Preserving Modularity

Andreas Rossberg

Degrees of Modularity

Whole-program compilation & deployment, no linking

Separate compilation, static whole-program linking & deployment

Separate compilation, static incremental linking, whole-program deployment

Separate compilation & deployment, client-side whole-program linking

Separate compilation & deployment, client-side incremental linking

…all of these are possible today in Wasm!

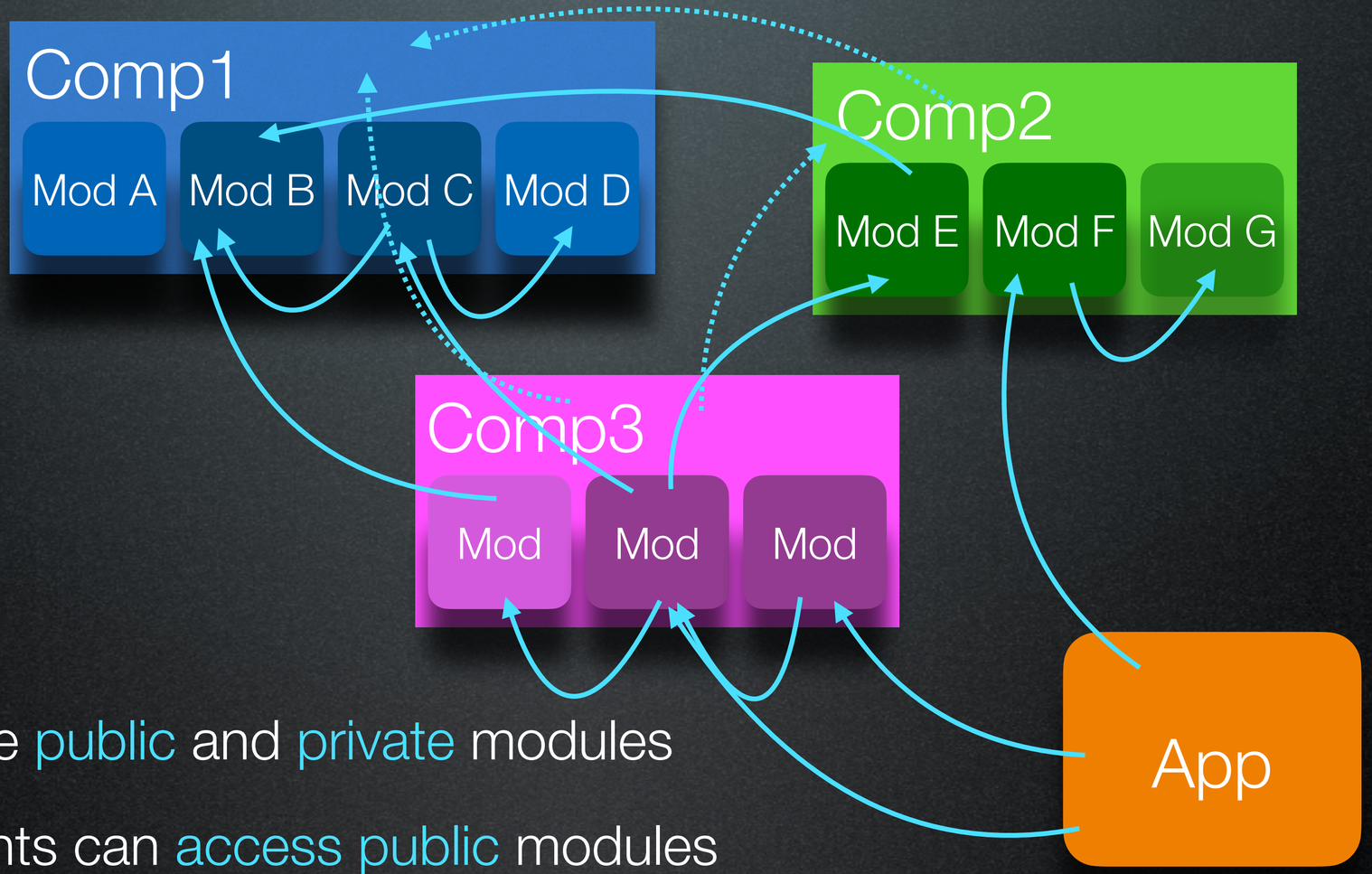
Degrees of Coupling

Shared-nothing components with language-agnostic interfaces
(cf. interface types, Luke's CG presentation)

Mixed degrees of couplings on multiple levels
(e.g., diverse components build from language modules and runtimes)

Shared-everything language-specific program modules
(mapping language-level modules and separate compilation to Wasm)

Use case: Layering

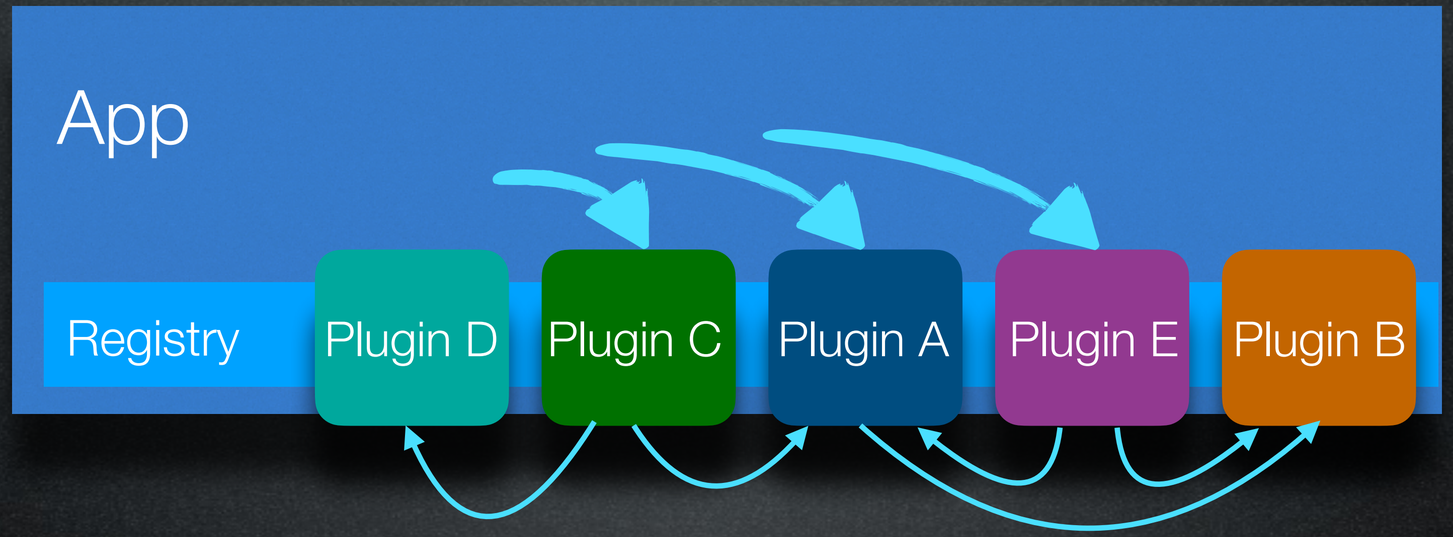


Components have public and private modules

App or components can access public modules

Internal linking logic is implementation detail of a component (cf. module linking proposal)

Use case: Plug-ins as DLLs



Plug-ins can be requested by app or by other plugins

Registry hands out entry points, e.g. through a function table

App makes no assumptions about plugin-to-plugin call signatures

Use case: REPL

(the ultimate plugin system)

(Wob demo)

Goals

Incremental linking

Dynamic linking

Compiling source modules into Wasm modules faithfully

Running in language-agnostic ecosystems

Linking multi-language apps (e.g., interface types)

(not possible if linking mechanisms are language-centric)

Modular Linking

Modularity is *not* just the ability to break up a program into a fixed set modules and stitching them back together

It means linking itself can be done in a modular fashion!

i.e., without requiring global knowledge in any part of an application

Modular Linking

This is all possible with Wasm as is today

Module linking proposal takes us further in that direction

Using GC must not regress modularity

Non-Goals

Top-down (“deferred”) linking against unknown types with no overhead

- …would require jitting
- …less efficient solution is possible with indirections and casts

Recursive linking

- …typically based on similar techniques
- …requires runtime checks and rather non-trivial type theory

Neither is possible today, hence out of scope

Modularity

Typical import model

Imports refer to other modules by name

…syntactic name, symbolic name, file path, URL, etc

…may involve some normalisation step (e.g., relative paths)

Multiple imports of the same name map to the same module instance

…sharing observable via state or other generative notions

…or prescribed informally by cost model

Imports are an implementation detail of each module

…clients do not know about transitive imports, only export signature matters

This is the essence of many module systems, static and dynamic

Implications

A module does not know its clients

A client of a module does not know about other clients

No single point in a program knows all clients of any given module

(A linker or loader may compute the full client set of a module,
but only when doing whole-program linking)

With incremental linking:

- Can use a module before linking (or even knowing) other/all clients
- Yet sharing is observable for clients linked later

Example: JS loader

```
let registry = {__proto__: null};

async function link(name) {
  if (! (name in registry)) {
    let binary = await fs.readFile(name + ".wasm", “binary");
    let module = await WebAssembly.compile(arraybuffer(binary));

    for (let im of WebAssembly.Module.imports(module)) {
      await link(im.module);
    }

    let instance = await WebAssembly.instantiate(module, registry);
    registry[name] = instance.exports;
  }
  return registry[name];
}

async function run(mainname) {
  let exports = await link(mainname);
  return exports.return;
}
```

