Module Harmony:
interaction semantics

Scope of the discussion

- ✓ Cross-proposal discussions
- ✓ Discussions about the general modules space
- ✓ Clarifying questions about cross-cutting concerns or individual proposals
- ✗ In-depth discussion regarding specific proposals are better stuired for those proposal's presentations/github issues.

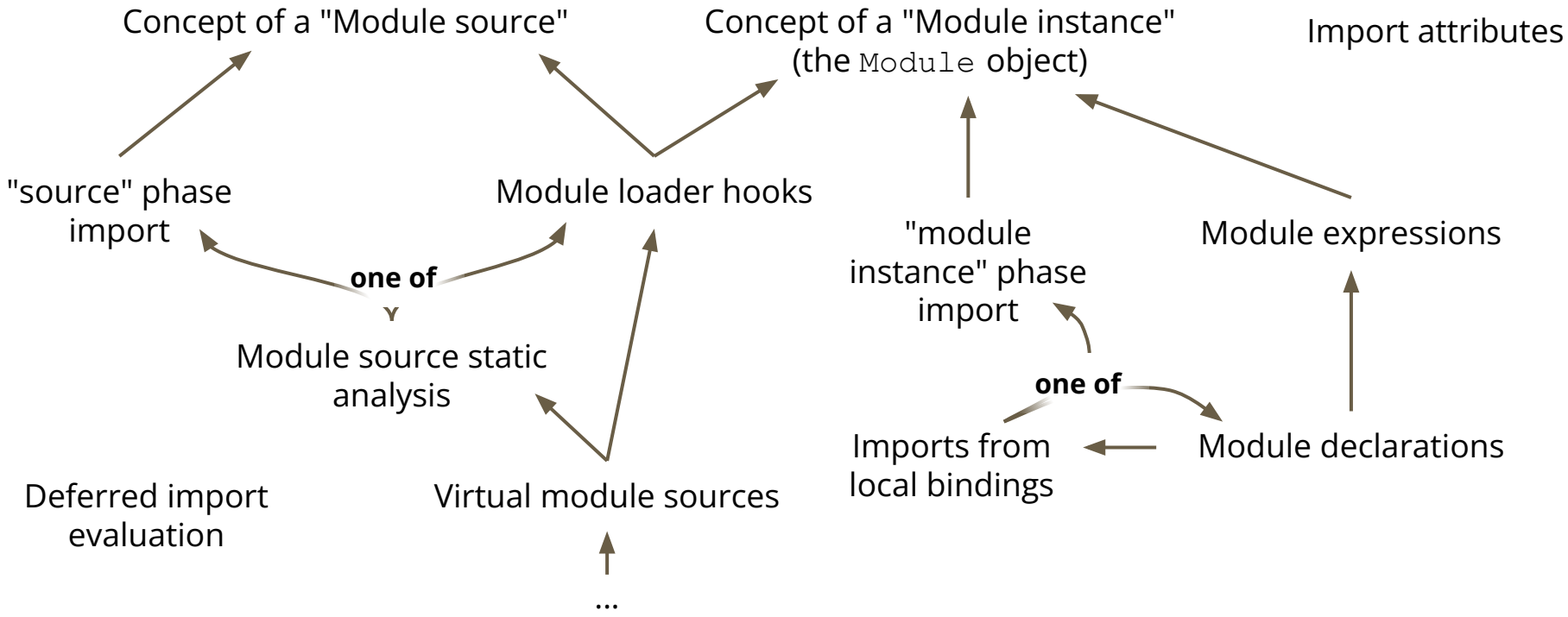
Proposals covered by this presentation

Proposals covered in this presentation

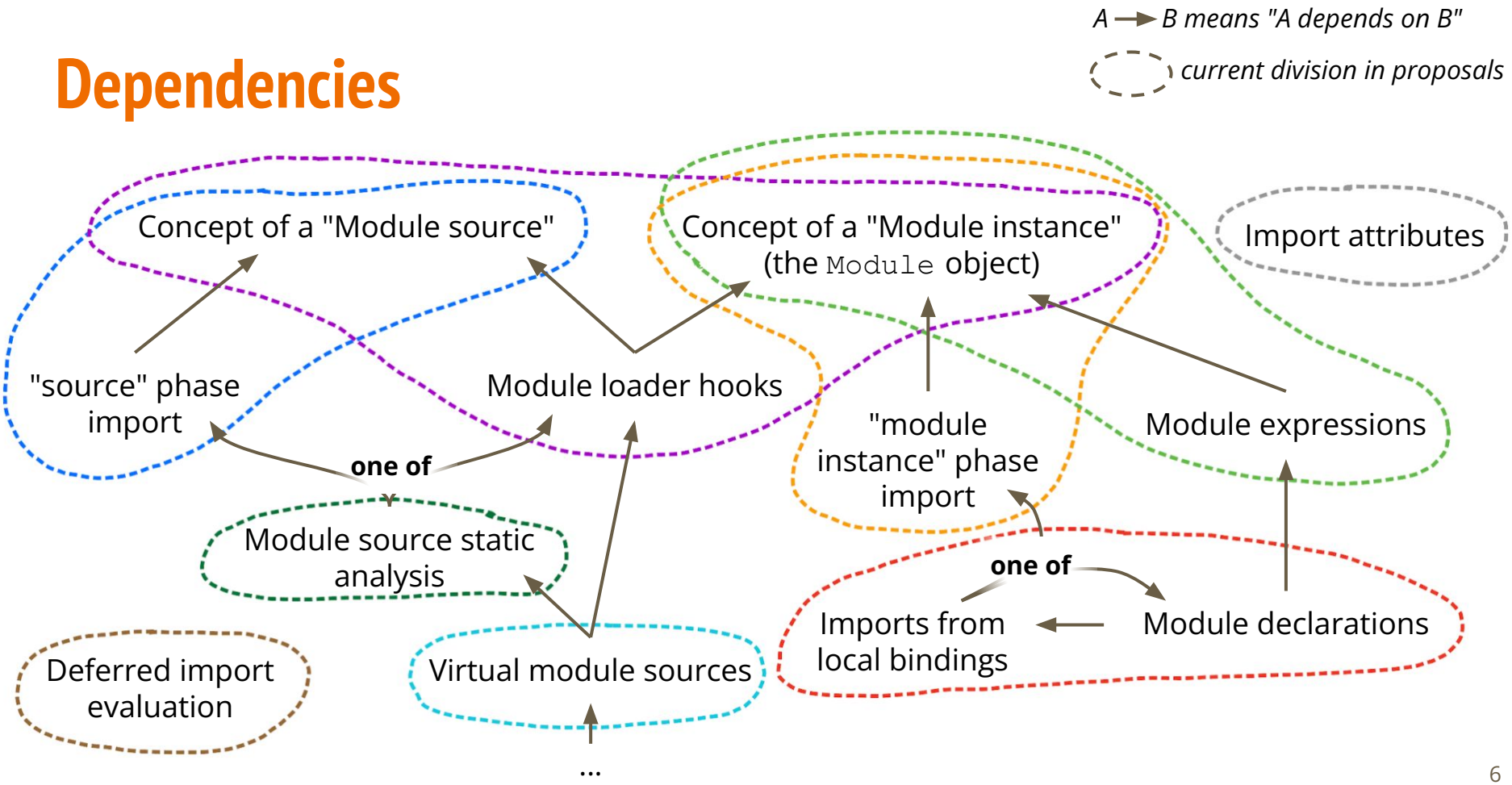
- Import attributes (formerly *import assertions*)
- Module expressions
- Module declarations & local imports
- Module source imports (formerly *import reflection*)
- Module instance imports (formerly *import reflection*)
- Deferred import evaluation
- Module loader hooks (aka *compartments layer 0*)
- Module source static analysis (aka *compartments layer 1*)
- Virtual module sources (aka *compartments layer 2*)

A → B means "A depends on B"

Dependencies



Dependencies



Interaction semantics

Module source/instance imports and deferred import evaluation

Module source imports

Allow importing state-less and context-less representation of module sources, such as **`WebAssembly.Module`** objects.

```
import source wasm from "./mod.wasm";
wsam instanceof WebAssembly.Module;
```

Module instance imports

Allow importing a representation of an unlinked module together with its evaluation/linkage context.

Deferred import evaluation

Allows loading and linking a module, while deferring it's evaluation until its contents are actually used.

