Scoping and Layering the Module Linking and Interface Types proposals

WebAssembly CG

April 27th, 2021

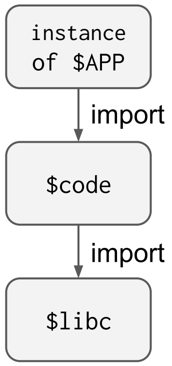
Outline

- Background context
- Proposed scope
- Proposed use cases and requirements
- Proposed next steps
  - … which include carving out an MVP that could reach Stage 3 / Developer Preview this year
- Discussion + Polls:
  - Does the general proposed direction sound good?
  - Should we proceed with the proposed next steps?

Current Module Linking proposal summary

- Module Linking proposes a host-independent way to link wasm modules
- Currently, linking is host-dependent and only spec’d for JS and C APIs

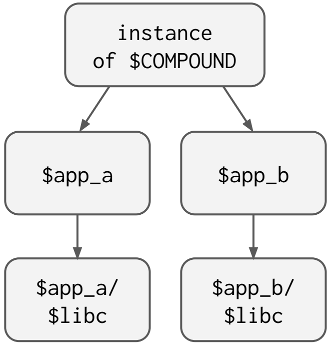
```
(module $APP
  (import “libc” (module $LIBC
    (export “malloc” (func (param i32) (result i32)))
  ))
  (instance $libc (instantiate $LIBC))
  (module $CODE
    (import “libc” (instance $libc
      (export “malloc” (func (param i32) (result i32)))
    ))
    (func (export “run”) (param i32 i32) …)
  )
  (instance $code (instantiate $CODE (import “libc” (instance $libc))))
  (func (export “run”) (param i32 i32)
    (call (func $code “run”) …)
  )
)
```



Current Module Linking proposal summary

- Module Linking allows modules to be *composed*: `(module*) → module`
- Module Linking allows 0..N instances of any module (unlike many module systems)

```
(module $COMPOUND
  (module $LIBC … )
  (module $APP_A
    (import “libc” (module $LIBC …))
    (instance $libc (instantiate $LIBC))
    …
  )
  (module $APP_B
    (import “libc” (module $LIBC …))
    (instance $libc (instantiate $LIBC))
    …
  )
  (instance $app_a (instantiate $APP_A (import “libc” (module $LIBC))))
  (instance $app_b (instantiate $APP_B (import “libc” (module $LIBC))))
)
```



Current Module Linking proposal summary

- First presented to the CG in CG-06-09 (slides)
  - Renamed module-types repo to module-linking, iterated on the proposal there
- Since then:
  - We have a reasonably-complete Wasmtime implementation + tests (kudos Alex Crichton!)
  - This experience led to a number of refinements / tweaks
  - It also surfaced the duplicate import issue presented at CG-03-02 (slides)
- Follow-on discussion in design/#1402 surfaced three concerns:
  - Some embeddings (e.g. JS) do actually have use cases for duplicate imports (overloading)
  - Names shouldn't have meaning in core wasm at all
  - There are different ways to conceptualize "linking", Module Linking is just one approach
- Tentative solution: factor Module Linking out into a new host-agnostic layer

What would Module Linking look like as a new layer?

Core wasm is unchanged, all new things go in a new “adapter module”:

The current (core) proposal:

```
(module
  (import “libc” (module $LIBC …))
  (instance $libc (instantiate $LIBC))
  (module $CODE
    (import “libc” (instance $LIBC …))
    …
    (func (export “run”) …)
  )
  (instance $code (instantiate $CODE
    (import “libc” (instance $libc))
  ))
  (export “run” (func $code “run”))
)
```

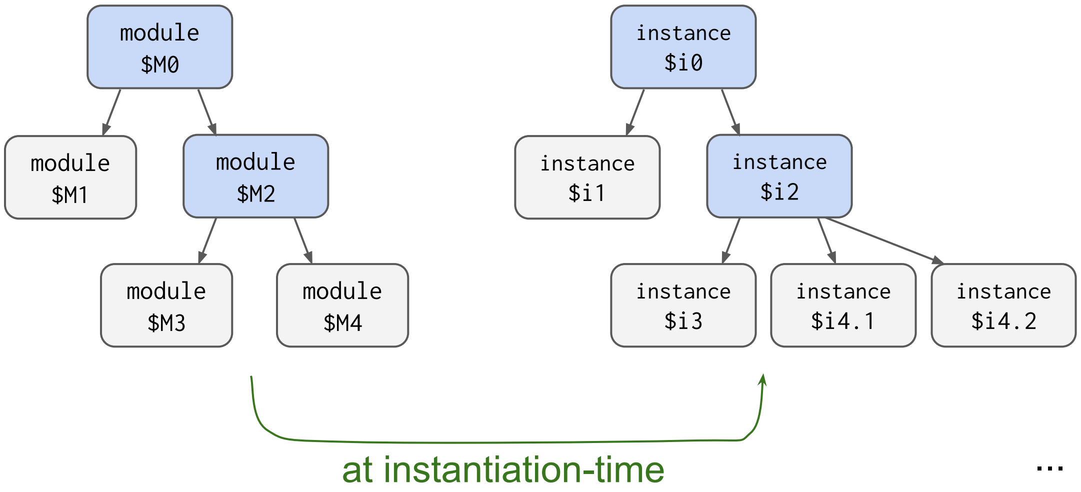
As a layered proposal:

```
(adapter module
  (import “libc” (module $LIBC …))
  (instance $libc (instantiate $LIBC))
  (module $CODE
    (import “libc” “malloc” (func …))
    …
    (func (export “run”) …)
  )
  (instance $code (instantiate $CODE
    (import “libc” “malloc” (func $libc “malloc"))
  ))
  (export “run” (func $code “run”))
)
```

What would Module Linking look like as a new layer?