Enter: Structural data

Structural Types

Not a fringe problem,
present in pretty much every high-level language

arrays, tuples, functions, closures, pointers, …

records, objects, variants, nested modules, …

Arise naturally at module boundaries as well

And in lower-level APIs, if shared memory is to be avoided

…need to be able to pass aggregated data

Compiling Structural Types

Need to lower structural source types to Wasm types

While preserving module dependency order

…if not, modularity isn't mapped faithfully

There is an infinite number of structural types

…with possibly a very large number of representations

…so they cannot be predefined in language runtime

Example: Structural Types in Wob

Bool[] ⇝ array i8

(Int16, Bool, Float, C) ⇝ struct (field i16 i8 f64 (ref $C-inst))

…needed for unboxing of non-reference types

…as well as cast avoidance of reference types

…the number of tuple types is infinite, even if bounded by length

(Bool, Bool) -> Bool ⇝ struct (field (ref $i8_i8->i8-code)) = $i8_i8->i8-clos

where: $i8*i8->i8-code = func (param (ref $i8_i8->i8-clos) i8 i8) (result i8)

…the number of closure types is again infinite

Example: Structural Types in Waml

…tuples, closures, etc as before

```
module M : {
  type T;
  val x : Bool;
  val f : Int -> T;
  val g : T -> Int;
}
```

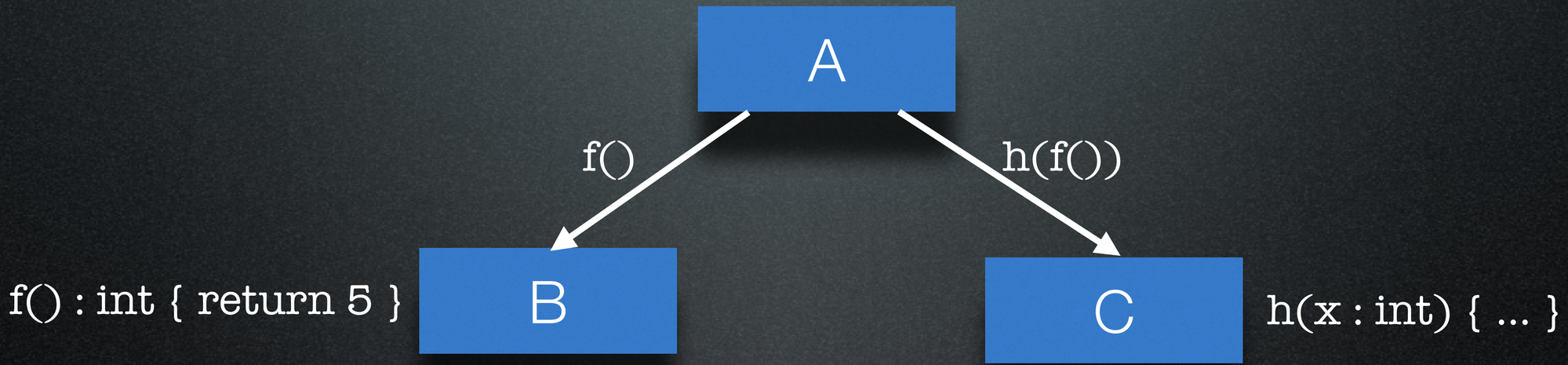
⇝ struct (field i8 (ref $int->any-clos) (ref $any->int—clos))

…again needed for both unboxing and cast avoidance (ought to be zero-overhead)

…number of module types is again infinite

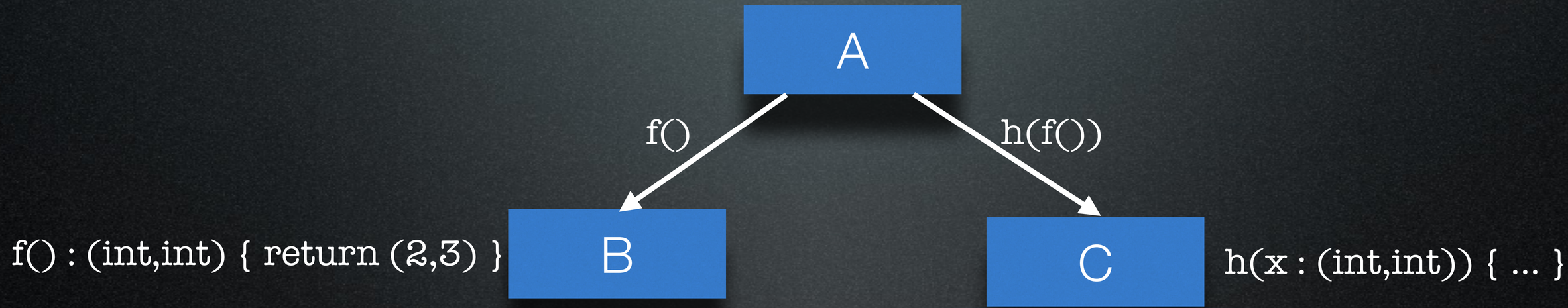
NB: module subtyping is structural and usually implemented coercively in ML compilers!

Example: V import



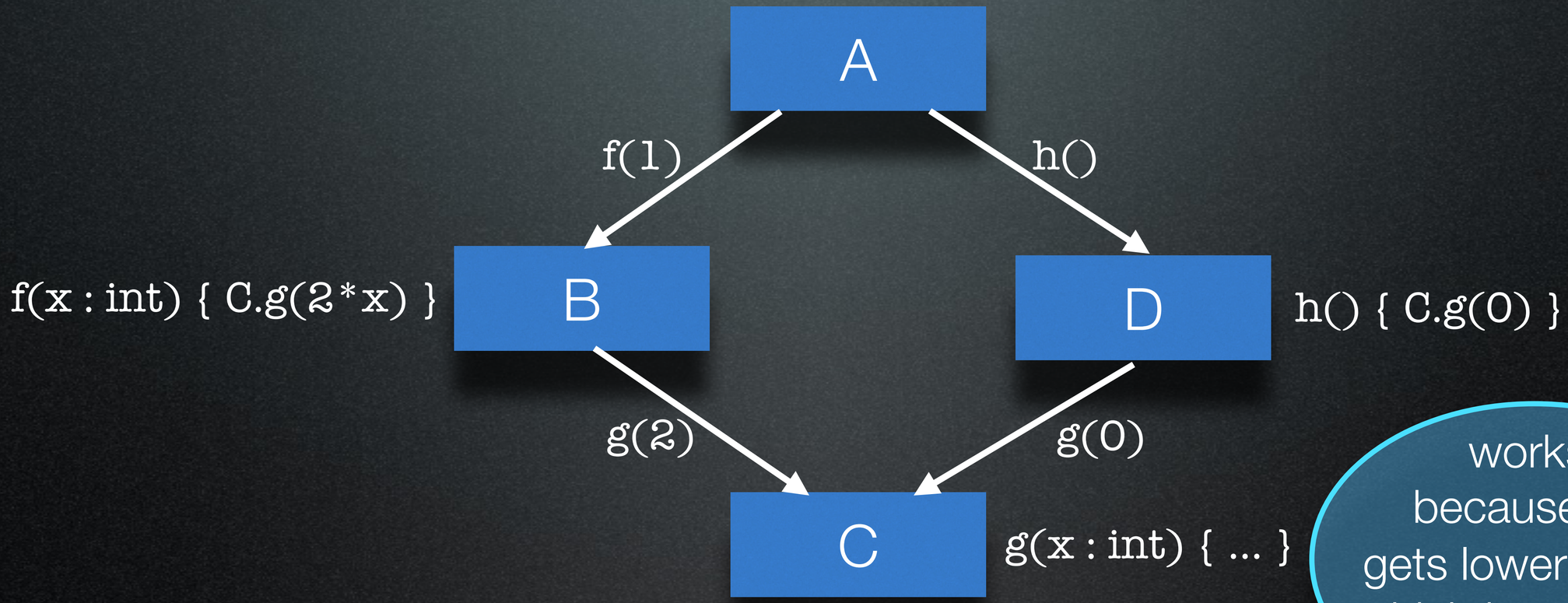
B and C are independently developed libraries