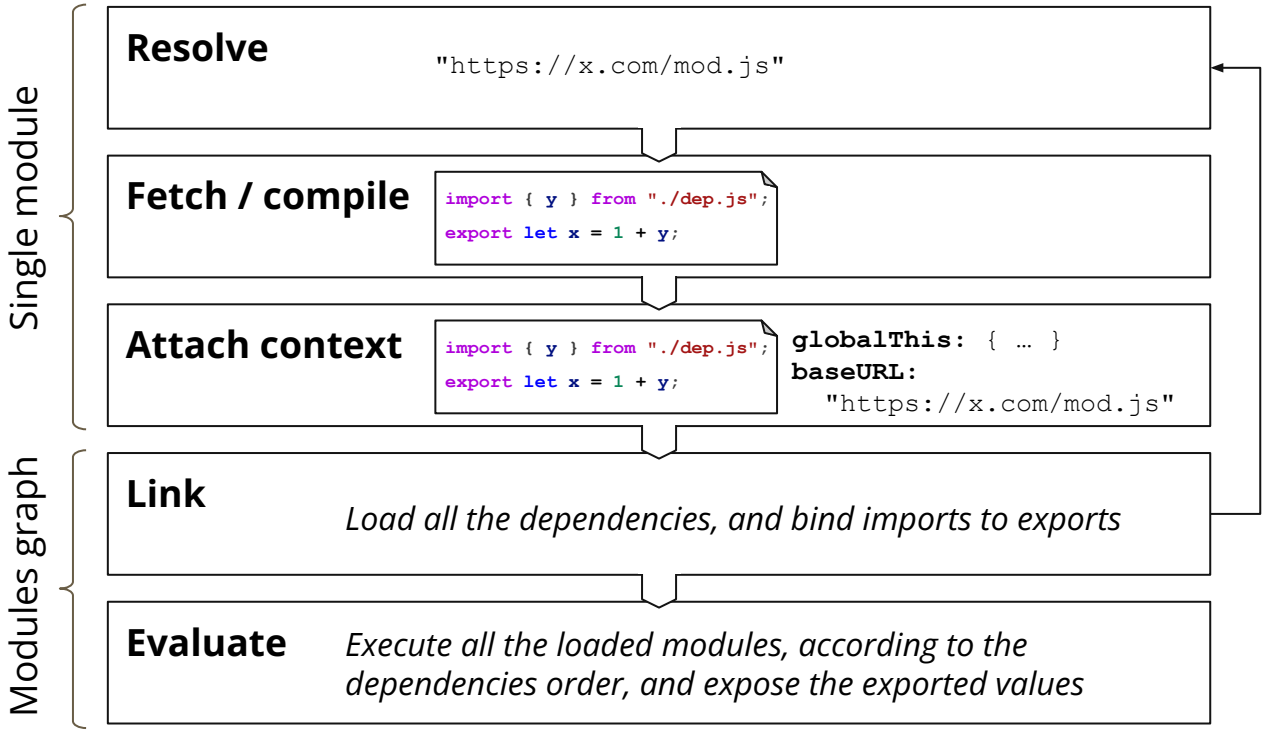
```
import defer * as ns from "./mod.js";

function later() {
  ns.anExport; // Evaluate mod.js
}
```

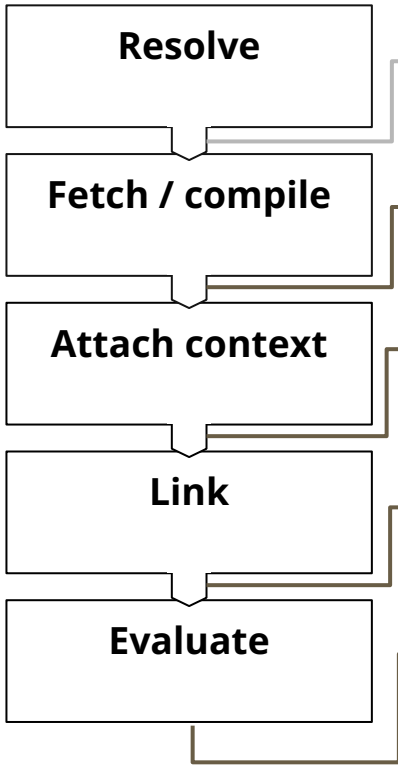
Import phase modifiers

The module loading process can be divided in multiple "phases", split between the host and ECMA-262.

