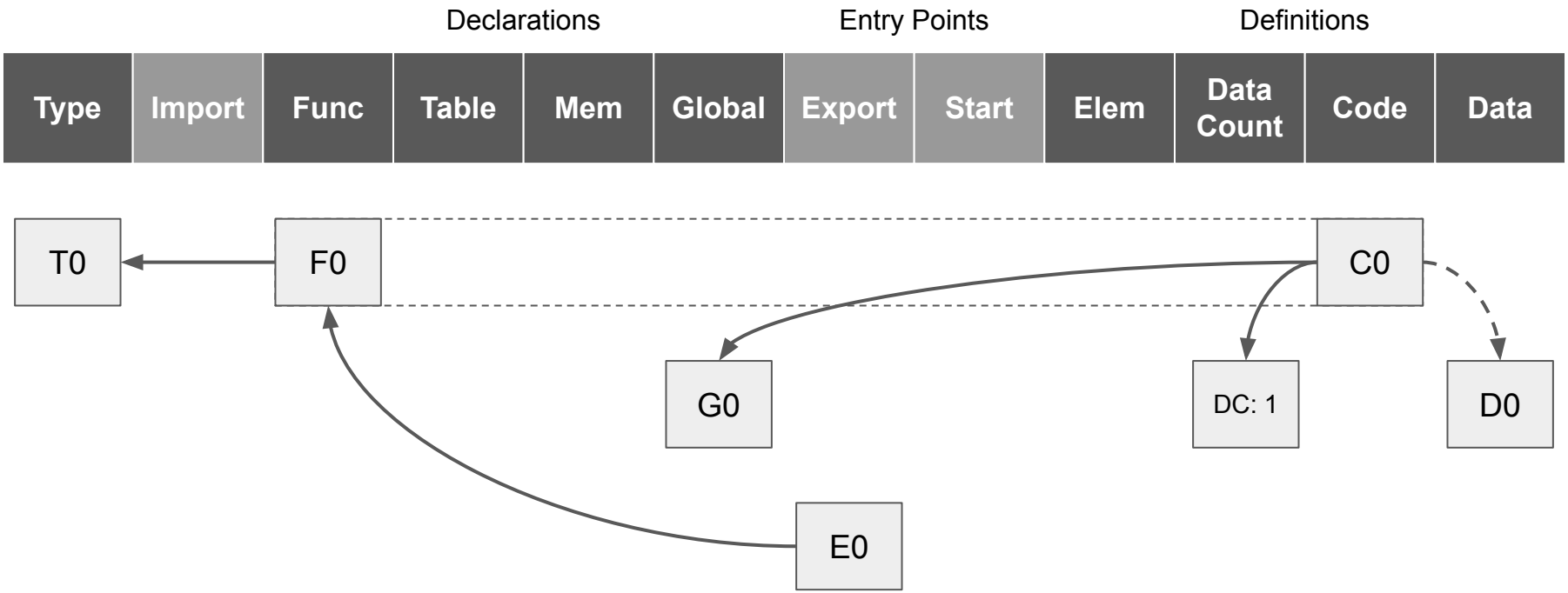
Exported Function Calling Later Function