- What can go inside an adapter module?
  - Types, Imports, Exports, **Core Modules, Adapter Modules, Instances, Aliases**
  - … but not other core wasm sections, thus adapter modules are purely *(typed) wiring*
- Adapter modules/instances form a tree, with core modules/instances only at leaves:

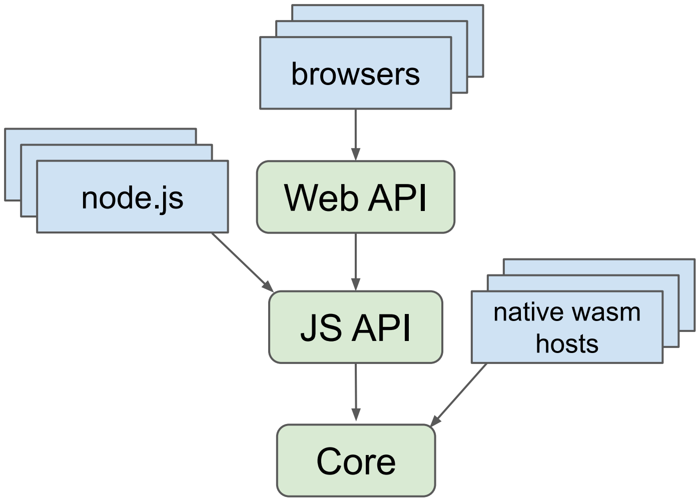
```
(adapter module $M0
  (module $M1 …)
  (instance $i1 (instantiate $M1 …))
  (adapter module $M2
    (module $M3 …)
    (instance $i3 (instantiate $M3 …))
    (module $M4 …)
    (instance $i4.1 (instantiate $M4 …))
    (instance $i4.2 (instantiate $M4 …))
  )
  (instance $i2 (instantiate $M2 …))
)
```



…

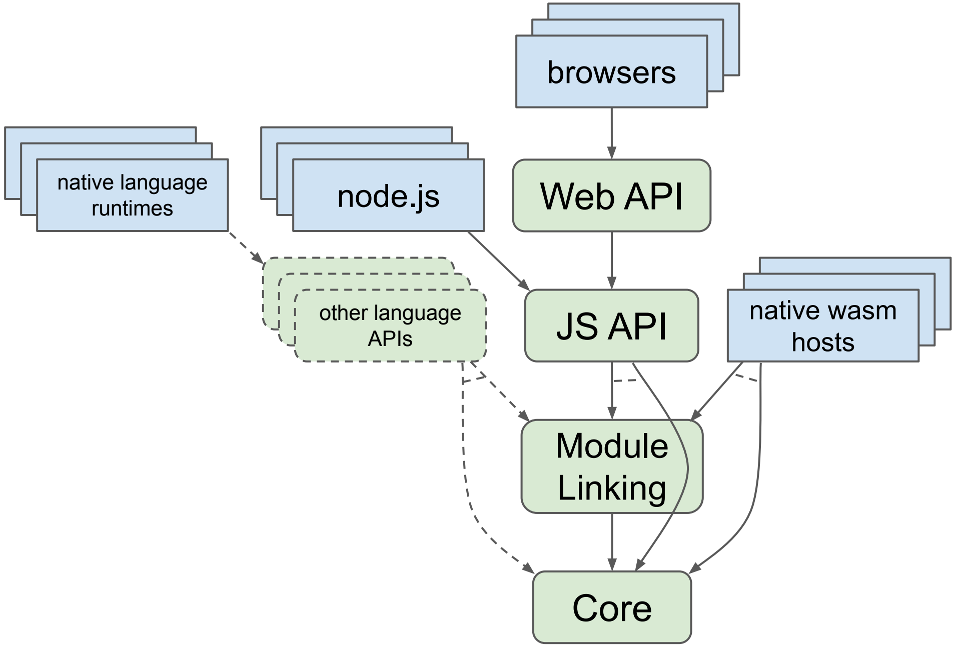
What would Module Linking look like as a new layer?

From a spec document POV (before):



What would Module Linking look like as a new layer?

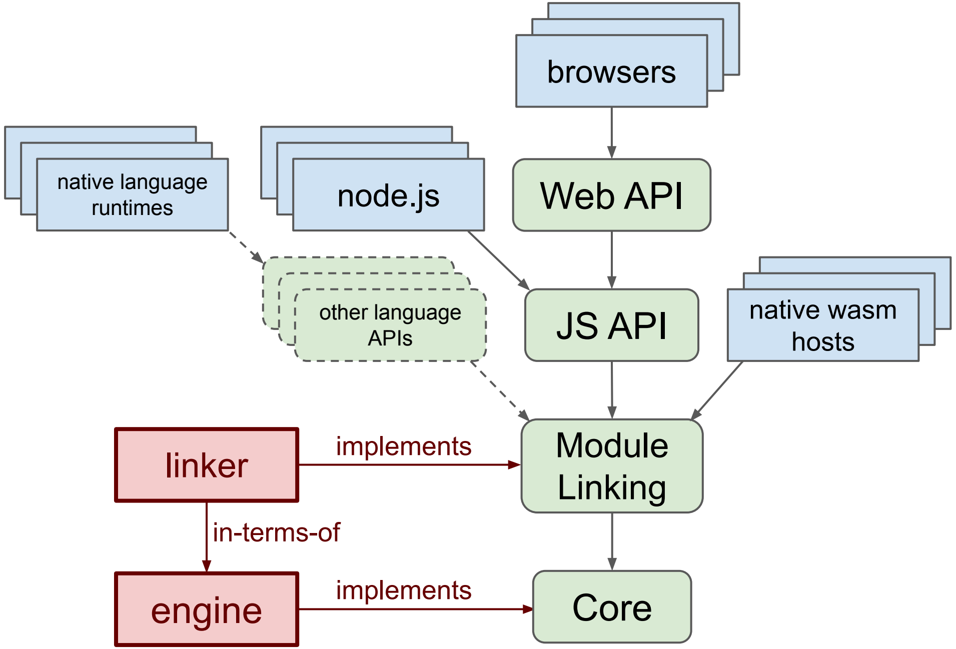
From a spec document POV (after):



...