Example: V import



B and C are independently developed libraries

Example: Diamond import

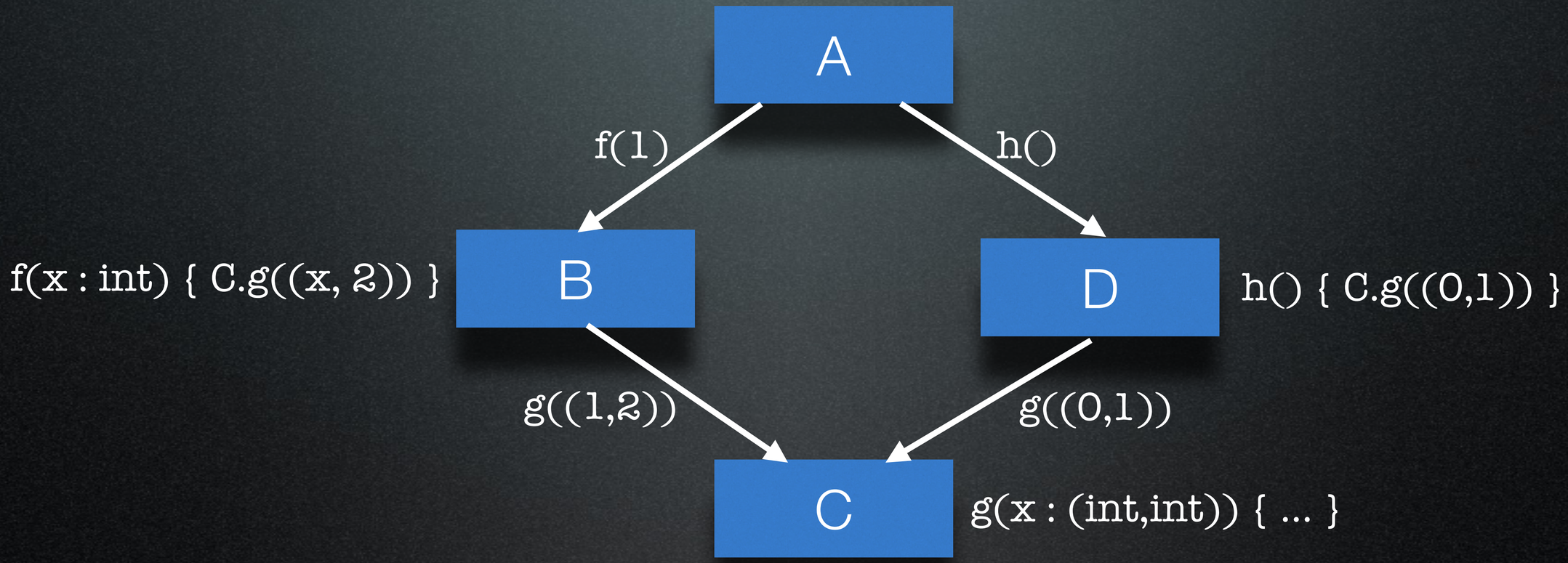


works, because **int** gets lower to **i32**, which is structural

B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

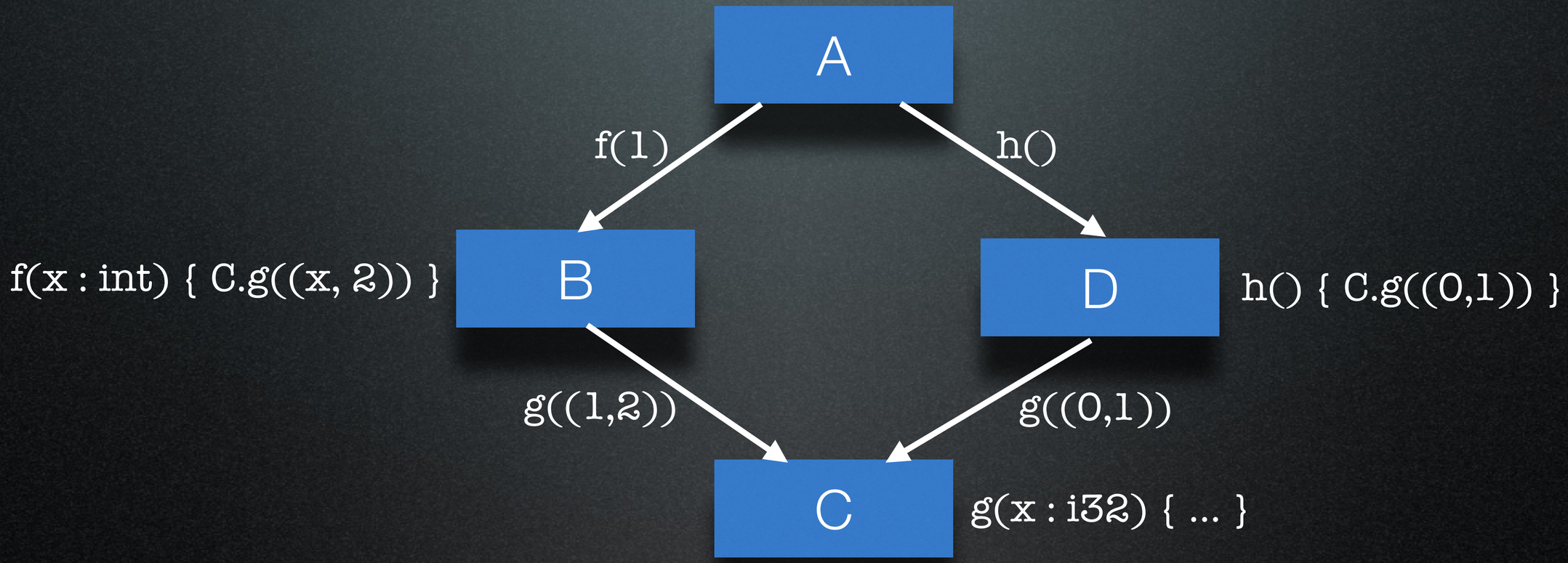
Example: Diamond import



B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

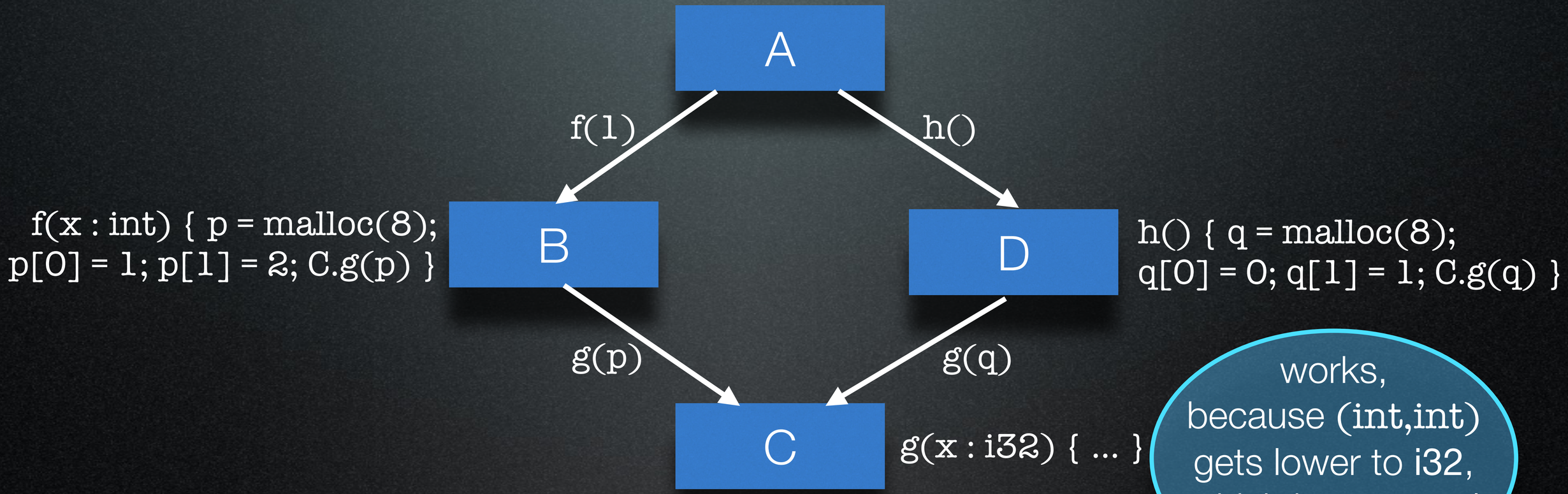
Diamond import, linear memory



B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

Diamond import, linear memory



B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

Observations

B and D independent

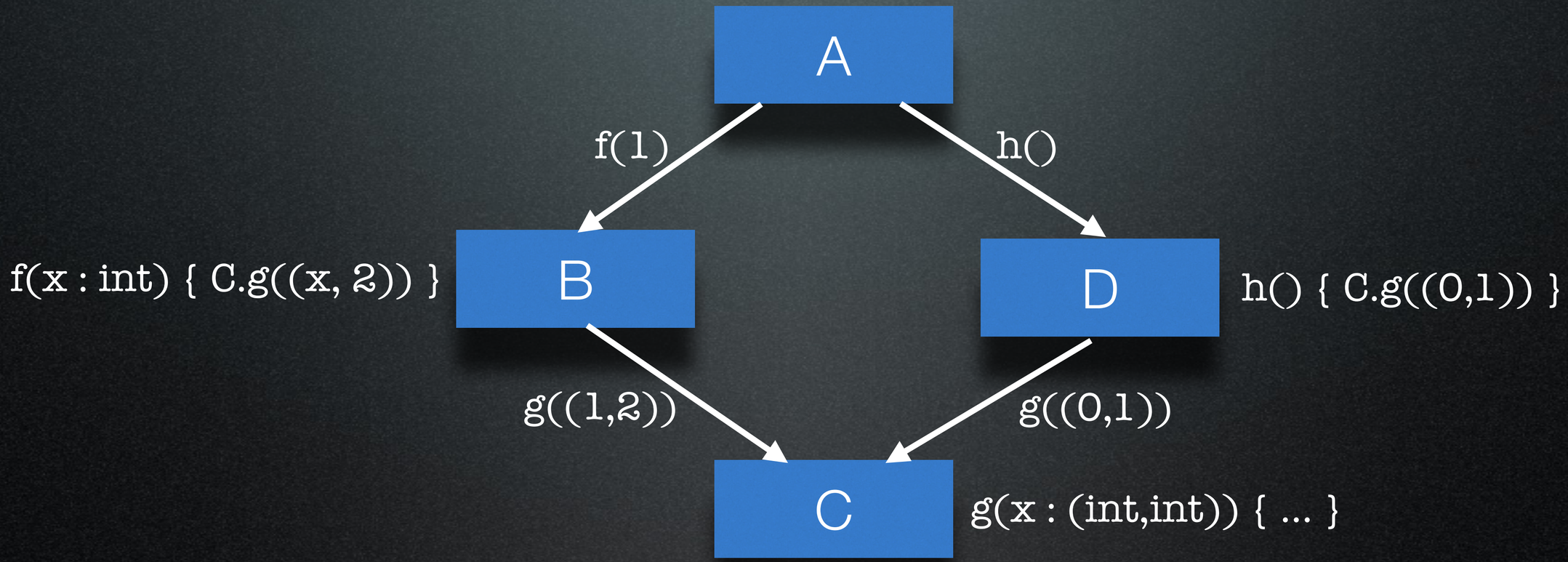
B, C, and D share knowledge about data layout