Exported Function Using a Global



Exported Function Using Globals and Data



Note: Code and data make up the vast majority of **module** size

Index Spaces

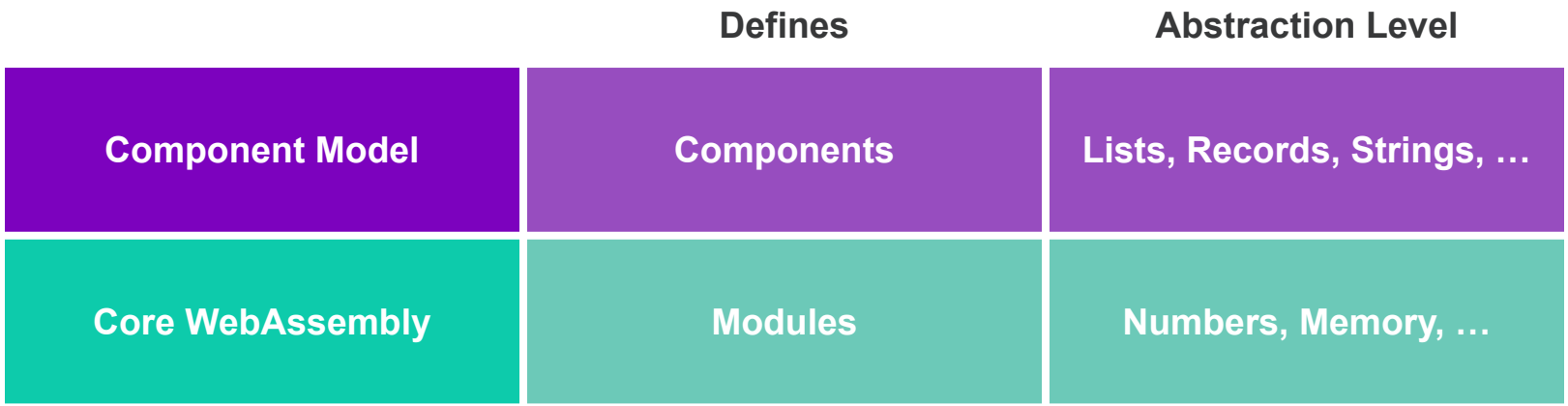
- Imports and definitions share the same index spaces
- Module-level index spaces
  - Types
  - Functions
  - Tables
  - Memories
  - Globals
  - Elements
  - Data
- Function-level index spaces
  - Locals
  - Labels

Instruction Set

- Stack-machine
  - Instructions pops and push values onto the implicit operand stack
  - No instructions for duplicating or re-ordering values!
  - Each function can have statically-defined locals
- Kinds of instructions
  - Arithmetic (add, sub, mul, div, etc.)
  - Locals and globals (local get/set, global get/set)
  - Control (block, if, loop, branch, etc.)
  - Call and Call Indirect
  - Memory (size, grow, load, store, copy, fill, etc.)

Wasm Components

Component Model



Component Value Types

- Boolean
  - `bool`
- Numbers
  - Integers
    - Unsigned (**`uNN`**) or Signed (**`sNN`**)
    - Sizes: 8, 16, 32, and 64 bits
  - Floating Point:
    - Same sizes as core wasm: **`f32`** and **`f64`**
    - Not guaranteed to preserve NaN bit-pattern
- Text
  - `char` - Unicode Scalar Value (USV)
  - `string` - Sequence of USVs
- Sequence
  - `list<T>`

Component Value Types (cont.)

- Composite Types
  - `record` - named and has named fields
  - `tuple` - anonymous and has numbered fields
- Bitsets
  - `flags` - a compact collection of boolean values
- Multi-case
  - `enum` - enumerated type with no payload
  - `variant` - enumerated type with payload
  - `option<T>` - value that can be `some(T)` or `none`
  - `result<T, E>` - value that can be `ok(T)` or `err(E)`