The **import reflection** and **deferred import evaluation** proposals aim at exposing modules at earlier phases.



Import phase modifiers



Asset references

```
import asset s from "./m0.js";
```

"Source" phase imports

```
import source s from "./m1.js";
```

"Module instance" phase imports

```
import module m from "./m2.js";
```

Deferred import evaluation

```
import defer * as d from "./m3.js";
```

```
import * as ns from "./m4.js";
```

Module expressions, "module instance" phase import & module loader hooks

Module expressions

- Allow declaring modules inlined inside other modules

```
let mod = module {
  import { count } from "./db.js";

  export let x = 2 * count();
};
```

Module loader hooks

- Allow creating modules with a custom import behavior

```
let mod = new Module(source, {
  importHook(specifier) {
    return registry[specifier];
  },
});
```

Module expressions, "module instance" phase import & module loader hooks

All these proposals allow to first get a handle to a loaded module, and later load its dependencies and evaluate it.

	`// "get a module"`	`// and later import it`
Module expressions	`let mod1 = module {};`	`await import(mod1);`
Module declarations	`module mod2 {}`	`await import(mod2);`
Module instance imports	`import module mod3 from "./dep";`	`await import(mod3);`
module loader hooks	`let mod4 = new Module(source, hooks);`	`await import(mod4);`

Module expressions, "module instance" phase import & module loader hooks

They will all result in instances of the same **`Module`** class.

```
module {} instanceof Module;

module mod2 {};
mod2 instanceof Module;

import module mod3 from "./dep";
mod3 instanceof Module;

new Module(...) instanceof Module;
```

```
new Module(source, {
  importHook(specifier) {
    if (specifier === "a") {
      return import("a", { phase: "module" });
    } else if (specifier === "b") {
      return module { ... };
    } else if (specifier === "c") {
      return new Module(source, hooks);
    }
  },
});
```

Module expressions, "module instance" phase import & module loader hooks

Unlocked use case: *partial* virtualization of module loading.

- module loader hooks allows defining a custom module loader, that loads modules which *can* transitively have their dependencies virtualized.
- "module instance" phase import (and module expressions) allow delegating back to the host loader for a portion of the modules graph.

```
async importHook(specifier) {
  if (transitiveVirtualize) {
    return new Module(await loadSource(specifier), hooks);
  }
  return import(specifier, { phase: "module" });
},
```

Virtualize the dependency

Continue with host loader ⚡

"Source" phase import & module loader hooks

Source imports allow virtualizing the imports of "real" modules (as opposed to string-based eval):

```
const mainURL = new URL("./main.js", import.meta.url);
const mainSource = await import("./main.js", { phase: "source" });

new Module(mainSource, {
  async importHook(specifier) {
    const depURL = new URL(specifier, mainURL);
    const depSource = await import(depURL, { phase: "source" });
    return new Module(depSource, hooks);
  }
});
```

Module declarations & "module instance" phase import

Module declarations

- Allow declaring inline modules always bound to statically analyzable bindings

```
module Utils {
  export const id = x => x;
}
```

- Can be statically imported from

```
import { id } from Utils;
```

- Can be exported and re-imported

```
// a.js
export module Utils { /* ... */ }

// b.js
import { Utils } from "./a.js";
import { id } from Utils;
```

"module instance" phase import

- Allows declaring statically analyzable bindings referencing to unlinked external modules

```
import module react from "//cdn.js/react";
```

Module declarations & "module instance" phase import

Unlocked use case: in-band dependencies management.

- Non-npm environments use URLs and import maps to define dependencies
- Import maps can only be defined at the application level, and not at the library level

Deno's solution: **`deps.ts`**, which re-exports all the external dependencies:

```
// https://github.com/eveningkid/denodb/blob/master/deps.ts
export {
  Client as MySQLClient,
  configLogger as configMySQLLogger,
  Connection as MySQLConnection,
} from "https://deno.land/x/mysql@v2.11.0/mod.ts";
export { Client as PostgresClient } from "https://deno.land/x/postgres@v0.14.2/mod.ts";
export { DB as SQLiteClient } from "https://deno.land/x/sqlite@v3.7.0/mod.ts";
```

Module declarations & "module instance" phase import

Unlocked use case: in-band dependencies management.

