Name Binding and Name Resolution

Eelco Visser

TU Delft

CS4200 | Compiler Construction | September 24, 2020

Type Constraints

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT().
```

Head

Premises

For all s, e1, e2

If the premises are true, the head is true

Representing Name Binding with Scope Graphs

```
rules

  typeOfExp(s, e@Add(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT() | error $[integer expected]@e1,
    typeOfExp(s, e2) == INT() | error $[integer expected]@e2,
    @e.type := INT().
```

error message on constraint failure

set type attribute

```
$ 9 * 10 + 3
```

Type: INT

```
19 $ 1 * true
```

integer expected
> INT() == BOOL()
> statics/base!typeOfExp(Scope("","s_1-1"), True(), INT())
> statics/base!typeOfExp(Scope("","s_1-1"), Mul(Int("1"),True()), INT())
> statics/base!declOk(Scope("","s_1-1"), Exp(Mul(Int("1"),True())))
> statics/base!declsOk(Scope("","s_1-1"), [Exp(Mul(Int(...),True()))])
> ... trace truncated ...

Type: BOOL

Constraints with Error Messages

```
constraint | error $[message [term]]@origin
```

Programs with Names

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Name resolution complicates type checkers, compilers

Ad hoc non-declarative treatment

A systematic, uniform approach to name resolution?

Formalizing Name Binding in Type Systems

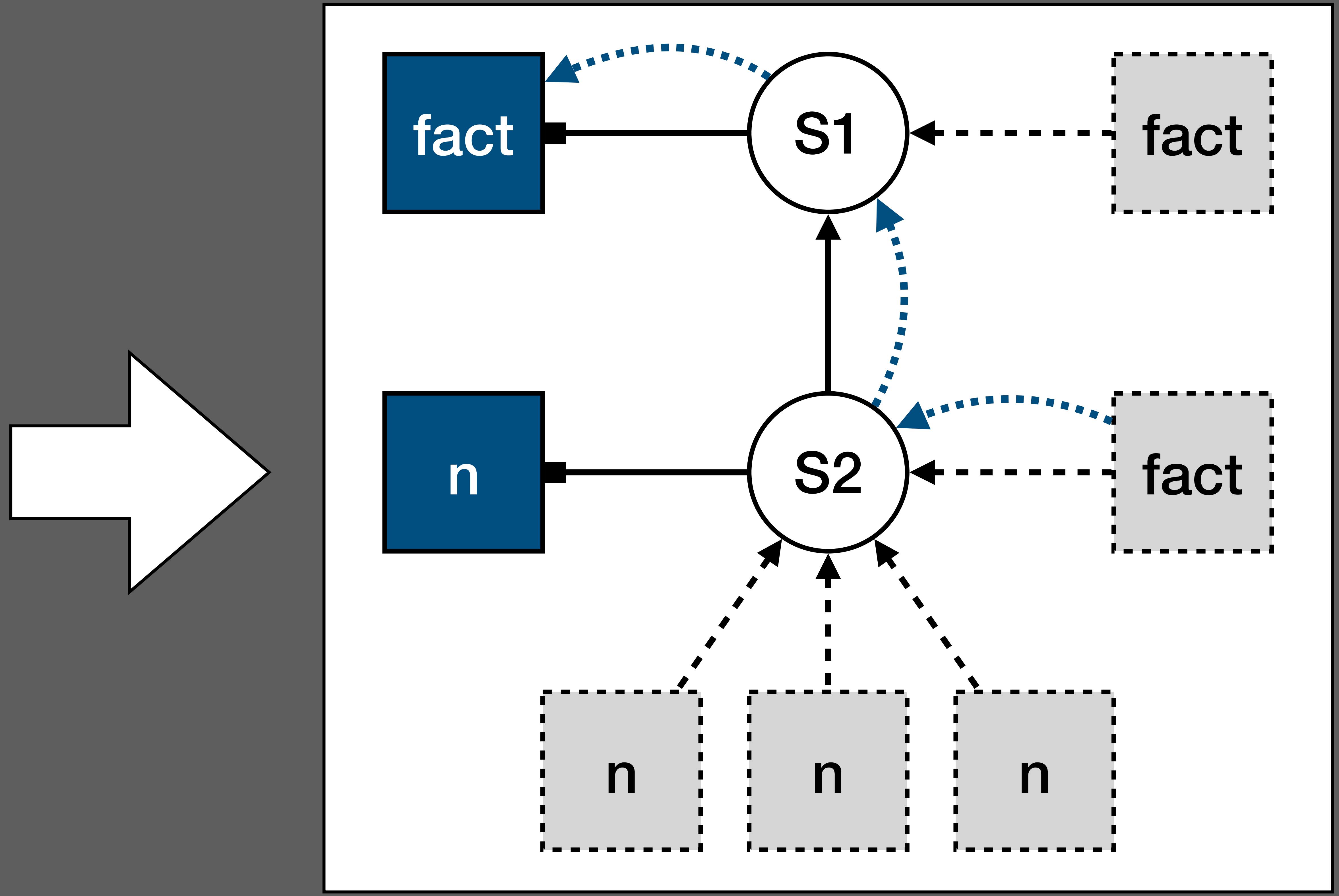
$$\frac{O(id) = T, \text{where } T \text{ is not a function type.}}{O, M, C, R \vdash id : T} \quad [\text{VAR-READ}]$$

Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph



Name Resolution

Name Resolution with Scope Graphs in Statix

Declarations and References

Lexical Scope

Modules

Records

Permission to Extend

Scheduling Resolution

Declaring and Resolving Names

Declarations and References