What would Module Linking look like as a new layer?

From an implementation POV:



What about Interface Types?

- *Already* proposed as a layer above core wasm
- Are there now *two* layers above core wasm?
- Note: Interface Types *already* copies all the linking concepts
  - Used to link core modules’ imports/exports to adapter functions
  - Used to specify how core state is encapsulated
- Thus: Interface Types is more like a *feature proposal*, extending Module Linking
  - The same way that Reference Types extended core wasm
  - If we see hosts that do support ML but not IT, IT could be an optional feature (like GC/SIMD)

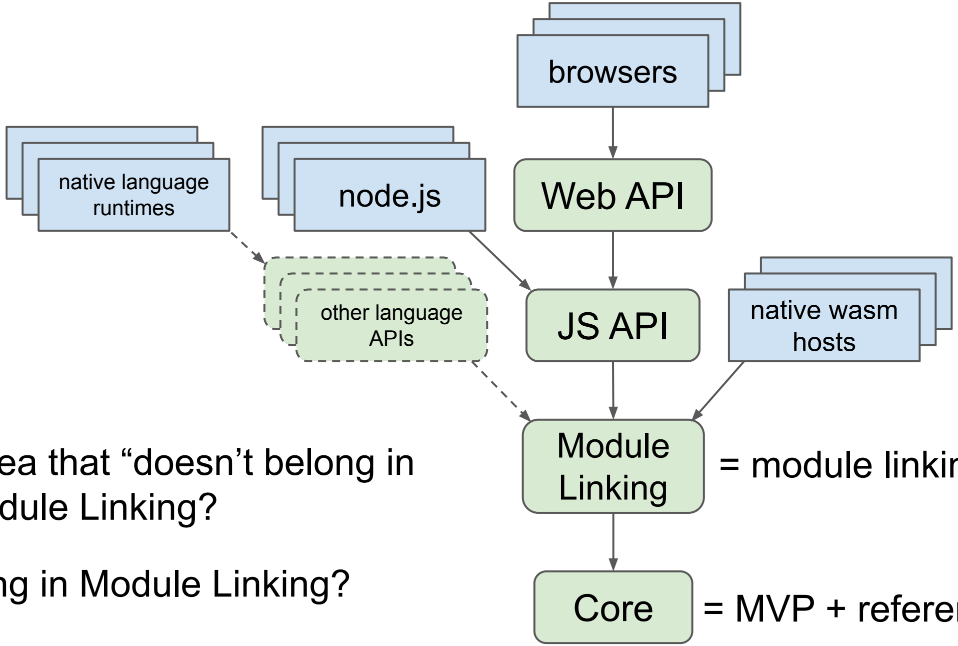
What about Interface Types?

Example of: Core + (Module Linking + Interface Types):

```
(adapter module
  (import “libc” (module $LIBC …))
  (instance $libc (instantiate $LIBC))
  (module
    (import “libc” “malloc” (func (param i32) (result i32)))
    …
    (func (export “run”) (param i32 i32) (result i32 i32) …)
  )
  (instance $core (instantiate $CORE (import “libc” “malloc” (func $libc “malloc"))))
  (adapter_func (export “run”) (param string) (result string)
    … lower param
    call (func $core “run”)
    … lift result
  )
)
```

…

What would ML+IT look like as a new layer?



= module linking + *interface types* + *what else ??*

= MVP + reference-types + bulk-mem + ...

Does *every* new idea that “doesn’t belong in core” belong in Module Linking?

What *doesn’t* belong in Module Linking?

This is going to go poorly if we don’t carefully define a *scope* for Module Linking.

Spectrum of linking dynamism

Module Linking	`(component $C …)`
	+
	`instantiate $C :`
	`[ im* ] → [(handle $C)]`
	+
	`call_export $C $ex :`
	`[ (handle $C) args* ] →`
	`[ results* ]`
		`(module $M …)`
		+
		`instantiate $M :`
		`[ im* ] →`
		`[(ref (instanceof $M))]`
		+
		`call_export $M $ex :`
		`[ (ref (instanceof $M))`
		`args* ] → [ results* ]`
			`(module $M …)`
			+
			`ref.module $M`
			+
			`instantiate :`
			`[(ref (module $MT)) im*] →`
			`[(ref (instance IT))]`
			+
			`instance.get_export $IT $ex :`
			`[(ref (instance $IT))] →`
			`[(ref (func FT))]`
				…
				+
				`compile $MT :`
				`[i32 i32] →`
				`[(ref module $MT)]`
					Full JIT support

existing concepts	existing concepts	existing concepts	existing concepts	existing concepts	~~existing concepts~~
No JIT	No JIT	No JIT	No JIT	~~No JIT~~	~~No JIT~~
static linkage	static linkage	static linkage	~~static linkage~~	~~static linkage~~	~~static linkage~~
No GC	No GC	~~No GC~~	~~No GC~~	~~No GC~~	~~No GC~~
static lifecycle	~~static lifecycle~~	~~static lifecycle~~	~~static lifecycle~~	~~static lifecycle~~	~~static lifecycle~~