- Non-npm environments use URLs and import maps to define dependencies
- Import maps can only be defined at the application level, and not at the library level

Deno's solution: `deps.ts`, which re-exports all the external dependencies:

- Shared namespace conflicts
- All dependencies are linked and evaluated, even if not actually used

Module declarations & "module instance" phase import

Unlocked use case: in-band dependencies management.

- Non-npm environments use URLs and import maps to define dependencies
- Import maps can only be defined at the application level, and not at the library level

Deno's solution: **`deps.ts`**, which re-exports all the external dependencies:

```
// deps.ts
export module MySQL from "https://deno.land/x/mysql@v2.11.0/mod.ts";
export module Postgres from "https://deno.land/x/postgres@v0.14.2/mod.ts";
export module SQLite from "https://deno.land/x/sqlite@v3.7.0/mod.ts";

// somehwere else
import { MySQL } from "./deps.ts";
import { Client } from MySQL;
```

Module declarations/expressions & import attributes

Module declarations/expressions

- Allow declaring inline modules that can later be imported.

```
module mod1 { ... }
const mod2 = module { ... };

import { x } from mod1;
const { y } = await import(mod2);
```

Import attributes

- Are parameters to the host loader that can affect how a module is loaded or evaluated.

```
import json from "./mod" with {
  type: "json" };

import css from "./mod" with {
  type: "css" };
```

Module declarations/expressions & import attributes

As a module declaration/expression defines an already loaded module, the host cannot further modify its behavior based on import attributes.

```
module mod1 {}
const mod2 = module {};
import mod1 with { type: "css" }; // Error!
await import(mod2, { with: { type: "css" } }); // Error!
```

On the other hand, import phase modifiers are allowed and make sense:

```
import source s from mod1; // A JS ModuleSource object
import defer * as ns from mod1; // Link and load deps, so not execute
```

Module loader hooks & import attributes

Module loader hooks

- The `Module` constructor accepts an `importHook` that loads an imported module given the specifier.
- `importHook` is called once per specifier.

```
new Module(source, {
  async importHook(specifier) {
    return /* a Module instance */;
  }
});
```

Import attributes

- They can affect how a module is loaded/evaluated.
- The host must define a finite set of supported attributes.
  (via `HostGetSupportedImportAttributes`)
- The order in which attributes are specified must not be significant.

Module loader hooks & import attributes

- The **`Module`** constructor accepts a **`supportedAttributes`** array, which defaults to **`[]`**.
- The **`importHook`** receives an **`attributes`** object with the import attributes specified for the given import.
- **`attributes`** properties are sorted lexically.
- The importHook is called once per (**`specifier`**, **`attributes`**) pair.

```
new Module(source, {
  supportedAttributes: ["type"],

  async importHook(specifier, attributes) {
    // ...
    const { type } = attributes;
    if (resolvedType(specifier) !== type) {
      throw new Error();
    }
    // ...
    return /* a Module instance */;
  },
});
```

Module loader hooks & import attributes

```
new Module(
  `import a from "./dep";
   import b from "./dep" with { type: "json" };
   import c from "./dep" with { type: "css" };
   import d from "./dep" with { type: "json" };`,
  {
    supportedAttributes: ["type"],

    async importHook(specifier, attributes) {
      console.log(specifier, attributes);
      return /* a Module instance */;
    },
  }
)
```

Logs:

- `"./dep", {}`
- `"./dep", { type: "json" }`
- `"./dep", { type: "css" }`

Module loader hooks & import attributes

```
new Module(
  import "./dep" with { webpackLoader: "image" };
,
  {
    supportedAttributes: ["type"],

    async importHook(specifier, attributes) {
      console.log(specifier, attributes);
      return /* a Module instance */;
    },
  }
)
```

Throws:

- **SyntaxError:** `webpackLoader` is not a supported attribute.

Module loader hooks & import phase modifiers

Module loader hooks

- The `Module` constructor accepts an `importHook` that loads an imported module given the specifier.
- `importHook` is called once per specifier.

```
new Module(source, {
  async importHook(specifier) {
    return /* a Module instance */;
  }
});
```

Import phase modifiers

```
import source s from "./mod.js";
import module m from "./mod.js";
import defer * as ns from "./mod.js";
```

- They affect how a module is exposed, but don't affect what the underlying module is.
- They are fully defined within ECMA-262, and cannot change meaning across different hosts.