```
signature
  constructors
    Var   : ID → Exp
    Bind  : ID * Exp → Bind
    BindT : ID * Type * Exp → Bind
    Def   : Bind → Decl
```

```
rules

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
```

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
def a : Int = 0
def b : Int = a + 3
def c : Int = a + b
> a + b + c
```

typed declarations

```
def a = true
def b : Int = a
def c = 1 + b
def e = b && c
```

type mismatch

```
def a = 0
def b = a + 1
def c = a + d
> a + e + c
```

undefined variable

```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

duplicate definition

```
> a + b + c
def a = 0
def c = a + b
def b = a + 1
```

use before definition

Declarations and References

```
signature
  constructors
    Var   : ID → Exp
    Bind  : ID * Exp → Bind
    BindT : ID * Type * Exp → Bind
    Def   : Bind → Decl
```

```
rules

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
```

```
rules

  typeOfExp(s, Var(x)) = T :-
    typeOfVar(s, x) == T.

  declOk(s, Def(bind)) :-
    bindOk(s, s, bind).

  bindOk(s_bnd, s_ctx, BindT(x, t, e)) :- {T1 T2}
    typeOfType(s_ctx, t) == T1,
    declareVar(s_bnd, x, T1),
    typeOfExp(s_ctx, e) == T2,
    subtype(T2, T1).

  bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
    typeOfExp(s_ctx, e) == T,
    declareVar(s_bnd, x, T).
```

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
def a : Int = 0
def b : Int = a + 3
def c : Int = a + b
> a + b + c
```

typed declarations

```
def a = true
def b : Int = a
def c = 1 + b
def e = b && c
```

type mismatch

```
def a = 0
def b = a + 1
def c = a + d
> a + e + c
```

undefined variable

```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

duplicate definition

```
> a + b + c
def a = 0
def c = a + b
def b = a + 1
```

use before definition

Representing Name Binding with Scope Graphs

```
rules

  declareVar : scope * ID * TYPE

  typeOfVar  : scope * ID --> TYPE
```

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

Scope Graphs: Declarations

```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE

  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s.
```

declaration relation

variable x is declared in scope s with type T

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

Scope Graphs: Declarations

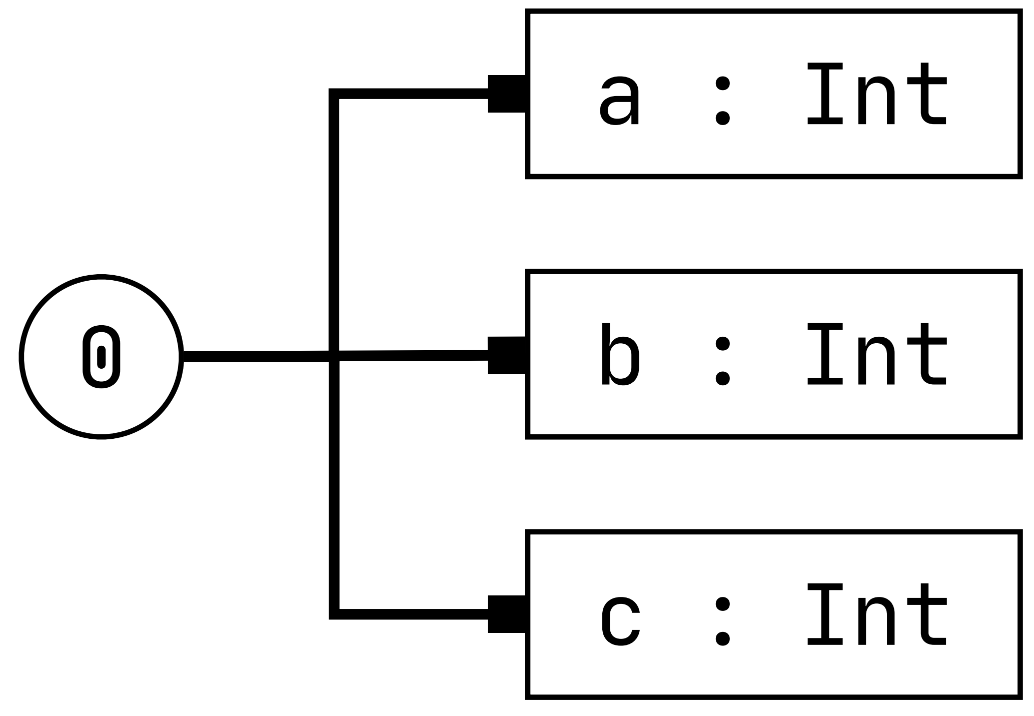
```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE