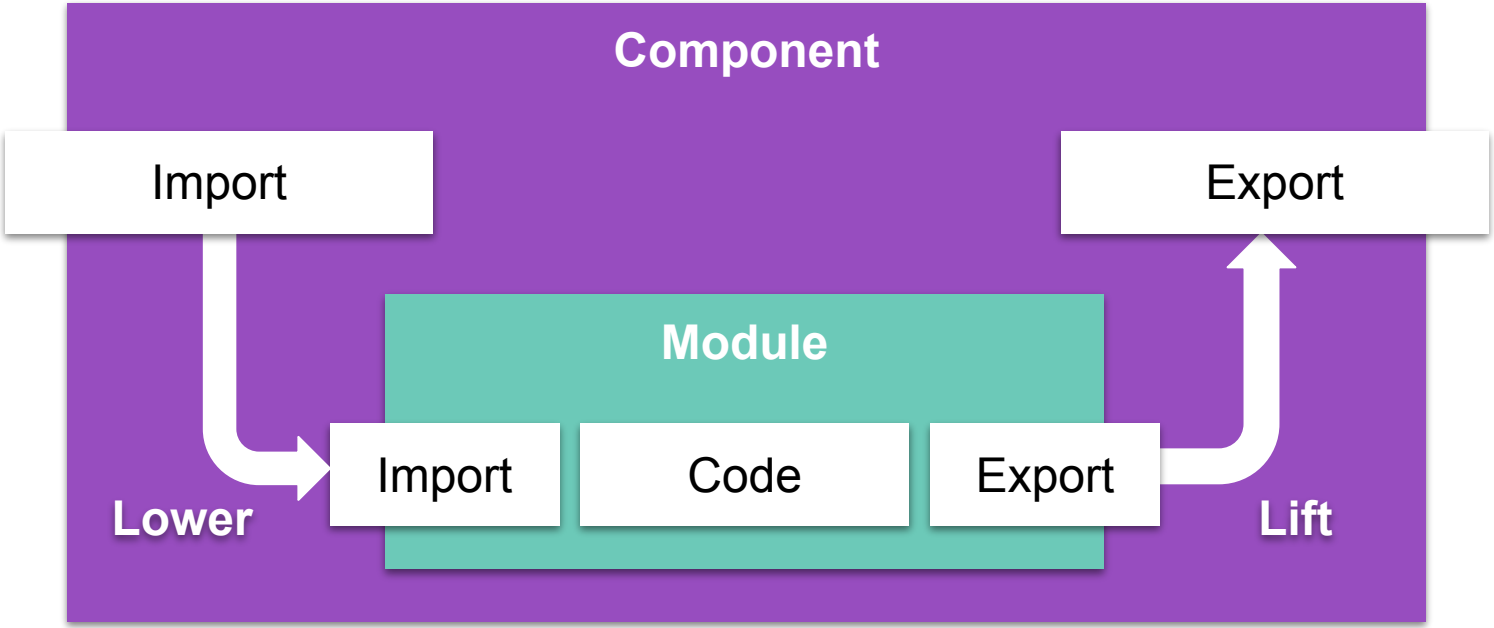
Component Value Types (cont.)

- Resources - `resource`
  - Opaque handles
  - Methods can be called on them
  - Used to represent
    - System resources like files or sockets
    - Data you want to share access to without copying

Wasm Interface Types (WIT) IDL

```
world test {
    import foo: func() -> string;
    export test-interface;
}
interface test-interface {
    bar: func() -> string
}
```

Component Model



Component Binary Format

- **Header**
  - magic number
  - version
- **All Sections**
  - Optional, repeatable and allowed in any order
- **Custom Sections**
  - any bytes can go here!
  - toolchains may ignore or remove these!

https://github.com/WebAssembly/component-model/blob/main/design/mvp/Binary.md

Component Sections

Section	Description
Type	The defined types (e.g. function types)
Import	Use funcs, instances, etc. from outside
Lower	Map a function to a core function

Component Sections (cont.)