Module loader hooks & import phase modifiers

- **`importHook`** *doesn't* receive the import phase.
- **`importHook`** is called once per specifier, regardless of how many different phases are used.
- The **`Module`** instance returned by the **`imprtHook`** will be advanced up to the requested phase.

```
import source s from "./mod.js";
```

The import returns the **`.source`** property of the **`Module`** returned by **`importHook`**.

```
import module s from "./mod.js";
```

The import returns the **`Module`** returned by **`importHook`**.

```
import defer * as ns from "./mod.js";
```

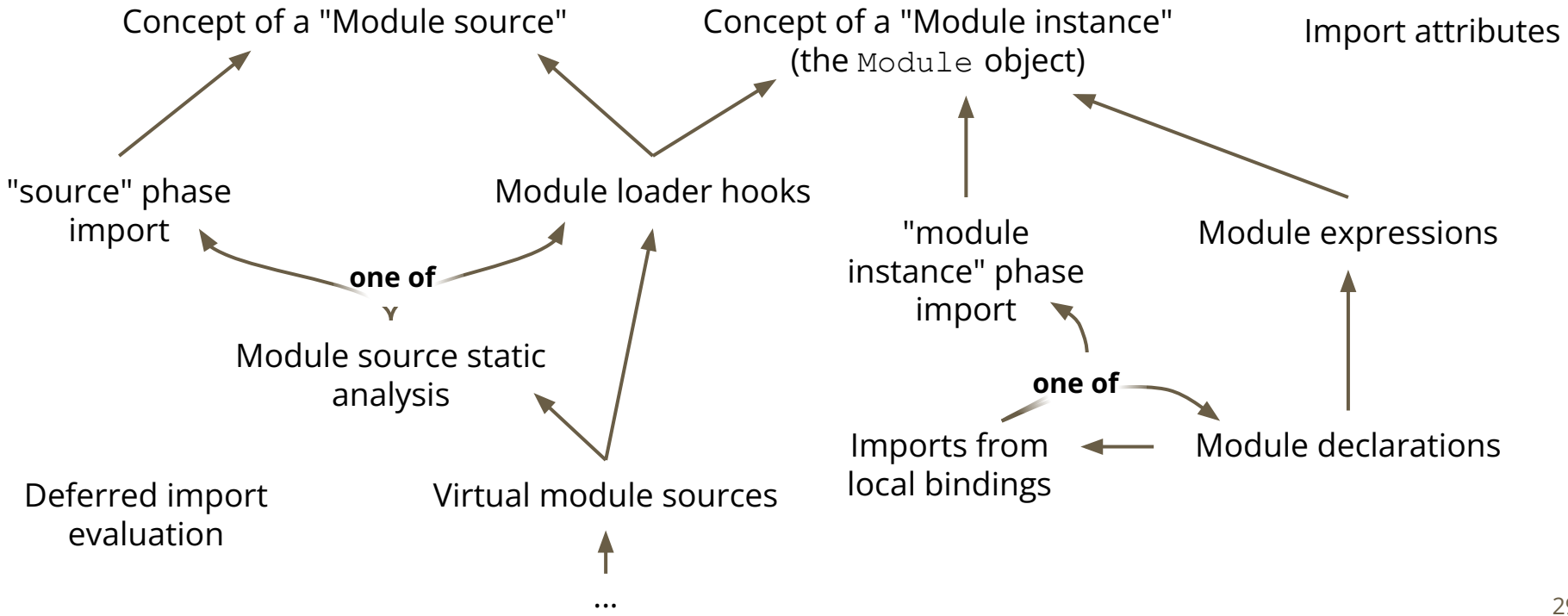
The loader continues loading all the dependencies of the **`Module`** returned by **`importHook`**, and then defers evaluation.