  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s.
```

declaration relation

variable x is declared in scope s with type T

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference



Scope Graphs: Name Resolution Queries

```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE
  resolveVar : scope * ID → list((path * (ID * TYPE)))
  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s.



  resolveVar(s, x) = ps :-
    query var
      filter e and { x' :- x' == x }
         min and true
          in s ⟼ ps.

  typeOfVar(s, x) = T :- {x'}
    resolveVar(s, x) == [(_,(x', T))].
```

declaration relation

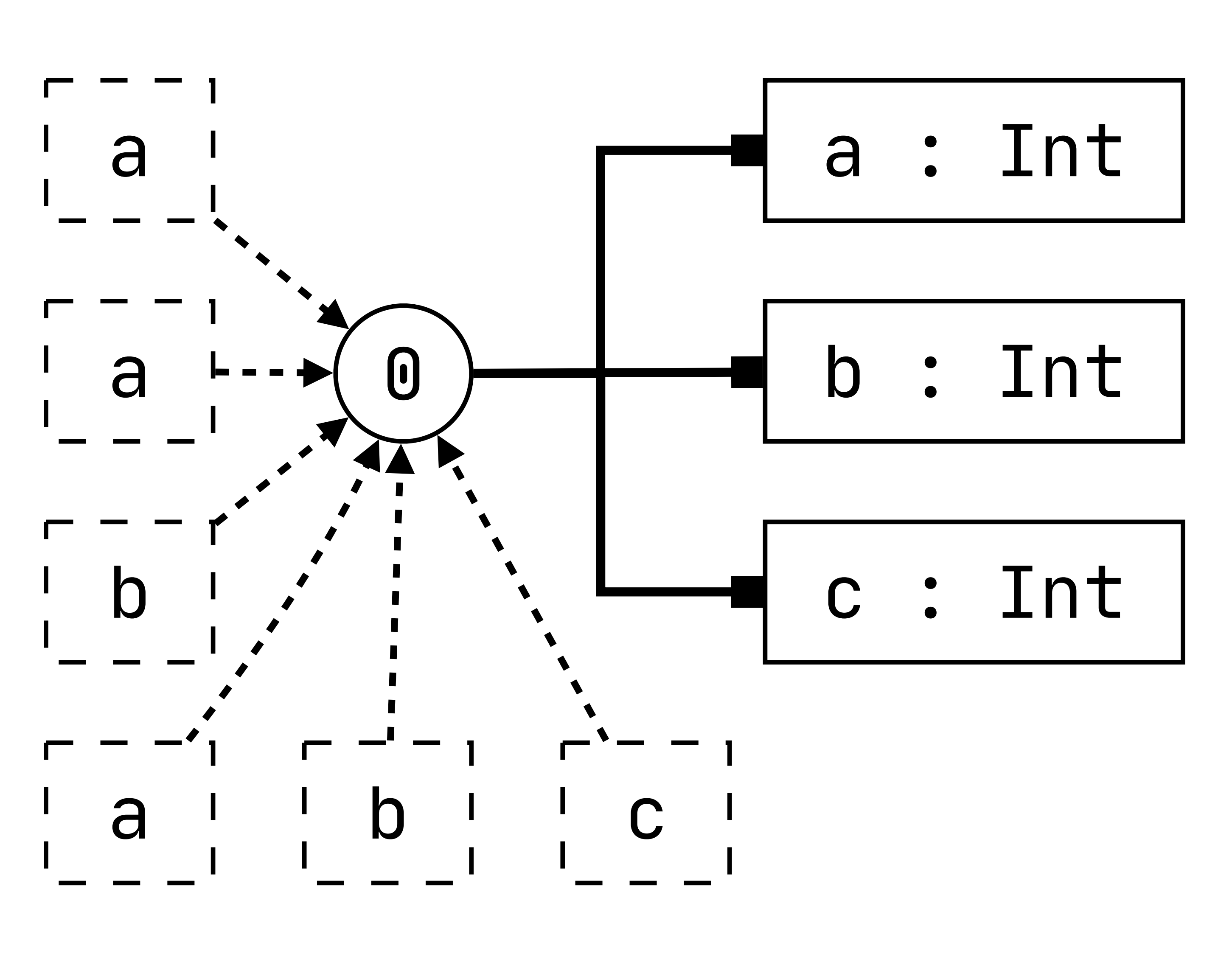
variable x is declared in scope s with type T

variable x in scope s resolves to list of declarations

variable x in scope s resolves to declaration x' with type T

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference



Undefined Variable