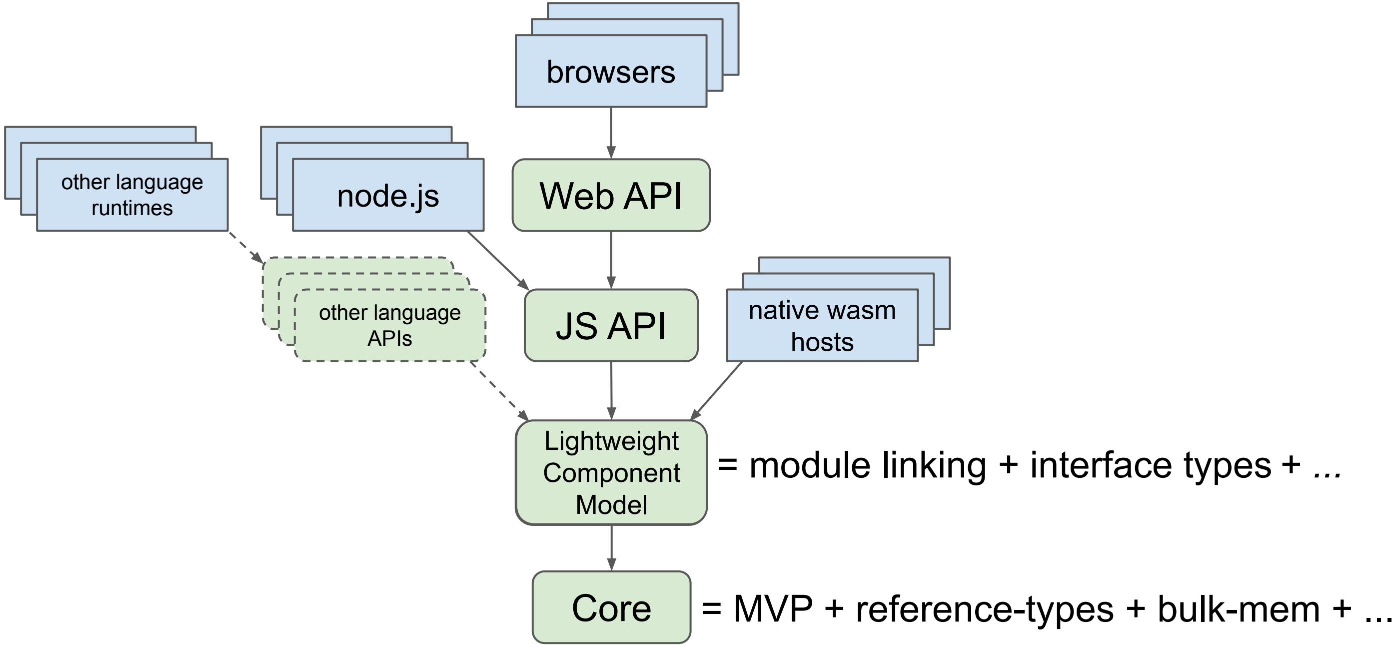
- Not an either/or situation: wasm could offer all of these (in different {GC,JIT} profiles)
- But we can't simply provide the most powerful one: we lose the useful invariants
- Which of these belong in Module Linking? All? Some?

…

Outline

- Background context
    - Why we want to factor Module Linking out of core wasm into a layered spec
    - What a layered spec could look like
    - Why we need think carefully about the scope of this new layered spec
- Proposed scope
- Proposed use cases and requirements
- Proposed next steps
- Discussion + Polls

What is the scope of Module Linking?



What’s a *component*?

- “Component” usually makes people think of COM / CORBA / EJB
  - … and react in horror
- But the term is *much* older:
  - 1968 Presentation: Mass Produced Software Components, Douglas McIlroy
  - “Software components” as the software analogue of existing “hardware components”
    - Fun fact: the notch in the wasm logo is also meant to evoke an IC
- Many incarnations of “components” over time in academia/industry:
  - The COM/CORBA/EJB trio occupy a particularly dynamic point on the spectrum
    - Always virtual dispatch at component boundaries
    - Components linked through global shared mutable registry of components and instances
  - More-recent examples that are more static (like Module Linking):
    - Knit: built for OSKit, third try after ELF+ld (too limited) and COM (too dynamic)
    - Fuchsia Components: dynamic “typing”, but static component manifest

What's a *component*?

- Many definitions over time
  - See, e.g., the 14 definitions in Chapter 11 of Szyperski 2002
- Common themes:
  - Separate compilation and deployment
  - Fully explicit dependencies
  - Black-box (often cross-language) reuse
  - External composition by independent parties
- Why isn't a wasm module already a component?
  - Separate compilation and deployment: ✓ (multiple .wasms, same program)
  - Fully explicit dependencies: ✓ (almost uniquely so!)
  - Black-box reuse: not if memory is shared / not cross-language
  - External composition by independent parties: not from within wasm
- But an adapter module *can* be a component
  - Module Linking for external composition + Interface Types for black-box reuse ✓
- Side note: adapter modules allow *both* Interface Types and core types
  - So "component" would refer to a restricted *subset* of adapter modules
  - ... and adapter modules could be used for non-component purposes

What’s a component *model*?

- It’s a *specification* that toolchains can target and runtimes can implement
  - Thus, (Module Linking + Interface Types) could be (the start of) a component model
- From a low-level POV, a component model is basically an ABI (like ELF)
  - … just higher-level and with greater scope due to cross-language-composition use cases
  - E.g., the toolchain could emit an adapter module binary for `-target wasm32-wasi-component`
    - No XML/JSON manifests, zips or directory structures; **just a single `.wasm`**
- Cross-language/toolchain composition raises tricky questions:
  - Do all components share a single linear memory or other forms of state?
    - If no: how do they pass complex values between components?
  - Is there a global namespace of components and/or component instances?
    - If no: then how are components linked so that they can communicate?
  - Are all modules instantiated using the same global import map?
    - If no: how are the imports determined?
  - Are all modules instantiated exactly once?
    - If no: how and when are instances created and what is their lifecycle?
  - Is there a global event loop shared by all components ?
    - If no: how does a component perform an asynchronous call to another component’s export?