Dependencies - Our goals

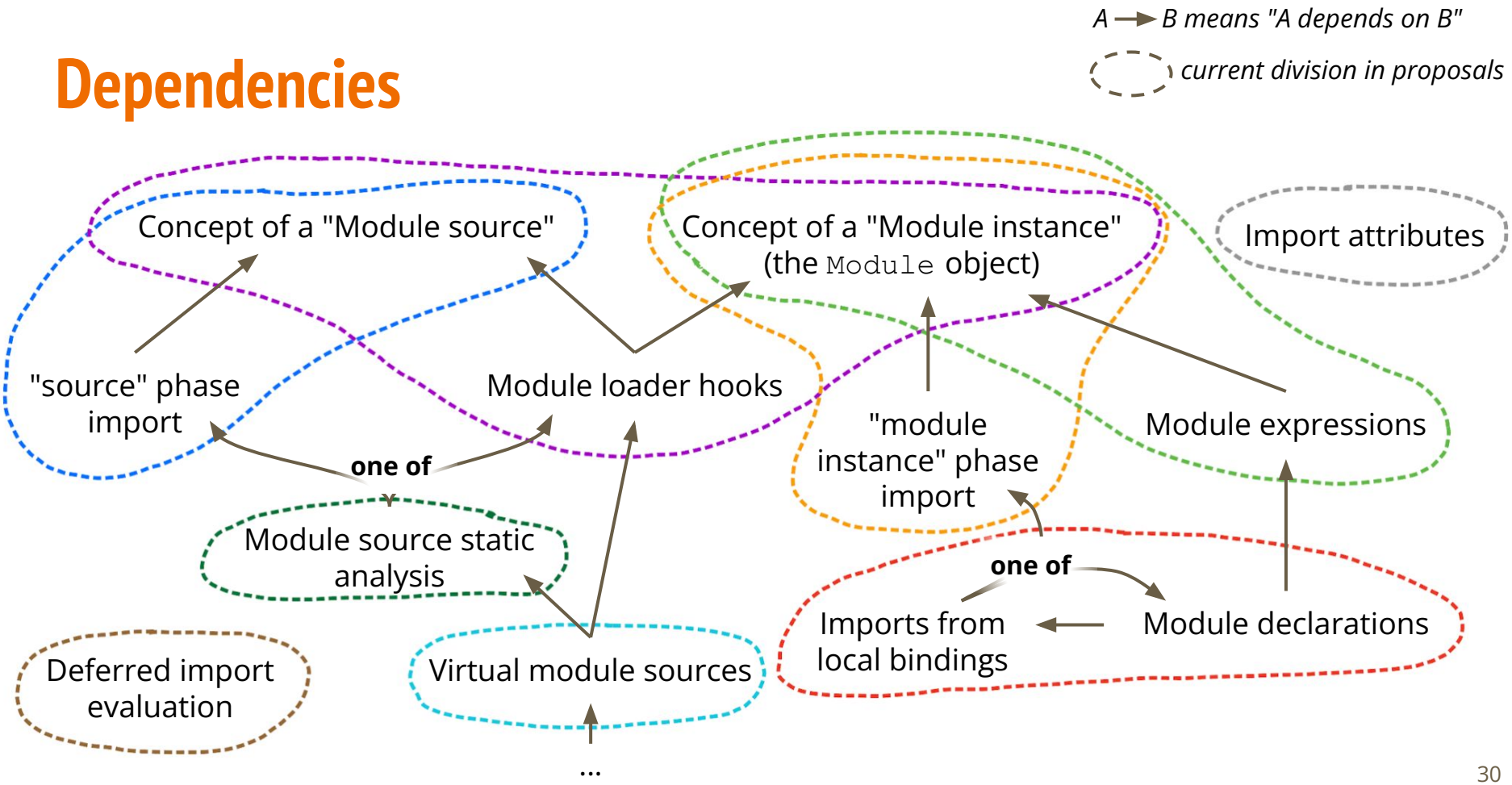
- Proposals should be able to move ahead independently from each other in the standardization process
- Proposals can only move independently if they are self-motivated
- If any proposal "gets stuck", the remaining pieces must still make sense on their own
- Proposals should have minimal dependencies on each other, to allow being developed in parallel

A → B means "A depends on B"