```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE
  resolveVar : scope * ID → list((path * (ID * TYPE)))
  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s.

  resolveVar(s, x) = ps :-
    query var
      filter e and { x' :- x' == x }
         min and true
          in s ↦ ps.

  typeOfVar(s, x) = T :- {x'}
    resolveVar(s, x) == [(_,(x', T))].
```

declaration relation

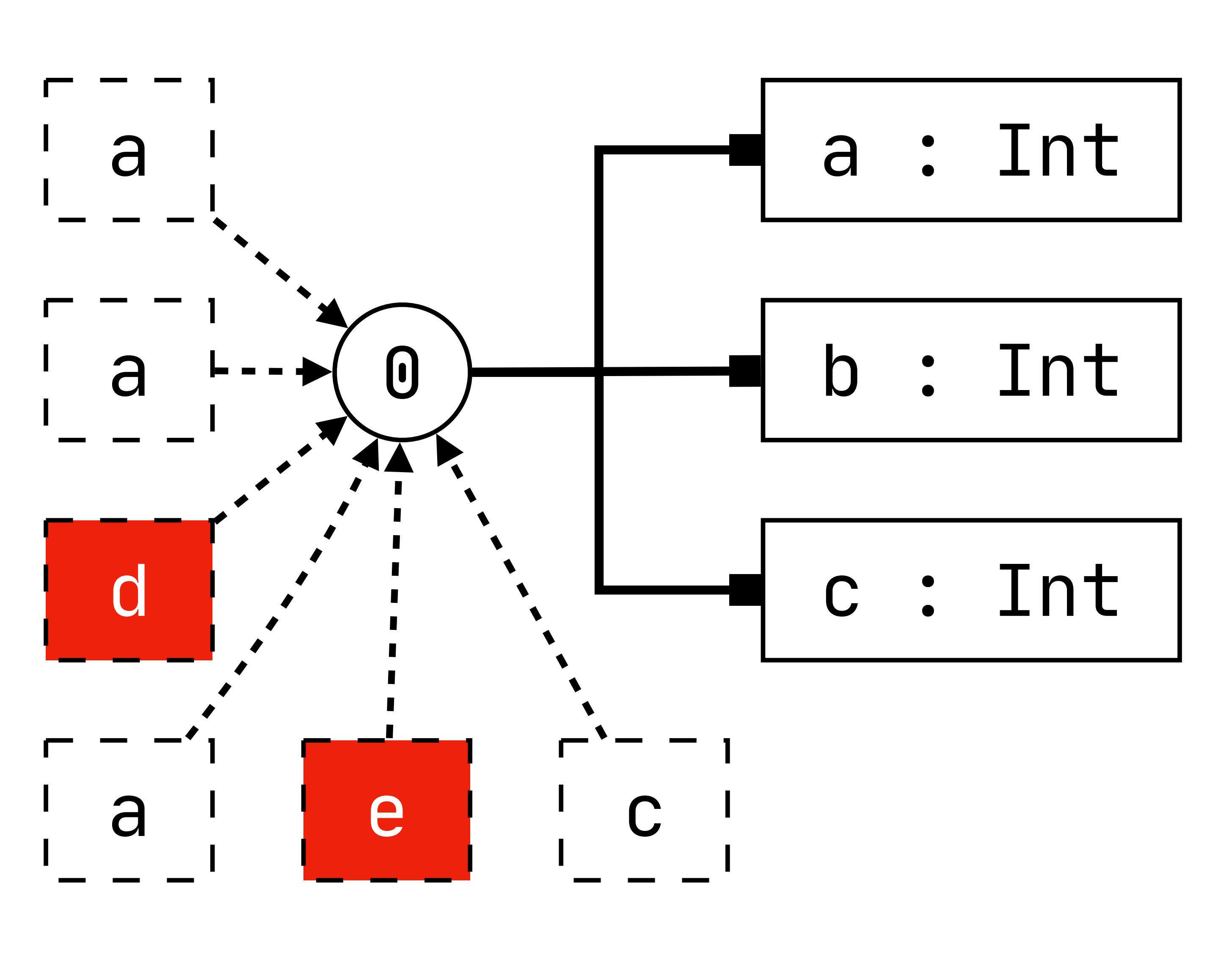
variable x is declared in scope s with type T

variable x in scope s resolves to list of declarations

variable x in scope s resolves to declaration x' with type T