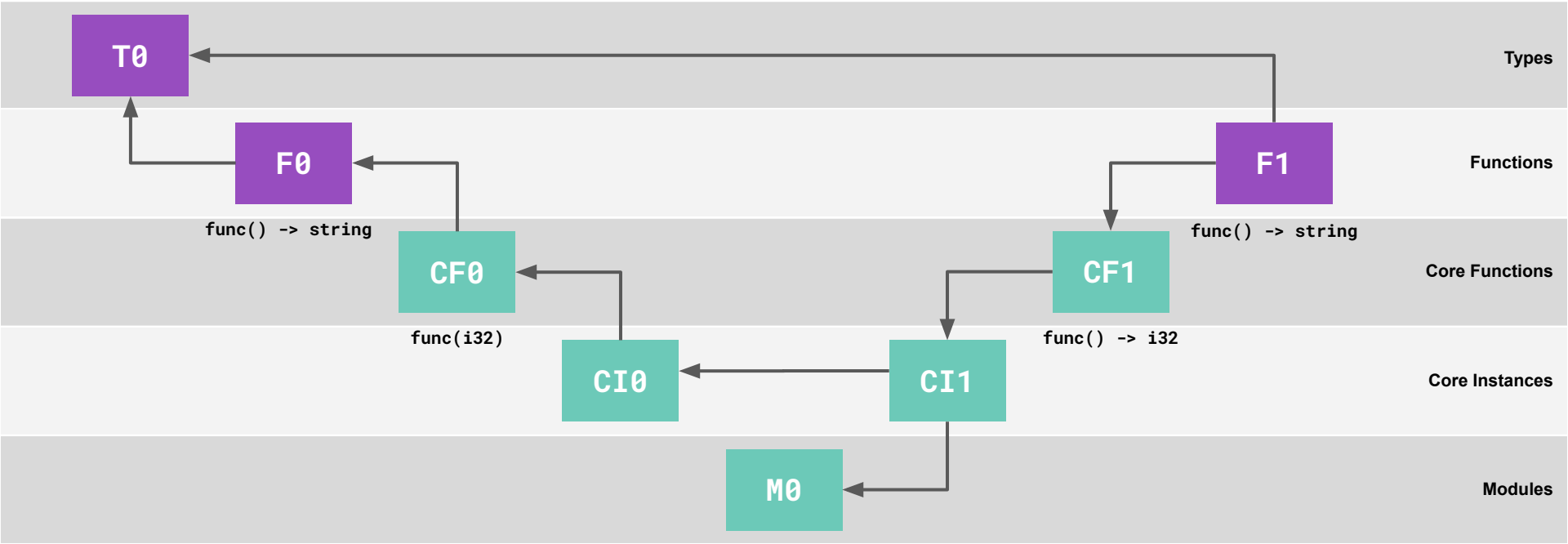
Section	Description
Instance	Create a module instance from a module or loose core functions
Module	An actual Core WebAssembly Module
Alias	Bring an export from an instance into this components index space

Component Sections (cont.)

Section	Description
Lift	Map a core function to a component function
Export	Associate a function with a public name

Exported Function Calling Imported Function

Type	Import	Lower	Instance	Module	Instance	Alias	Lift	Export
`func() -> string`	`"ifunc" of type T0`	`string-encoding=utf8`	`CF0 as "ifunc"`	Inline Core Module	`M0 with CI0 as "in"`	Use `"efunc"` from I1	`string-encoding=utf8`	`"efunc"`



Note: Inline **modules** make up the vast majority of **component** size

WASI

What is WASI?

- WASI is a **W3C Subgroup**
- Focused on creating **standardized interfaces**
- WASI is essentially the “standard library” for Wasm
- WASI is made up of individual **proposals**
  - Each proposal is made up of WIT documents and specification
  - Each proposal advances through a phase process

WASI 0.2

- A set of worlds and interfaces that met release criteria
- WASI 0.2 acts as a stable foundation for language support to build on

- **`wasi:http/proxy`**
  - Basic server
  - Handles requests
  - Is able to send outbound requests
- **`wasi:cli`**
  - Basic command line application
  - Exports a run command
  - Imports file-system, sockets, clocks, random, etc.

WASI's Async Journey

- **WASI 0.2** does not have native Component-Model async
  - Instead `wasi:io` defines resource types for polling
- **WASI 0.3** will be built on top of native Component-Model async
  - It will come with built-in `stream` or `future` types
  - WASI interfaces will use these instead of defining their own resources for them

We will be able to automatically adapt **0.2** components into **0.3** components, implementing `wasi:io` in terms of native C-M async!!