But this only requires (out-of-band) convention,
not (in-band) coordination

A is agnostic to C and the use of tuples in B and D

Module dependencies preserved

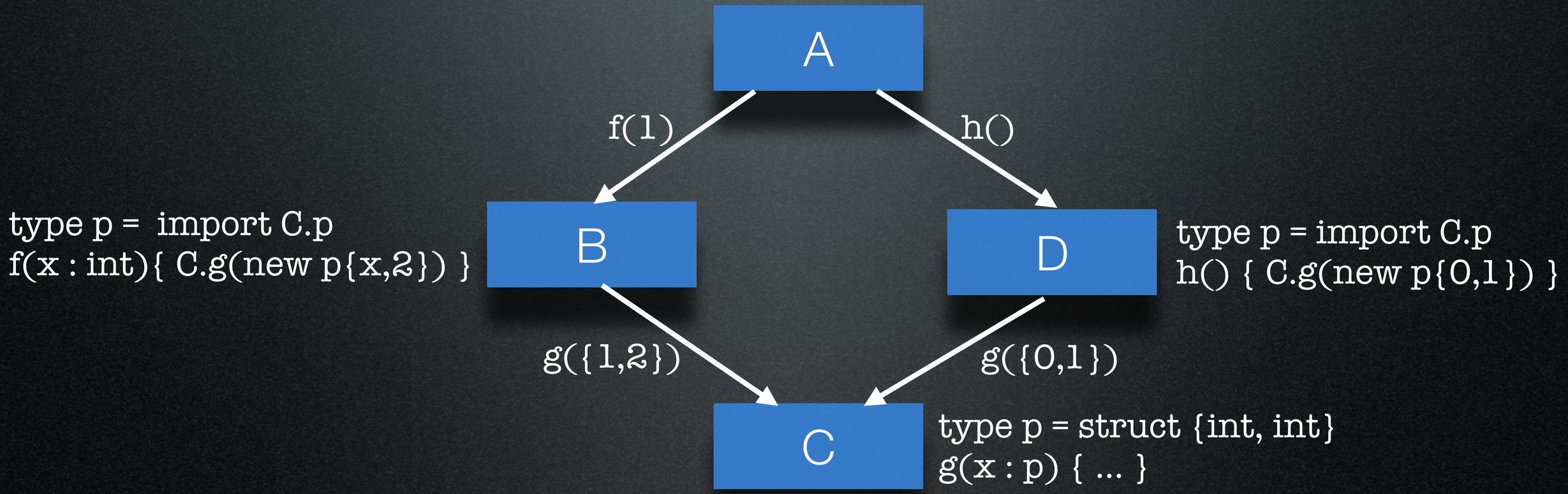
Diamond import, structural types



B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

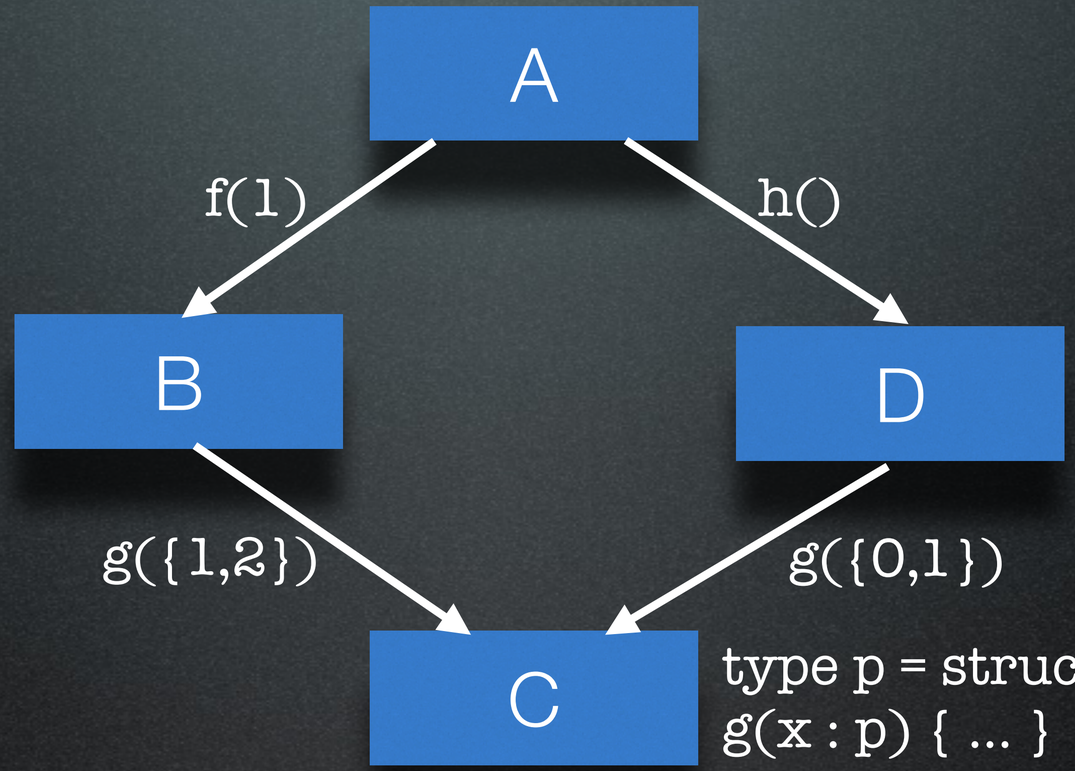
Diamond import, structural types



B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

Diamond import, structural types




```
type p =  struct {int,int}
f(x : int){ C.g(new p{x,2}) }
```

```
type p = struct {int,int}
h() { C.g(new p{0,1}) }
```

```
type p = struct {int,int}
g(x : p) { ... }
```

works, because (int,int) gets lower to **struct**, which is structural

B and D are independently developed libraries

A is unaware of C, implementation detail of B and D

Observations

B and D independent

B, C, and D share knowledge about data layout

But this only requires convention, not coordination

A is agnostic to C and the use of tuples in B and D

Module dependency not affected

In other words: everything works as with linear memory

Expressing structural data with nominal types

The Problem

Cannot replicate nominal type definitions

All potential clients must share a single type definition

Requires coordination, not just convention!

In fact, global coordination, i.e., centralisation

- …because type may scope over any number of components
- …cannot generally tell extend of scope locally or statically
- …with dynamic linking, scope may grow over time

Proposed Solution

Modules don't export "structural" types, but import them

…this way, single nominal definition can be shared

But import from where? Two known answers:

1. supplied by the client of each module
2. from a central infrastructure module (or set thereof)

Attempt 1: Import inversion

Import Inversion

Client of a module supplies type imports

To achieve sharing,
client supplies the same types to multiple imports

Detour: Referencing other modules

Definite module references:

- refers to specific (logical) module by name
- implementing module gets to chose

Indefinite module reference:

- declare a module parameter by interface
- client module gets to choose

Most module systems do not distinguish the two properly,
though analogous to difference between variables in scope and function parameters

Wasm only has indefinite references, but decorated with import names

- allows delegating to generic loader to interpret indefinite as definite
- but no established convention for distinguishing intent

Adding module parameters

Assume Wasm and module linking are extended with a mechanism for supplying parameters to module imports

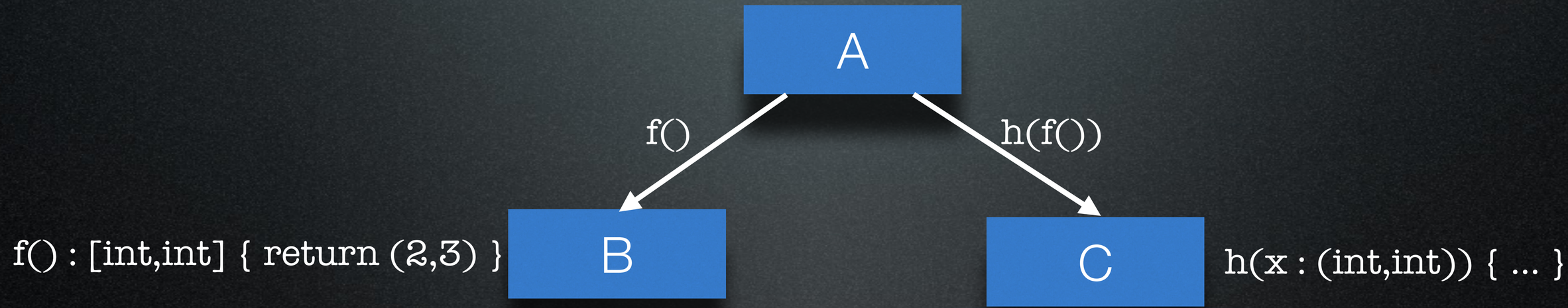
import “C” **with type** “t” = $t

Roughly amounts to **module.bind** before invoking loader

A loader registry now needs to index not just by name, but also index by parameters!