```
def a = 0
def b = a + 1
def c = a + d
> a + e + c
```

undefined variable



`resolveVar` returns empty list of declarations

Duplicate Definition

```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE
  resolveVar : scope * ID → list((path * (ID * TYPE)))
  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s.

  resolveVar(s, x) = ps :-
    query var
      filter e and { x' :- x' == x }
         min and true
          in s |-> ps.

  typeOfVar(s, x) = T :- {x'}
    resolveVar(s, x) == [(_,(x', T))].
```

declaration relation

variable x is declared in scope s with type T

variable x in scope s resolves to list of declarations

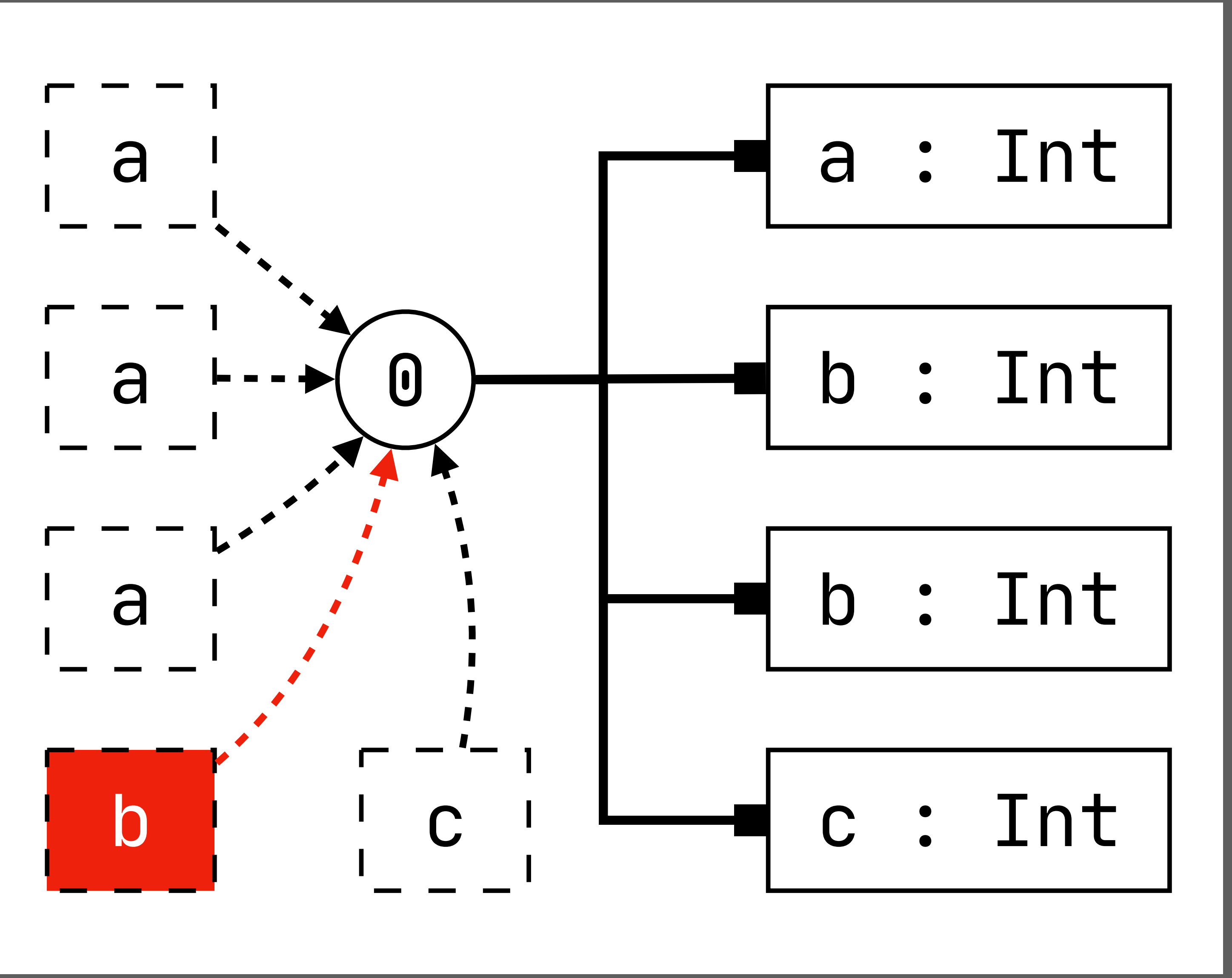
variable x in scope s resolves to declaration x' with type T