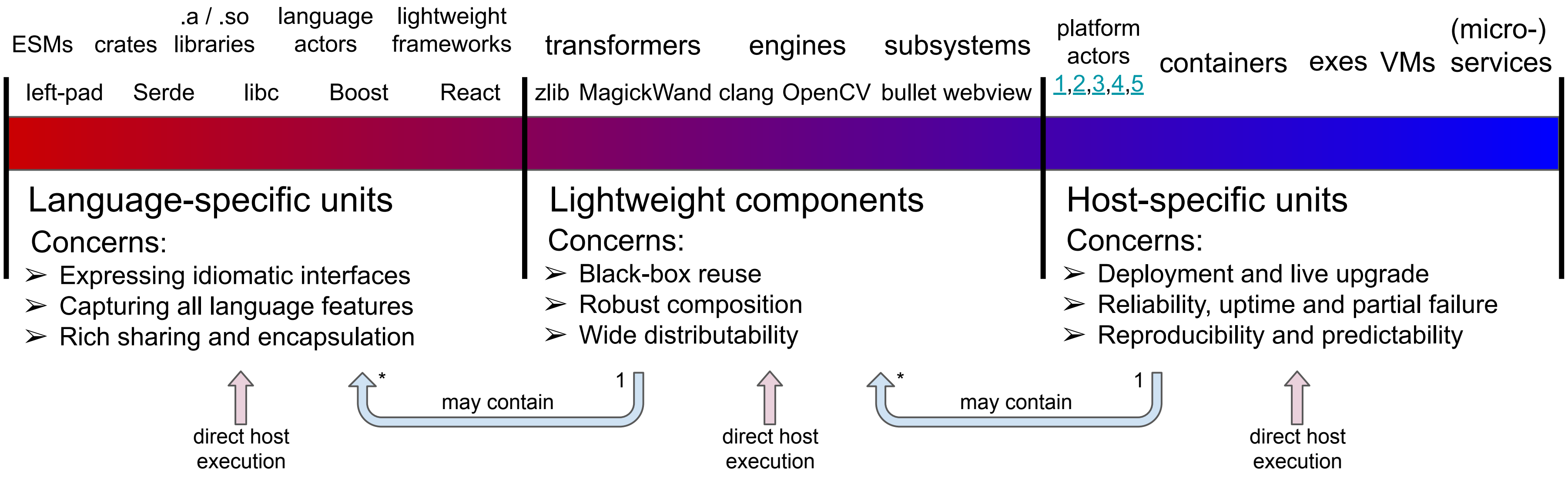
What’s a component *model*?

- We can’t sidestep these questions by exposing more low-level primitives
    - That just kicks the can down the road to some *other* spec
    - A component model needs enough meat on the bones to address whole composition use cases
- This forces a component model to have some “opinions” on the answers
    - On performance vs. robust composition
        - E.g., shared-nothing vs. shared-everything
    - On dynamic expressivity vs. structural invariants
        - E.g., the “spectrum of linking dynamism” earlier
    - Ideally, the “opinions” are strictly derived from a target set of *use cases* and *requirements*
    - But the answers won’t be “universal” in the way we want from core wasm proposals
- *Layering* is what makes this ok
    - Alternative layered specs can capture distinct sets of use cases
    - E.g.: network protocol layering (IP ← {TCP, UDP, …})

What’s a *lightweight* component model?

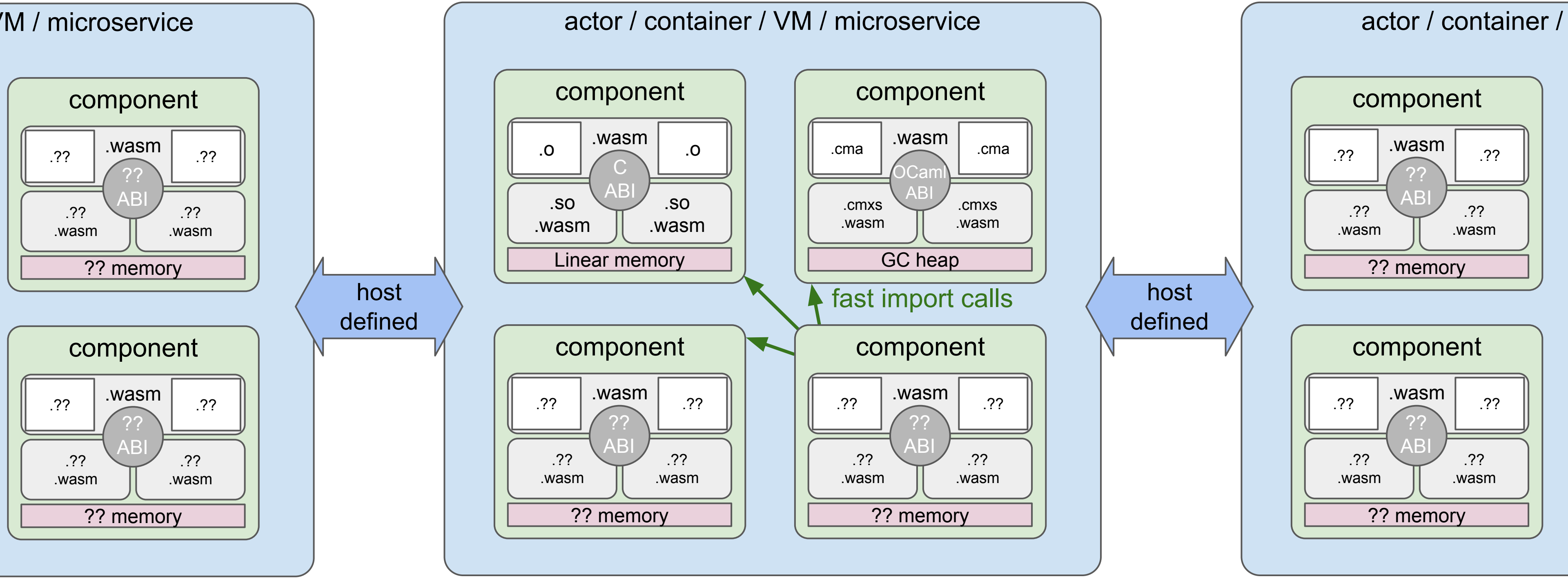
- COM* / CORBA / EJB are pretty *heavyweight*
- They define a *lot* more than a “composable, reusable, distributable unit of code”:
  - Compound document elements, GUI “controls” (OLE, ActiveX)
  - Implicit distributed programming with transactions etc (DCOM, COM+, CORBA)
  - General services (discovery, events, persistence, pooling, caching, …)
- A “lightweight” component model would:
  - Make distributed computing wholly out of scope (problems to be solved at a higher layer)
  - Not define every component to have its own thread of control, inbox, etc (like core wasm today)
  - Not bake any “services” into the component model (this is the domain of WASI)
- But can a lightweight component model actually be valuable without these?
  - Pure COM (without OLE, ActiveX, DCOM, COM+) is actually pretty lightweight
  - While use of the heavier-weight features has decreased, pure COM is going strong
  - So, “yes”