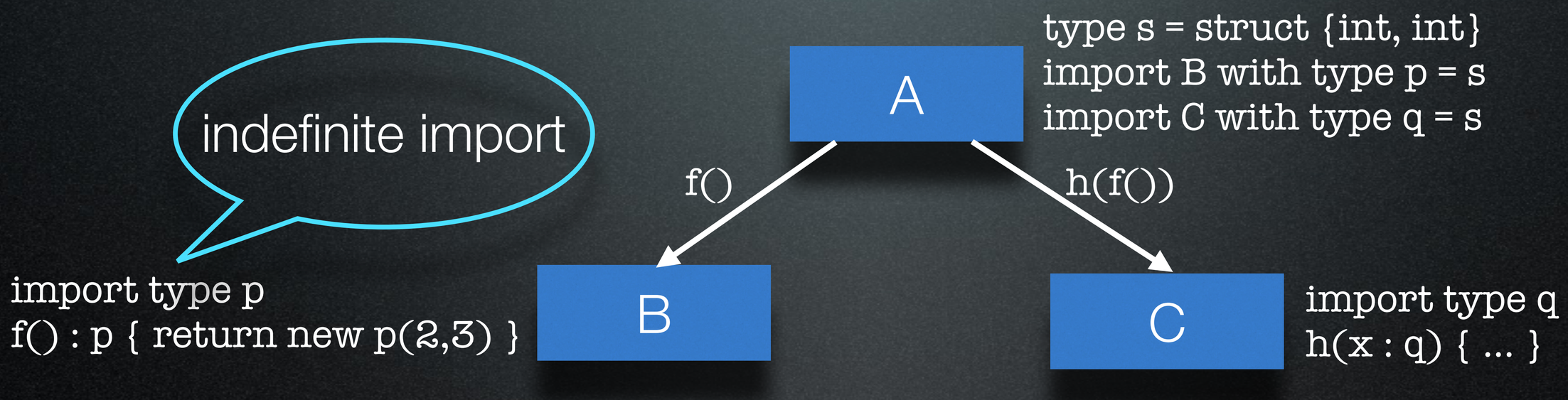
Tools also need to understand this

Example: V import



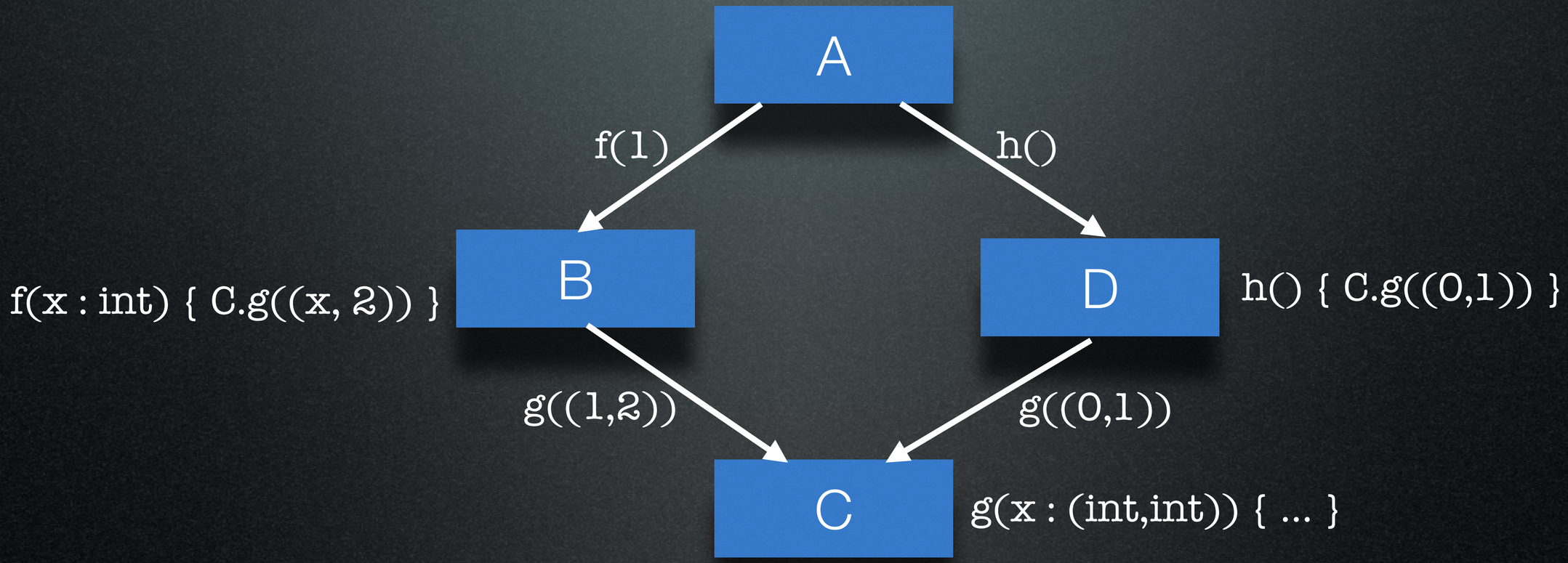
B and C are independently developed libraries

V import, nominal inversion

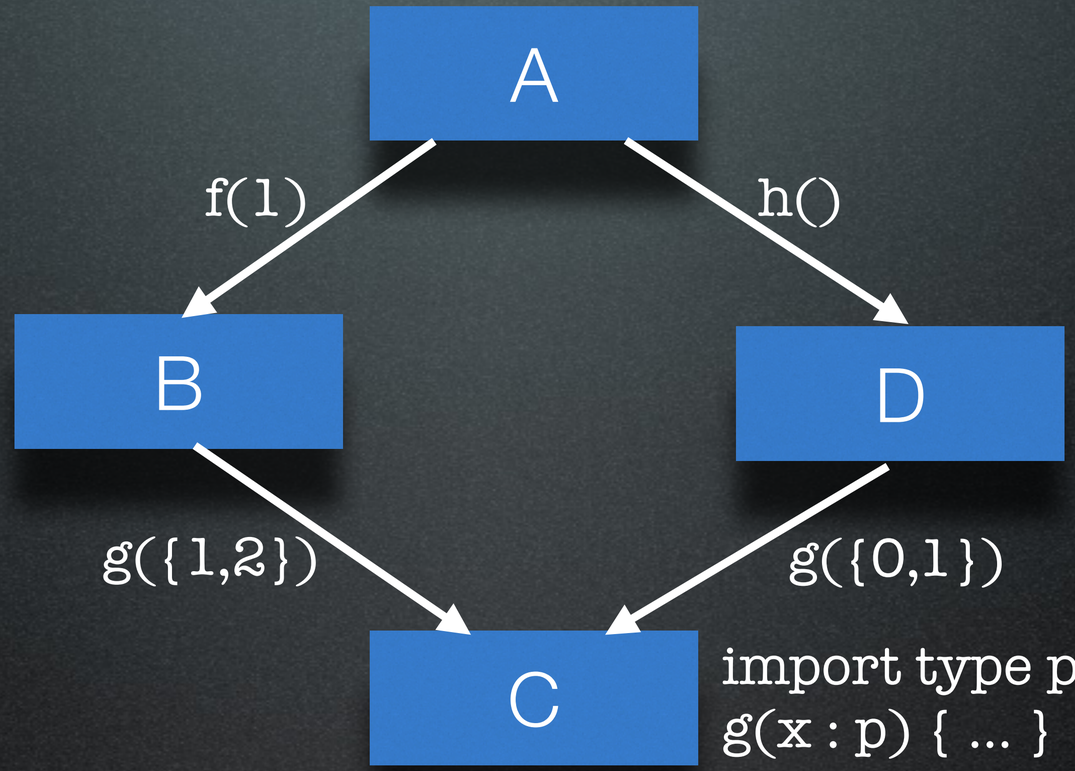


B and C are independently developed libraries

Diamond import



Diamond import, nominal inversion

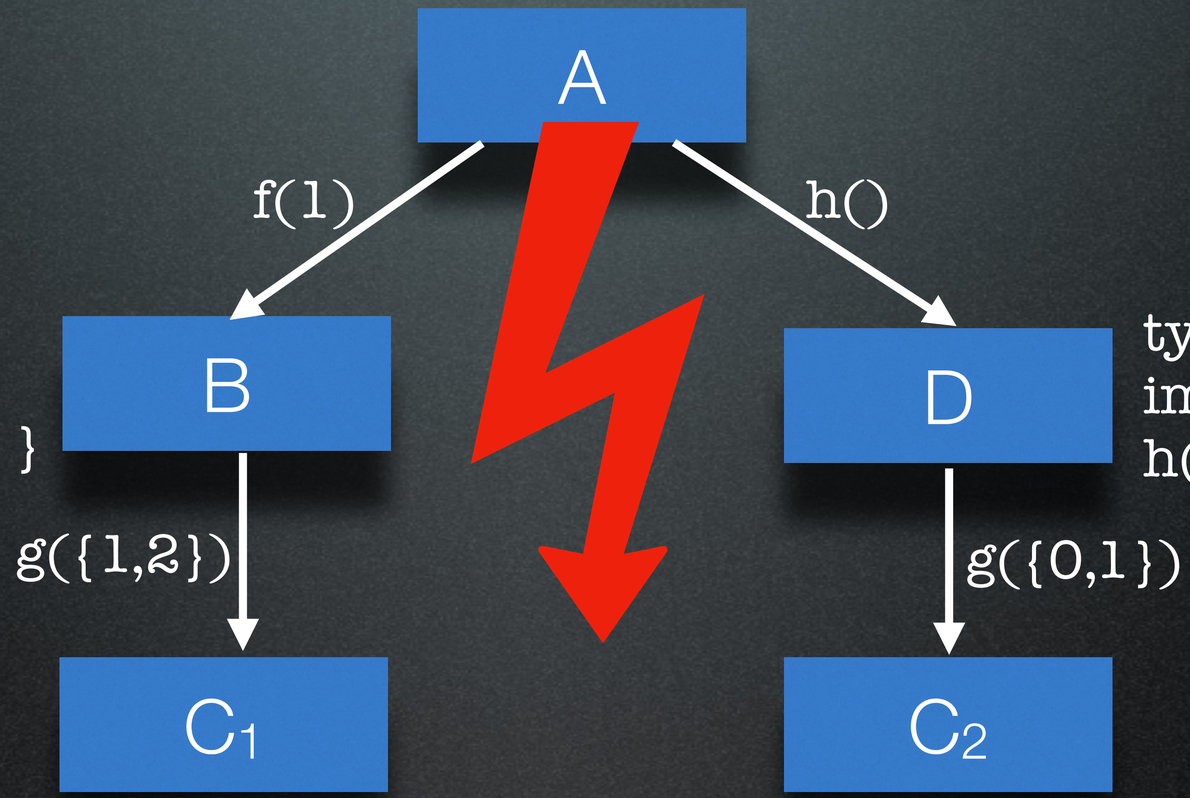



```
type p =  struct {int,int}
import C with type p
f(x : int){ C.g(new p{x,2}) }
```

```
type p = struct {int,int}
import C with type p
h() { C.g(new p{0,1}) }
```

```
import type p
g(x : p) { ... }
```