Language Support

Languages that can produce Wasm

Languages that can produce Wasm

WIP Component Support

BYTECODE
ALLIANCE

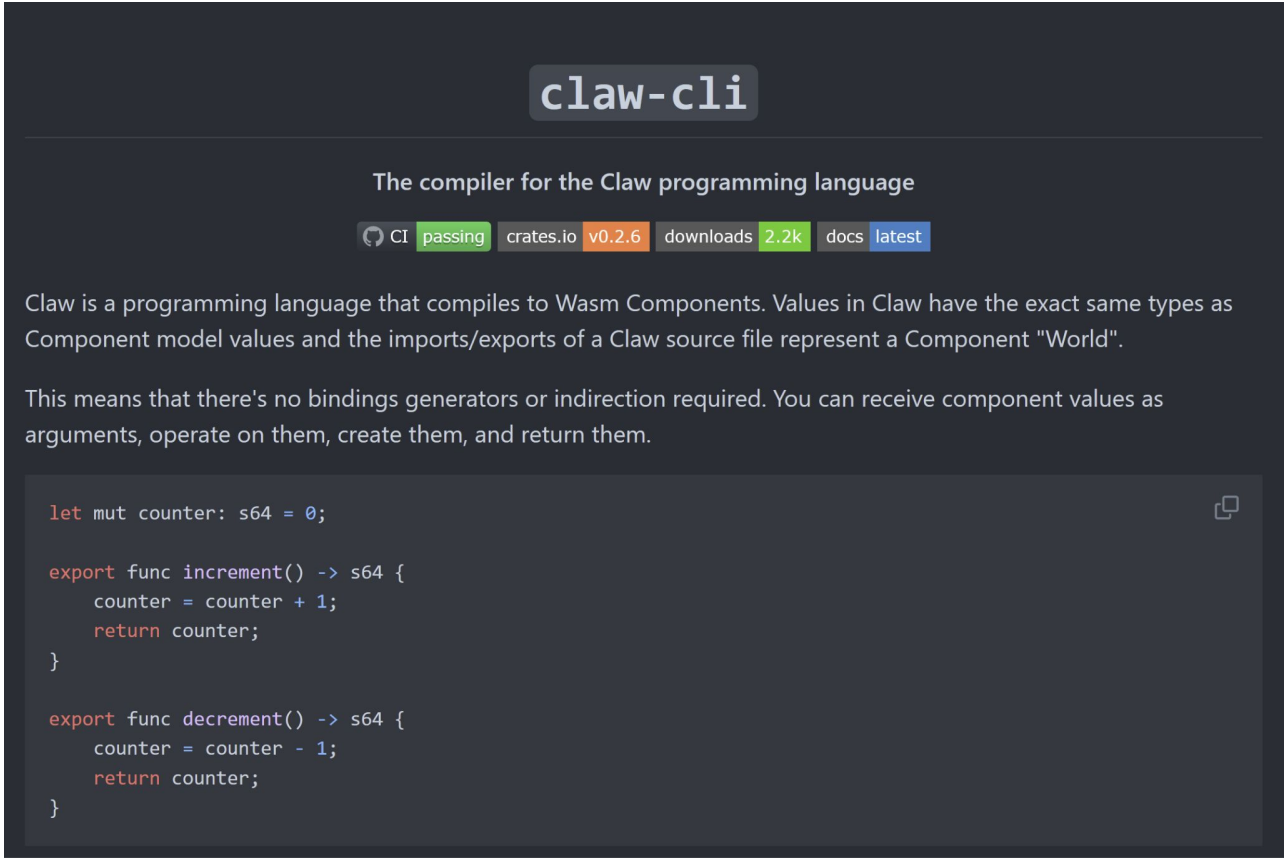
Bytecode Alliance Special Interest Groups