Spectrum of granularity and concerns



- Lightweight components don’t simply replace an existing unit of code
- In particular, not a drop-in replacement for modules / packages / crates / libraries
- But they *can* relieve some of the tension between wanting the isolation and language-independence of ${host unit} yet the low overhead of ${language unit}

What’s a *lightweight* component model?



Lightweight enough to do this ⤴

...

Outline

- Background context
  - Why we want to factor Module Linking out of core wasm into a layered spec
  - What a layered spec could look like
  - Why we need think carefully about the scope of this new layered spec
- Proposed scope
  - Lightweight component model
- Proposed use cases and requirements
- Proposed next steps
- Discussion + Polls

Background

- An accumulation of complementary use cases over the years:
  - Very early: wasm modules as whole apps vs. apps with many small wasm modules
    - Leading to `Wasm.instantiateModule()` → `WA.Module` / `WA.Instance` / `WA.Memory` split
  - Making wasm look and feel more like ES Modules
    - Lin Clark's post-MVP roadmap post (small modules interop)
  - Supply-chain attack mitigation through capabilities + fine-grained isolation
    - Lin Clark's Bytecode Alliance post
  - Ephemeral serverless wasm instances with low-latency instantiation
    - Fastly and Shopify posts
- Broken into 3 categories:
  - Host embedding use cases
  - Composition use cases
  - Static analyzability use cases

Host-embedding use cases

- A developer imports a component from their native language and calls its exports, passing high-level, language-native values instead of `i32`s with manual linear memory manipulation.
- A language runtime imports a component using its native module system, making direct calls to the component's exports as if they were native module exports. (In JS, this would be via ESM-integration.)
- A browser instantiates a component via `<script type='module'>`, supplying unwrapped Web APIs for component imports (via some TBD ESM loader extension).
- A generic wasm CLI calls the exports of components directly by parsing argv according to the high-level typed signature of the invoked component export. (E.g., WASI Typed Main.)
- A serverless runtime executes many components concurrently, maintaining isolated instance state while sharing compiled machine code between instances.
- An embedded device with limited resources runs untrusted applications or sandboxed subsystems as wasm components.
- A monolithic native system sandboxes an unsafe library by compiling it to a component and then compiling that component to an object file that can be linked natively. (E.g., RLBox.)
- A scriptable native platform exposes its functionality to a wasm component, containing the script code, as an importable API.
- A developer instantiates a component with native host imports in production and with mock or emulated imports in local development and testing.

...

Composition use cases

- A component is faithfully reimplemented in a different language or with a different toolchain, possibly switching between linear- and GC-memory, and client components are unaffected.
- A component attempting to violate component model rules fails validation or dynamically traps within the component's code; other components' internal state is never corrupted.
- A component makes an efficient synchronous call to the exports of another component, avoiding queues or context switches, allowing low-overhead fine-grained program decomposition.
- A component efficiently passes high-level values to another component, without both having to agree on a shared memory representation or management scheme.
- A component implements a resource by handing unforgeable handles out to its clients and receiving safe destructor calls that it can use to free linear memory allocations.
- A client component imports a component dependency and creates a fresh instance of this dependency, private to and encapsulated by the client instance.
- A client component imports a component dependency and arbitrarily virtualizes (attenuates, adapts or synthesizes) the dependency's imports.
- A component declares the component types of its dependencies; component subtyping allows reordering, ignored imports/exports and new params and fields with default values.
- A component efficiently streams data to another component, where the streamed data can be arbitrary high-level values and handles, not just raw bytes.

...

Composition use cases (future features)

- A component lazily creates an instance of another imported component dependency on the first call to its exports.
- A component dynamically instantiates, calls, then destroys another component on which it statically depends, treating it as a subcommand with a bounded lifecycle.
- A component creates a fresh instance of itself every time its exports are called, avoiding any reused state and aligning with the usual assumptions of C programs' `main()`.
- A component bundles its own event loop logic, being able to effectively call async functions in other components for other languages (that may have their own event loop).
- A component implemented in a language without native async support is able to call the exported async function of another component with reasonable blocking and non-blocking options.
- A component forks a call to another component, achieving task parallelism while preserving determinism due to the absence of shared mutable state.
- Two components connected by a stream execute in different threads, achieving pipeline parallelism while preserving determinism due to the absence of shared mutable state.

...

Static analyzability use cases

- For fixed imports, an AOT compiler transforms all named imports and exports to constant indices or more direct forms of reference.
- For fixed imports, an AOT compiler performs cross-module inlining to eliminate the call overhead induced by separate compilation.
- For fixed imports, an AOT compiler performs reliable cross-component fusion of marshalling code to eliminate temporary intermediate copies and allocations.
- For fixed imports, an AOT polyfill build tool allow components to run on hosts implementing only core wasm by automatic translation of components into core wasm + host glue code.
- A tool visualizes the possible side effects of a component (including its transitive dependencies) based on the component’s public interface.
- A tool visualizes how the set of static capabilities imported by a component are transitively delegated to that component’s dependencies without having to analyze any core module code.
- A tool gives clients a chance to evaluate the validity of a dependency update that imports new capabilities based on the dependency’s new and old public interface.