Diamond import, nominal inversion



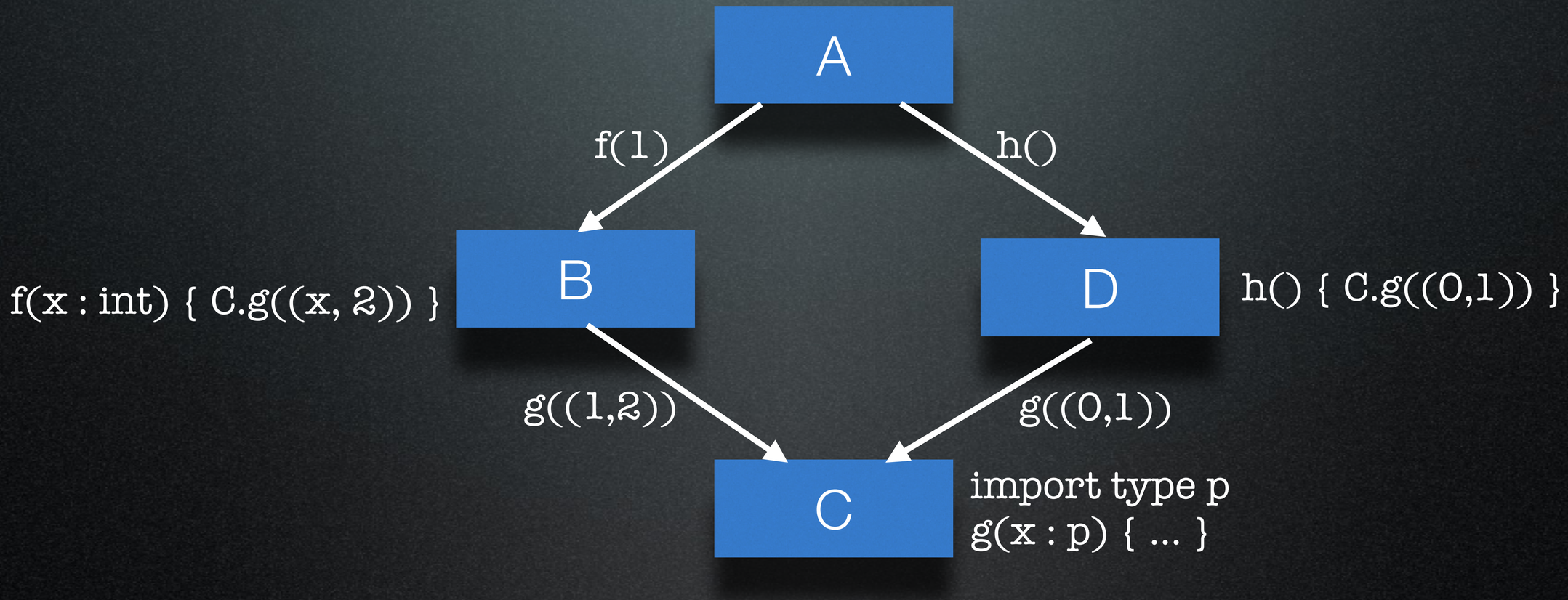
type p = struct {int,int}
import C with type p
f(x : int){ C.g(new p{x,2}) }