```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

what we want

```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

what we get



Duplicate Definition: Permissive Resolution

```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE
  resolveVar : scope * ID → list((path * (ID * TYPE)))
  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s,
    resolveVar(s, x) == [(_, (_, _))]
      | error $[Duplicate definition of variable [x]].


  resolveVar(s, x) = ps :-
    query var
      filter e and { x' :- x' == x }
         min and true
          in s ↦ ps.

  typeOfVar(s, x) = T :- {x'}
    resolveVar(s, x) == [(_,(x', T))|_]
      | error $[Variable [x] not defined].
```

declaration relation

variable x is declared in scope s with type T

there should only be one declaration

variable x in scope s resolves to list of declarations

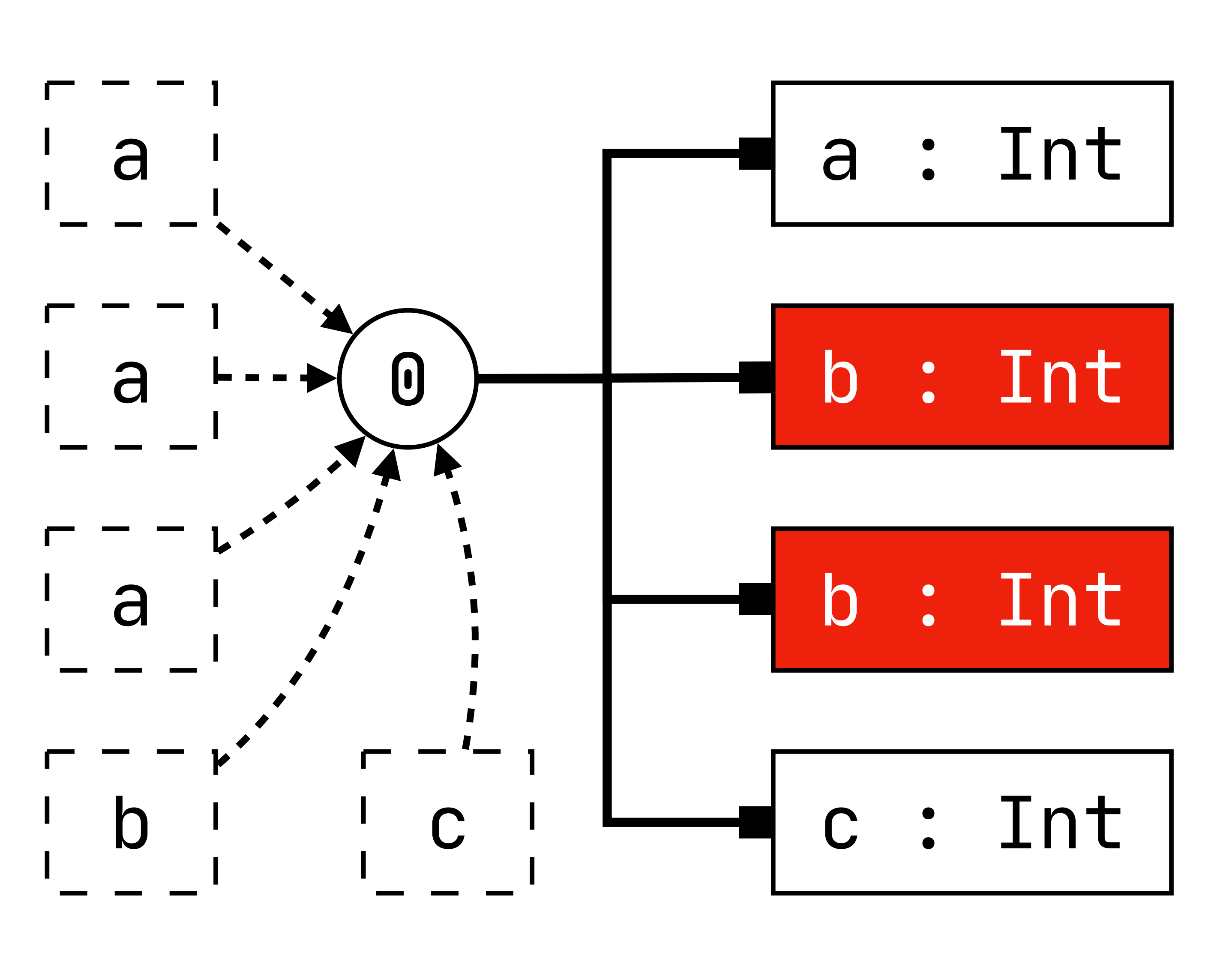
variable x in scope s resolves to declaration x' with type T

there should be at least one matching declaration