- Debugging
- Registries
- Guest Languages
- Documentation
- Community
- …

Guest Languages SIG Subgroups

- Rust
  - Cargo Component
- Python
  - componentize-py
  - CPython WASI target
- JavaScript
  - componentize-js
  - StarlingMonkey
- C#
  - Contributions to Mono and NativeAOT
- Go
  - Contributions to TinyGo and “Big” Go

Claw



claw-cli

The compiler for the Claw programming language

CI passing | crates.io v0.2.6 | downloads 2.2k | docs latest

Claw is a programming language that compiles to Wasm Components. Values in Claw have the exact same types as Component model values and the imports/exports of a Claw source file represent a Component "World".

This means that there's no bindings generators or indirection required. You can receive component values as arguments, operate on them, create them, and return them.

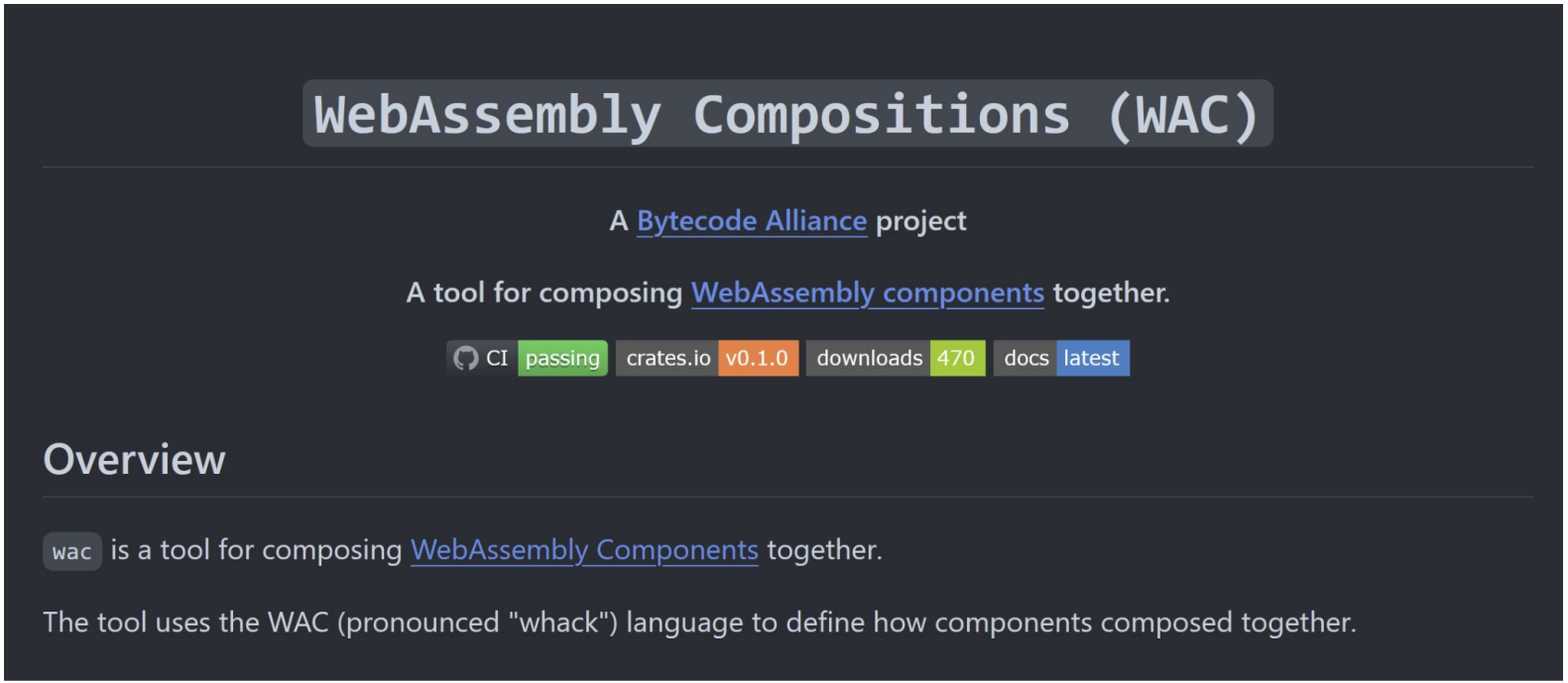
```
let mut counter: s64 = 0;

export func increment() -> s64 {
    counter = counter + 1;
    return counter;
}

export func decrement() -> s64 {
    counter = counter - 1;
    return counter;
}
```