type p = struct {int,int}
import C with type p
h() { C.g(new p{0,1}) }

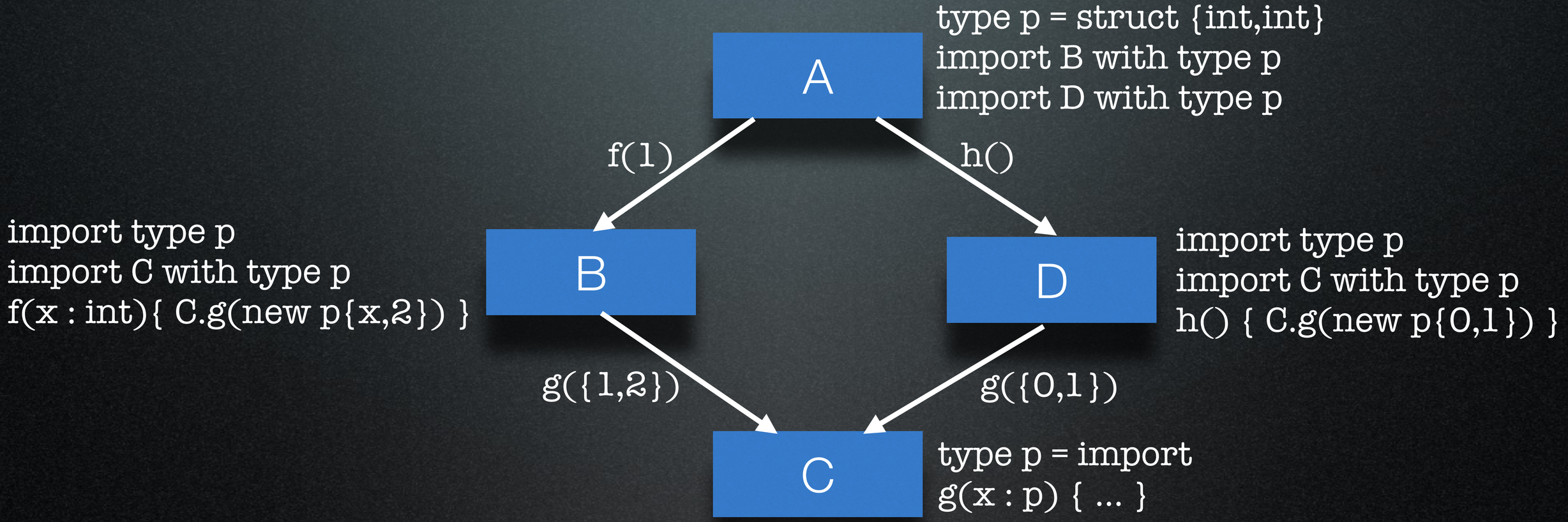
import type p
g(x : p) { ... }

import type p
g(x : p) { ... }

Diamond import, nominal, take 2



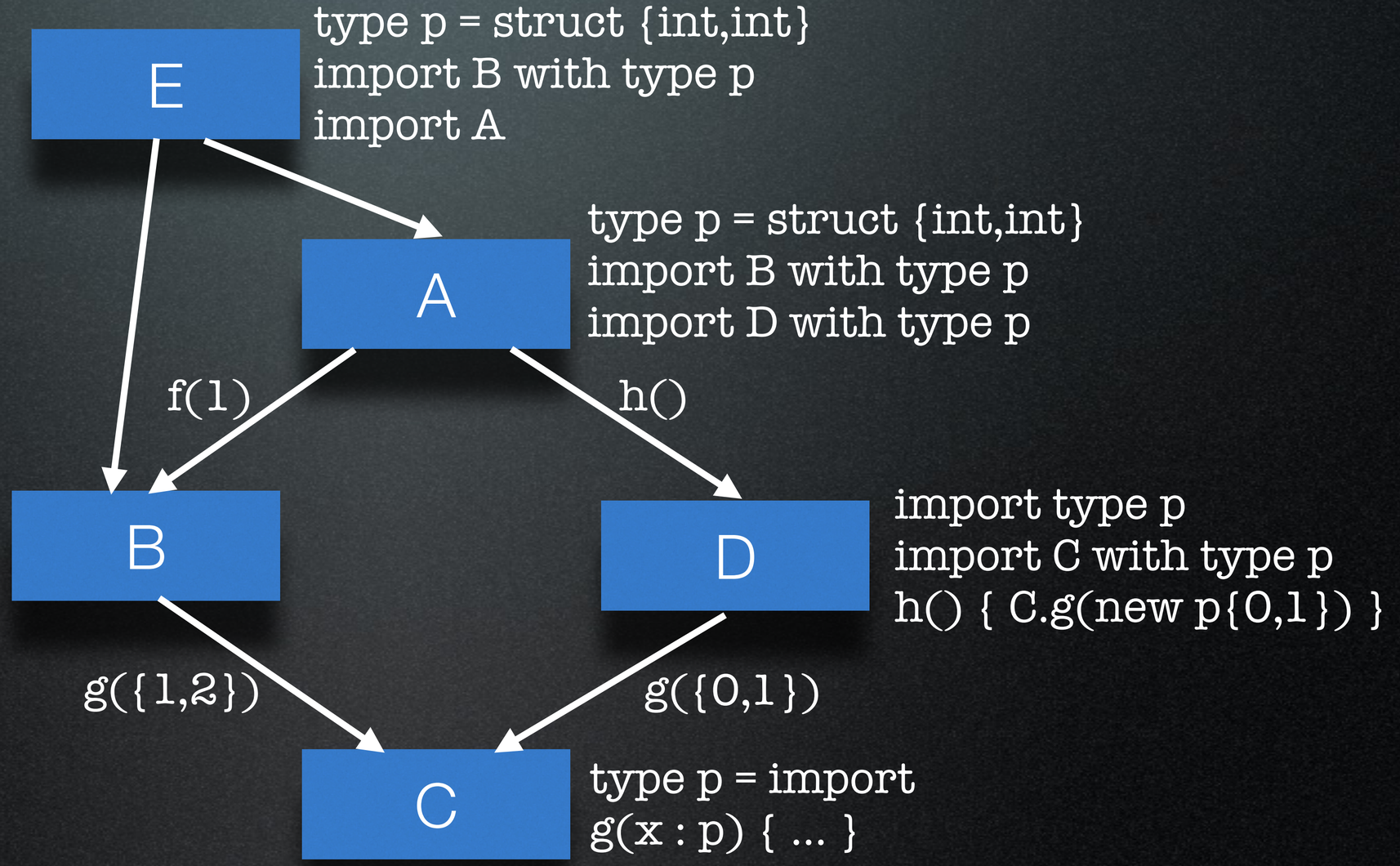
Diamond import, nominal, take 2



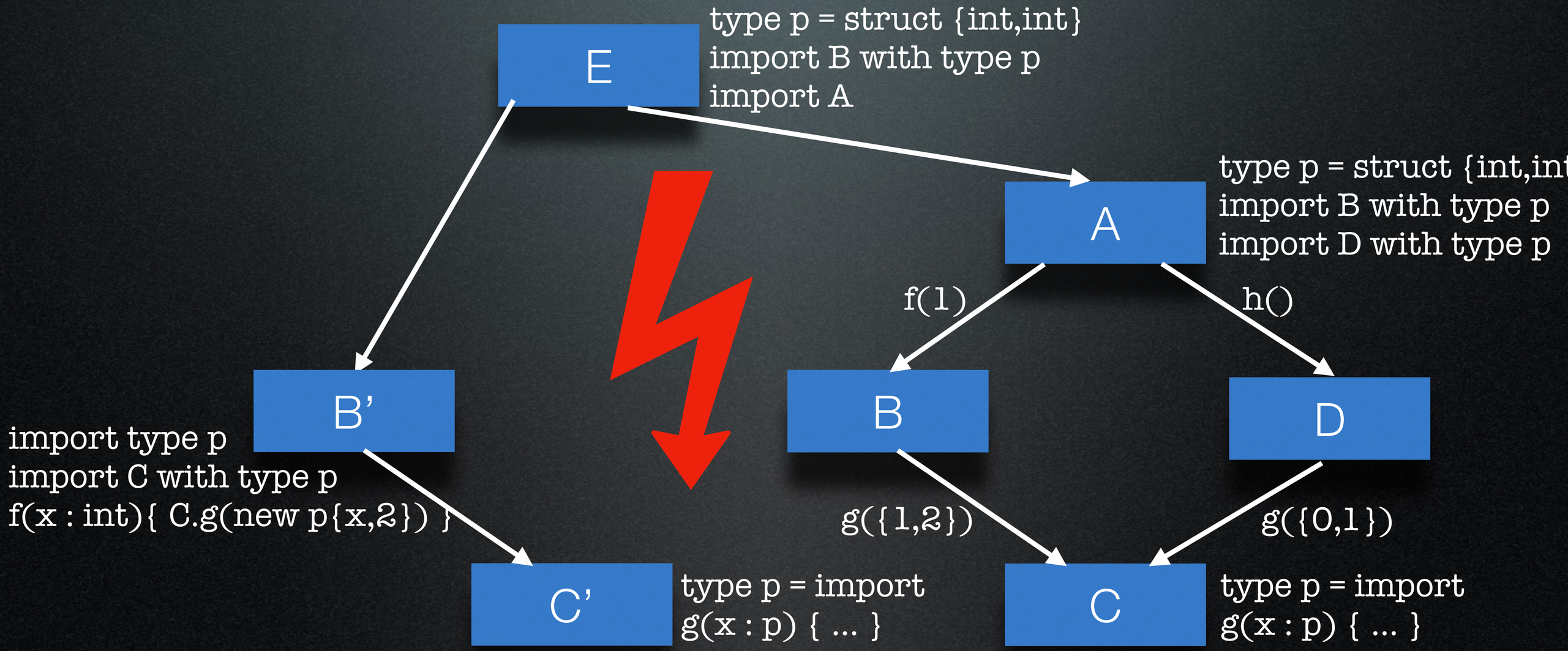
A no longer unaware of C — abstraction leak!