...

Requirements

- I think the requirements can mostly be derived from the use cases
  - As in, violating a requirement probably means breaking a use case
  - But of course there may be unexplored alternatives (hence the “mostly”/”probably”)
- One can easily imagine alternative use cases → alternative requirements, e.g.:
  - Ubiquitous GC
  - Full runtime dynamic linking expressivity
- These could well be alternative layered specs
  - All embedding the same core spec
  - It’s just a matter of seeing what use cases emerge to motivate a new scope
- Layering enables us to not have to address all conceivable use cases
  - Which is what makes adding features to core wasm Hard™

Requirements

- **Shared-nothing**
  - Components must fully encapsulate all mutable core wasm state (linear/GC memory, globals, tables).
- **Virtualizability**
  - Any interface importable by a component must be implementable by some other component.
- **Parameterization, not namespaces**
  - All sharing must be via explicit parameters chosen by the client, not a name-based runtime global registry.
- **Static component linkage**
  - Once root imports are fixed, all imported code pointers are fixed; only the instance pointers can vary.
- **Explicit acyclic ownership**
  - Destruction of resources and component instances must not require garbage- or cycle-collection.
- **No mandatory profiles** (GC, JIT, Threads, SIMD, …)
  - Individual component bodies may depend on profiles, but the component model itself must not.

…

Spectrum of linking dynamism

Module Linking

existing concepts

No JIT

static linkage

No GC

static lifecycle

```
(component $C …)
+
instantiate $C :
[ im* ] → [(handle $C)]
+
call_export $C $ex :
[ (handle $C) args* ] →
[ results* ]
```

existing concepts

No JIT

static linkage

No GC

~~static lifecycle~~

```
(module $M …)
+
instantiate $M :
[ im* ] →
[(ref (instanceof $M))]
+
call_export $M $ex :
[ (ref (instanceof $M))
args* ] → [ results* ]
```

existing concepts

No JIT

static linkage

~~No GC~~

~~static lifecycle~~

```
(module $M …)
+
ref.module $M
+
instantiate :
[(ref (module $MT)) im*] →
[(ref (instance IT))]
+
instance.get_export $IT $ex :
[(ref (instance $IT))] →
[(ref (func FT))]
```

existing concepts

No JIT

~~static linkage~~

~~No GC~~

~~static lifecycle~~

```
…
+
compile $MT :
[i32 i32] →
[(ref module $MT)]
```

existing concepts

~~No JIT~~

~~static linkage~~

~~No GC~~

~~static lifecycle~~

Full JIT support

~~existing concepts~~

~~No JIT~~

~~static linkage~~

~~No GC~~

~~static lifecycle~~

In scope for component model

May be added (in some form) to core wasm in the future; may be used by individual component *bodies*; but not baked into the *component model*

…

Outline

- Background context
  - Why we want to factor Module Linking out of core wasm into a layered spec
  - What a layered spec could look like
  - Why we need think carefully about the scope of this new layered spec
- Proposed scope
  - Lightweight component model
- Proposed use cases and requirements
  - What these entail for linking
- Proposed next steps
- Discussion + Polls

Proposed next steps

1. Create a new component-model repo
   a. Containing docs for high-level goals, use case, requirements, FAQ, etc (like the design repo)
   b. Later, merge in the formal spec and spec-interpreter (like the spec repo)
2. Rebase the module-linking repo onto the component-model repo
   a. Use module-linking to initialize the spec+interpreter and continue linking-specific discussions
   b. No core changes are proposed; the “remove duplicate imports?” issue is resolved “no”
3. Rebase the interface-types repo onto the module-linking repo
   a. It’s now just a feature proposal, but for the component-model spec
   b. The proposal adds new types and a new definition kind (adapter functions)
4. Split out new adapter-functions repo as a separate feature repo
   a. Adapter functions are the Hard part of Interface Types and there’s more churn coming
   b. … but ultimately they are just an *optimization* over using a fixed, canonical ABI
5. Add “canonical adapter functions” to the interface-types proposal

Canonical adapter functions

```
(adapter module
  (import “log” (func $log (param string)))
  (adapter_func $adapt_log (param i32 i32)
    (call $log (list.lift …))
  )
  (module $CORE
    (import “” “log” (func (param i32 i32))
    …
    (func (export “run”) (param i32 i32) …)
  )
  (instance $core (instantiate $CORE
    (import “” “log” (func $adapt_log))))
  (adapter_func $adapt_run (param string)
    (call (func $core “run”) (list.lower …))
  )
  (export “run” (func $adapt_run))
)
```

```
(adapter_func $adapt_log (param i32 i32)
  canonical import $log
)
```

In the binary format, there is `0` LEB immediate so we can later add a `memory=gc` option

```
(adapter_func $adapt_run (param string)
  canonical export (func $core “run”)
)
```

Proposed next steps

1. Create a new component-model repo
   a. Containing docs for high-level goals, use case, requirements, FAQ, etc (like the design repo)
   b. Later, merge in the formal spec and spec-interpreter (like the spec repo)
2. Rebase the module-linking repo onto the component-model repo
   a. Use module-linking to initialize the spec+interpreter and continue linking-specific discussions
   b. No core changes are proposed; the “remove duplicate imports?” issue is resolved “no”
3. Rebase the interface-types repo onto the module-linking repo
   a. It’s now just a feature proposal, but for the component-model spec
   b. The proposal adds new types and a new definition kind (adapter functions)
4. Split out new adapter-functions repo as a separate feature repo
   a. Adapter functions are the Hard part of Interface Types and there’s more churn coming
   b. … but ultimately they are just an *optimization* over using a fixed, canonical ABI
5. Add “canonical adapter functions” to the interface-types proposal
   a. Sidestep hard adapter function design questions by fixing a canonical ABI
   b. **… allowing module-linking + interface-types to be a component model MVP**