https://github.com/esoterra/claw-lang

Composition

A composition of **components** is itself a **component**!!

WAC



https://github.com/bytecodealliance/wac

WAC Language

```
package example:composition;

// Instantiate the `name` component
let n = new example:name {};

// Instantiate the `greeter` component by plugging its `name`
// import with the `name` export of the `name` component.
let greeter = new example:greeter {
  name: n.name,
};

// Export the greet function from the greeter component
export greeter.greet;
```

Component Ecosystem

Wasmtime

- Wasm runtime with Component-Model support
- Created by the Bytecode Alliance and written in Rust
- Uses the **`cranelift`** project to AoT compile Wasm
- Lets you use components from
  - CLI with **`serve`** (**`wasi:http`**) and **`run`** (**`wasi:cli`**) commands
  - The published Rust **`wasmtime`** crate
  - **Note:** *The C API does not yet support components*

Open Source Wasmtime-derived Hosts

- **WasmCloud** (CNCF)
  - Runs distributed applications
  - Built on top of NATS
- **Spin** (Fermyon)
  - Runs serverless functions
  - Also usable through Kubernetes with SpinKube
- **NGINX Unit** (F5)
  - Application server
  - Runs standalone or with NGINX

JavaScript Component Tools (JCO)

- **Transpiles** components to JS + Core Wasm!!
  - Browsers, node, deno, etc. can’t currently directly run components
  - But jco transpile can split up a component into modules and JS glue!!
- Provides Wasmtime-equivalent **run** and **serve** command
- Makes **wasm-tools** available as a library and CLI in the node ecosystem
- Provides a CLI for **componentize-js**

Warg

- Wasm component **registry protocol**
- It's **federated**, there hopefully won't be **"***an NPM***"**
- Offers exciting **Supply Chain Security** features
  - Based on ideas from Certificate Transparency
  - Offers what we call "Package Transparency"
- Implementations
  - wa.dev by JAF Labs
  - Official in-repo reference implementation

https://github.com/bytecodealliance/registry

Workspaces On Wasm (wow)

- Run **dev tools** as Wasm!!!
  1. Write a **`wow.kdl`** file with the tools you want specified
  2. Initialize your workspace
  3. Call your tools on the command line like normal!!
- Supports **`wasi:cli`**
- Executes commands in **Wasmtime** sandbox
  - Tools can only access files in your workspace
  - Tools can’t access the network without permission
  - Tools can’t access the shell

https://github.com/esoterra/wow

Demo Time!!

https://github.com/esoterra/UCSC-LSD-2024

Thanks!

github.com/**esoterra**

linkedin.com/**esoterra**

@**esoterra**@hachyderm.io

esoterra.dev

What a Claw Component Actually Looks Like

Module	Instance	Alias	Alias	Type	Import	Lower	Instance	Module	Instance	Alias	Lift	Export
Allocator Module	No extra arguments	Use “mem” from CI0	Use “alloc” from CI0	func() -> string	“ifunc” of type T0	string-encoding=utf8	CF0 as “ifunc”	Code Module	M1 with CI0 as “alloc” and CI1 as “in”	Use “efunc” from I0	string-encoding=utf8	“efunc”