Type **p** did not occur in either B's or D's source interface

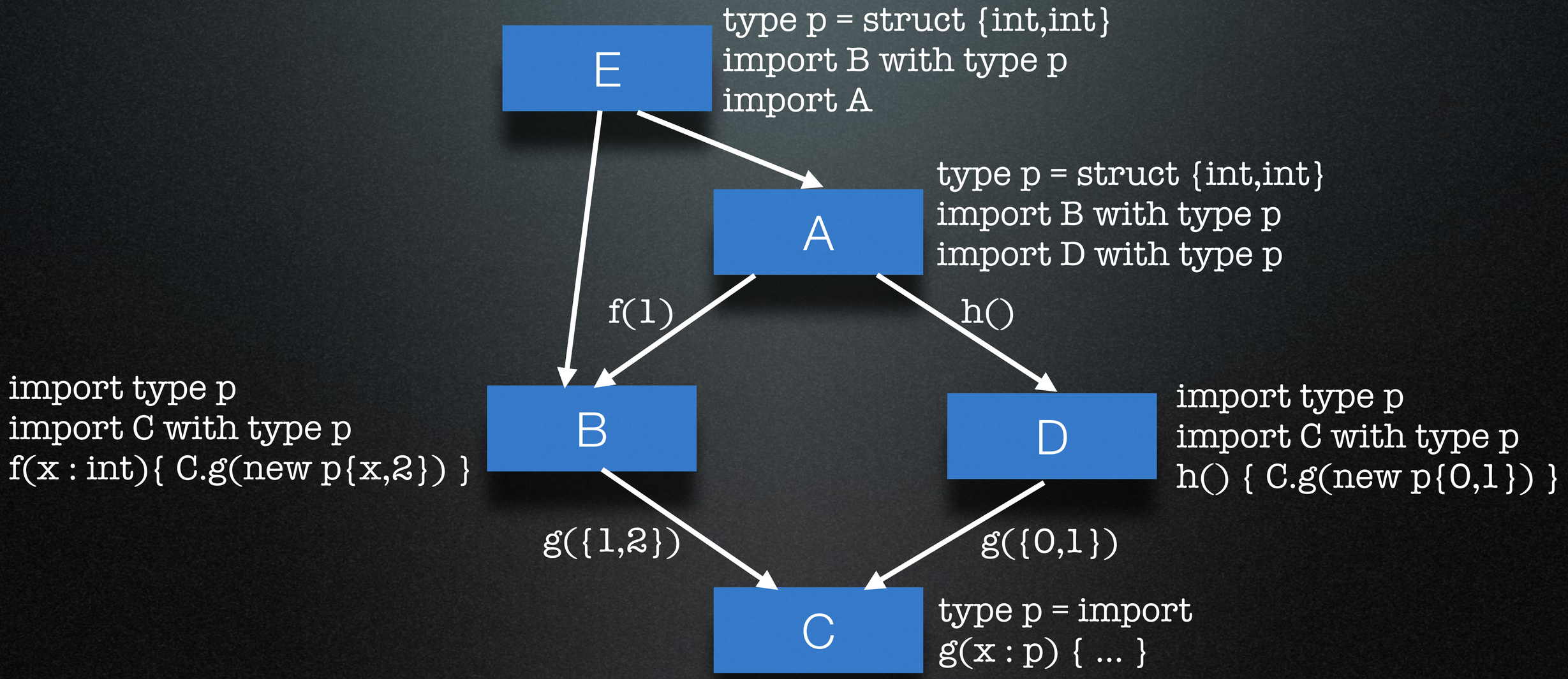
Diamond import, nominal, take 2



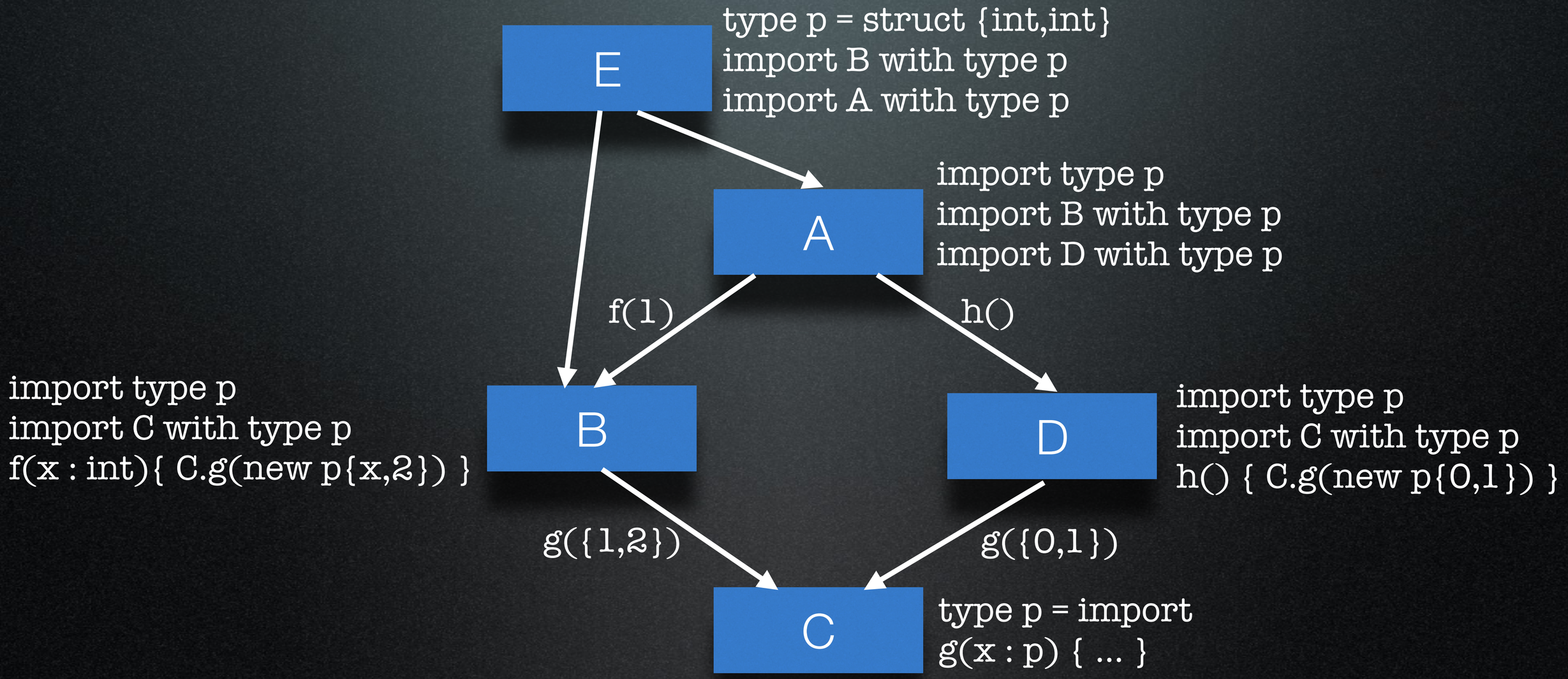
Diamond import, nominal, take 2



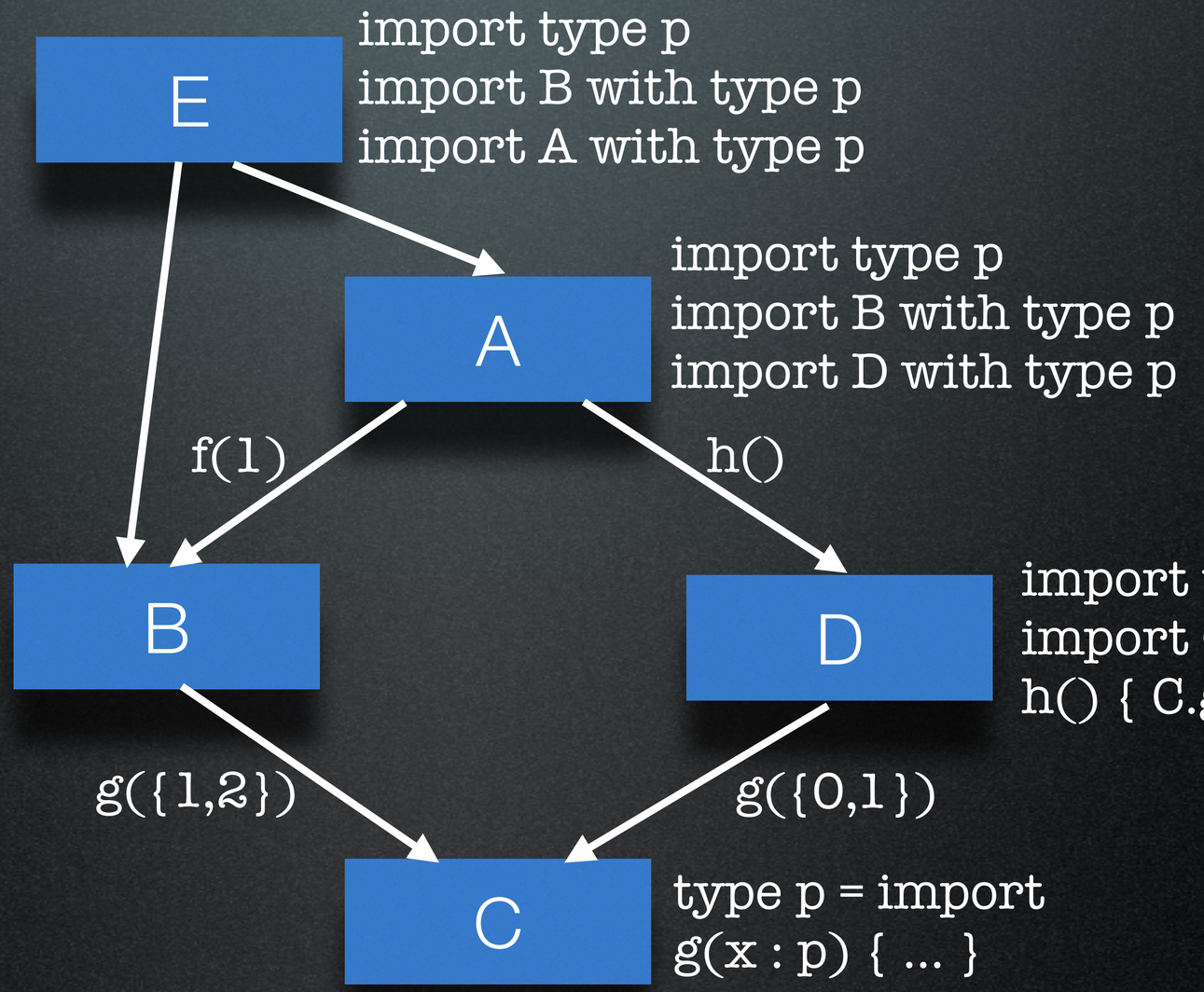
Diamond import, nominal, take 3



Diamond import, nominal, take 3



Diamond import, nominal, take 3



Implications

Repeats on every layer: Turtles all the way up!

Abstraction leak all the way up: every module needs to import all types from all lower layers

- even if they are not used in its own interface

Import blow-up: the transitive closure of types accumulates

- can be exponential!

Whole-program: some centralised root module has to be the “linking script” that generates all the types

- dynamic loading would require magic again

Remark: exportimport

Does not change the picture

To achieve sharing, have to be used as imports all the way up anyway, except in root module

Semantically, merely a shorthand for generating a type and supplying it locally (like, in the root module)

Conclusion on import inversion

Sharing only possible with exponential abstraction leaks

Which essentially makes it whole-program as well

Requires extensions to language and tools
(import parameters)

Attempt 2: Type centralisation

Centralisation

Dedicated module (or set thereof) defines all program types

If that module is supposed to be regular Wasm,
then this approach is inherently whole-program

…because that module must “know” all types in program

Would not be able to support dynamic linking

…because the set of modules & types is never closed

Centralisation, with magic

Dedicated module is virtual and implemented outside Wasm

Need some import naming conventions for this to work

NB: variations possible, e.g., ship modules with aux modules defining each type,
and loader analyses and globally dedupes aux modules based on types they define

Centralisation, with magic

Each relevant language implements a custom loader that

- uses reflection to discover and analyse type imports
- generates and caches suitable type definitions
- in other words, emulates structural types
  and implements canonicalisation

Tool chain

Loader is not enough, every tool operating on modules in the toolchain potentially has to understand and implement this logic

…static linkers/mergers (no longer just concatenation)

…optimisers (if they work cross-module)

…packers (depending on how they work)

…loaders for every additional environment to run in
(possibly implemented in different languages per embedding)

Includes type equivalence, type canonicalisation, subtyping, etc.

Environment challenges

Not every environment has a place for custom loaders

…module linking proposal

…ES6 module integration

…some eco systems

Type magic would need to be a built-in feature where not

Diversity challenges

Who is in charge in a multi-language setting?

…if loaders are language-specfic,
requires multiple loaders to coexist (and interact?)

…or mechanism needs to be unified and shared

…likelihood of greenspunning

Compatibility challenges

For multiple tools or languages to share “the same logic”, they have to do so in a compatible manner

…need a well-defined semantics

…which means we need to standardise it anyway

…but outside of semantics is a fragile foundation

Thoughts

Ultimately, this is an attempt to outsource an essential part of the semantics…

…by *multiplying* the burden for everybody else

Complex infrastructure for s.th that's operationally irrelevant!

Breaks the goal of enabling self-hosting for all modules

Completely untried, no evidence it is practical and scalable

Types are a genuine language problem,
the language should provide a solution.

Aside: Casts & Runtime Types

Escape hatch for limitations of Wasm type system

Casts recover type information lost by static typing

Hence, casts over structural types must be structural

…otherwise we'd be back to square one

That's what rtt.canon (and rtt.sub) enable

Other solutions?

Modularity Summary

Modules and imports are a common source-language feature

Structural types are a common source-language feature

Need to be compilable together

- …efficiently, such that unboxing isn’t impossible
- …faithfully, such that modular linking is still possible

Natural use cases: layering, plugins, interpreters, …

GC types should not regress modularity