```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

duplicate definition



Reference and Type Attributes

```
signature
  relations
    var : ID → TYPE
rules

  declareVar : scope * ID * TYPE
  resolveVar : scope * ID → list((path * (ID * TYPE)))
  typeOfVar  : scope * ID → TYPE

  declareVar(s, x, T) :-
    !var[x, T] in s,
    resolveVar(s, x) == [(_, (_, _))]
      | error $[Duplicate definition of variable [x]],
    @x.type := T.

  resolveVar(s, x) = ps :-
    query var
      filter e and { x' :- x' == x }
         min and true
          in s ↦ ps.

  typeOfVar(s, x) = T :- {x'}
    resolveVar(s, x) == [(_,(x', T))|_]
      | error $[Variable [x] not defined],
    @x.ref := x'.
```

declaration relation

variable x is declared in scope s with type T

there should only be one declaration

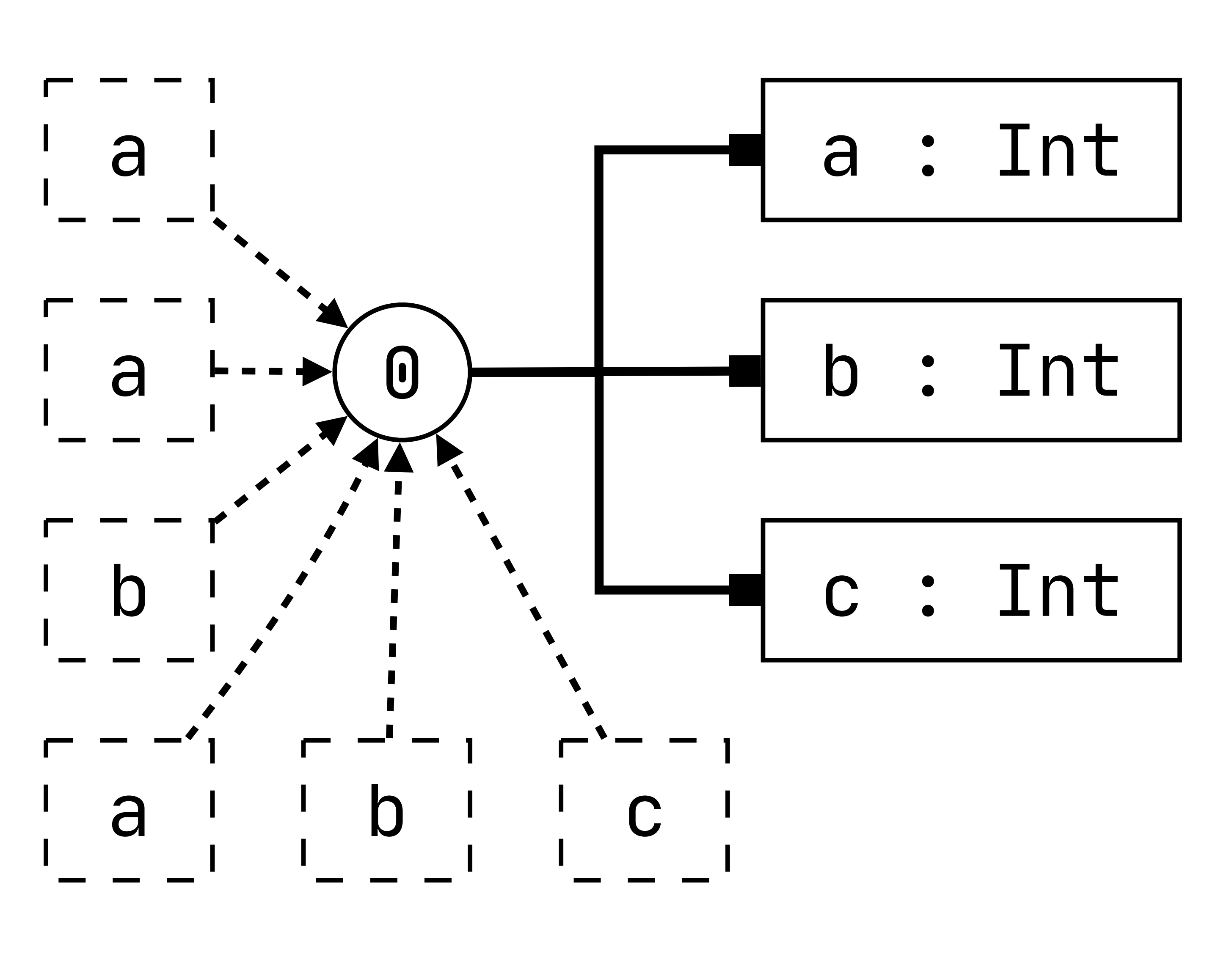
variable x in scope s resolves to list of declarations