…

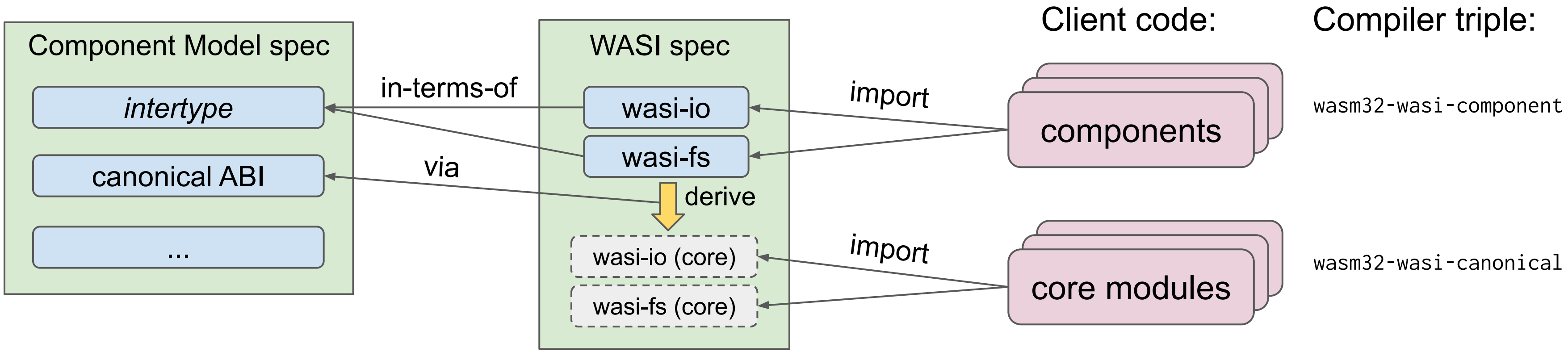
Developer Preview

- With a component model MVP, we could plan a “Developer Preview” release
  - Like the Browser Preview leading up to wasm MVP release
- Goal: provide a solid foundation for WASI
  - This requires adding handles (as presented) and buffers (in progress) to interface-types
- Goal: enable JS developers to try out components
  - This means supporting components in one or both of: JS API, ESM-integration
  - I assume browsers will want to wait to see developer usage before implementing natively
    - … like they did with the original ES Modules proposal
    - This means building an AOT polyfill in terms of core wasm + JS API, used by bundlers
    - ESM-integration is much more conducive to AOT polyfilling, so let’s start with that
- Goal: enable non-browser developers to try out components
  - Wasmtime module-linking implementation already underway (happy to collaborate with others)
- Goal: enable creating components by specifying witx interfaces
  - `witx-bindgen` tool: generate { host, guest } glue code from a `.witx`
  - Rust support underway, JS and C/C++ (wasi-sdk) planned (happy to collaborate with others)

…

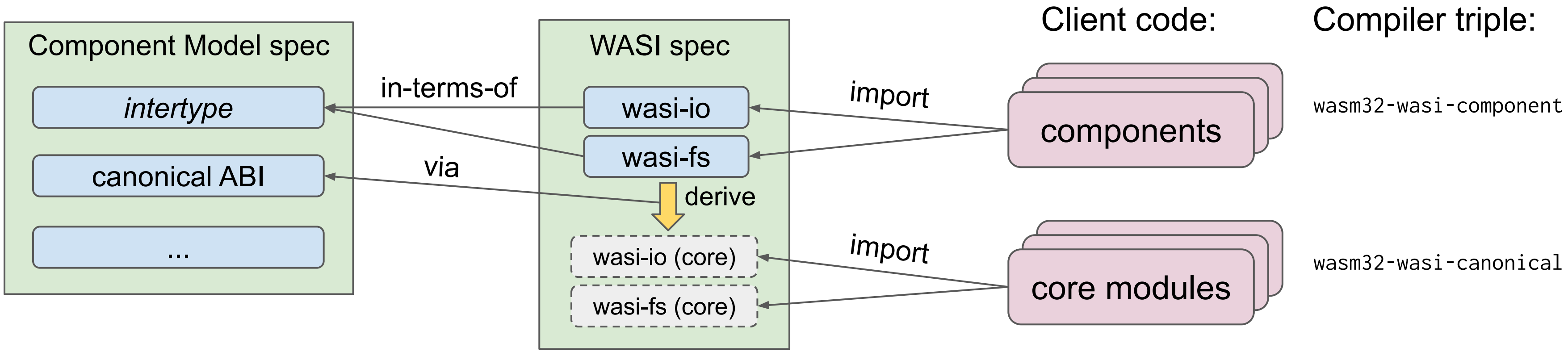
What about WASI?

- WASI wants to define its interfaces in terms of interface types
  - Which means that the WASI spec would depend on the component model spec
- But use of WASI shouldn't be restricted to components
  - Tools are producing/consuming core modules today and should be able to continue to do so
- How?



What about WASI?

- When using WASI from a core module, WASI looks basically like it does today
  - The Component Model factors out what WASI would otherwise duplicate in its IDL spec (witx)
- When using WASI from a component, you get some benefits:
  - Components encapsulate their memory and handle-tables
  - Components can implement (virtualize) WASI without magic (e.g., regarding “caller’s memory”)
  - Components can (eventually) avoid (de)serialization with adapter functions



...

Outline

- Background context
  - Why we want to factor Module Linking out of core wasm into a layered spec
  - What a layered spec could look like
  - Why we need think carefully about the scope of this new layered spec
- Proposed scope
  - Lightweight component model
- Proposed use cases and requirements
  - What these entail for linking
- Proposed next steps
  - Proposed next steps for the proposal repos
  - Developer Preview sketch
  - Implications for WASI
- Discussion + Polls

Discussion + Polls

1. Does the general proposed direction sound good?

2. Should we proceed with the proposed next steps?