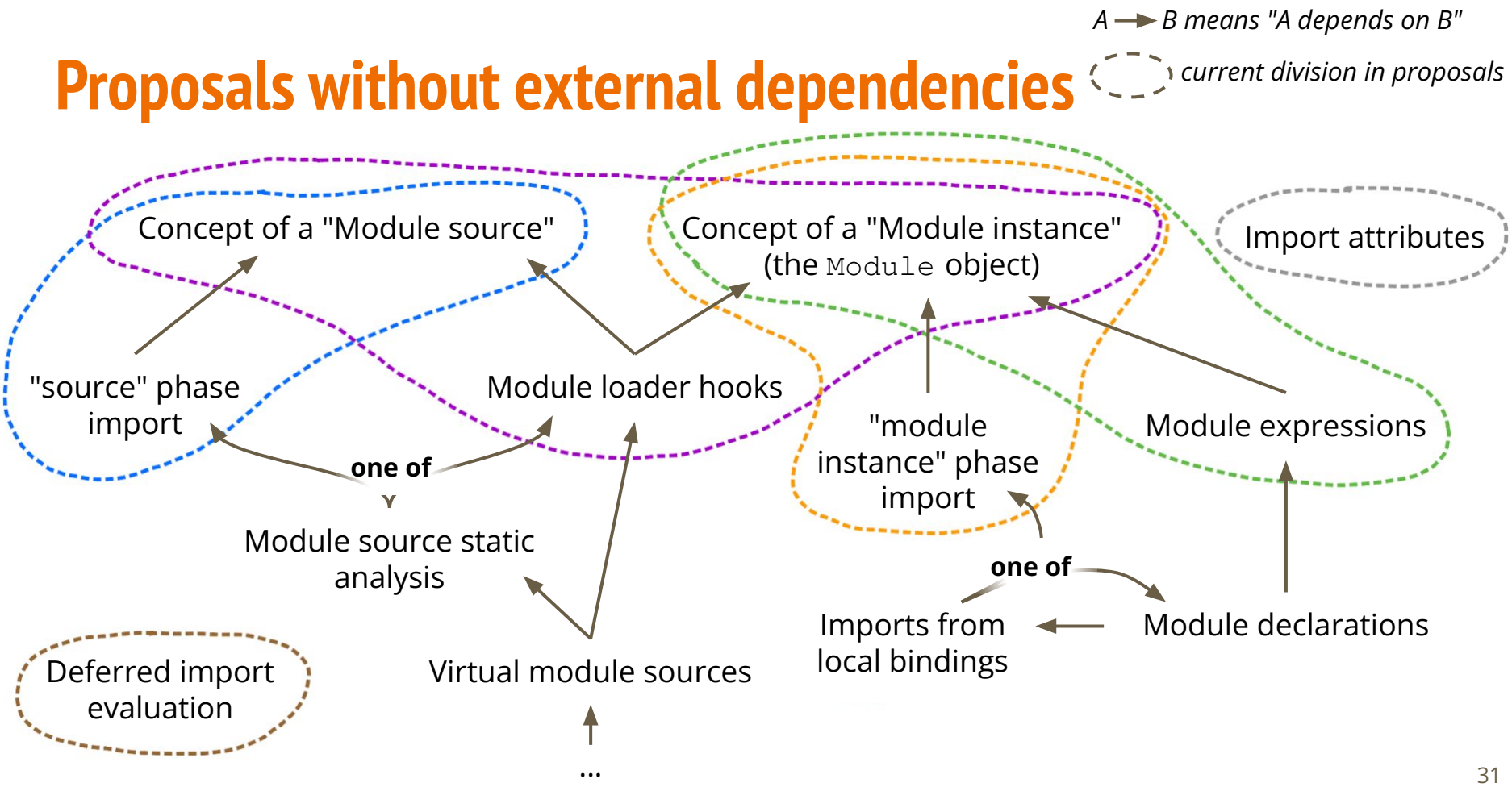
Dependencies



Dependencies



Proposals without external dependencies



Scope of the discussion

- ✓ Cross-proposal discussions
- ✓ Discussions about the general modules space
- ✓ Clarifying questions about cross-cutting concerns or individual proposals
- ✗ In-depth discussion regarding specific proposals are better stuired for those proposal's presentations/github issues.

Bonus section: use cases covered by any of the "Moule Harmony" proposal

Use case

Inline declaration of sharable code

Use case

JavaScript bundling

Use case

Wasm CSPs

Use case

Dependencies management

Use case

Reduce app initialization cost

Use case

Module virtualization and multiple instantiation

Use case

Custom module resolution

Use case

Custom module types