variable x in scope s resolves to declaration x' with type T

there should be at least one matching declaration

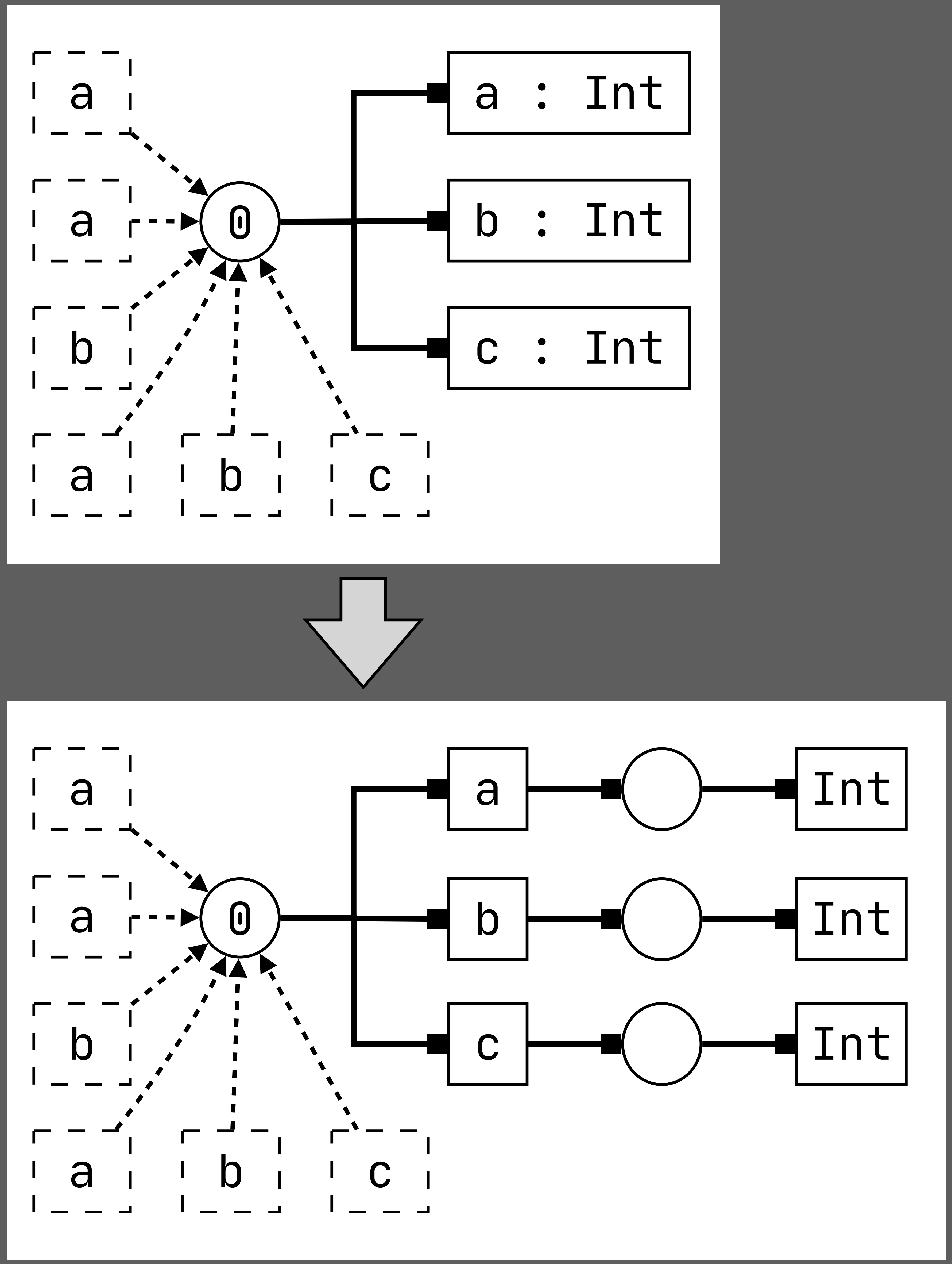
```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference



Type Annotation Indirection (for Parallel Type Checking)

```
signature
  relations
    var    : ID → scope
    typeOf : → TYPE

rules

  declareVar(s, x, T) :-
    !var[x, withType(T)] in s.

  typeOfVar(s, x) = typeOf(T) :- {x'}
    resolveVar(s, x) == [(_,(x', T))].

  typeOf : scope → TYPE

  typeOf(s) = T :-
    query typeOf
      filter e and true
      min /* */ and true
      in s ⟼ [(_, T)].

  withType : TYPE → scope

  withType(T) = s :-
    new s, !typeOf[T] in s.
```



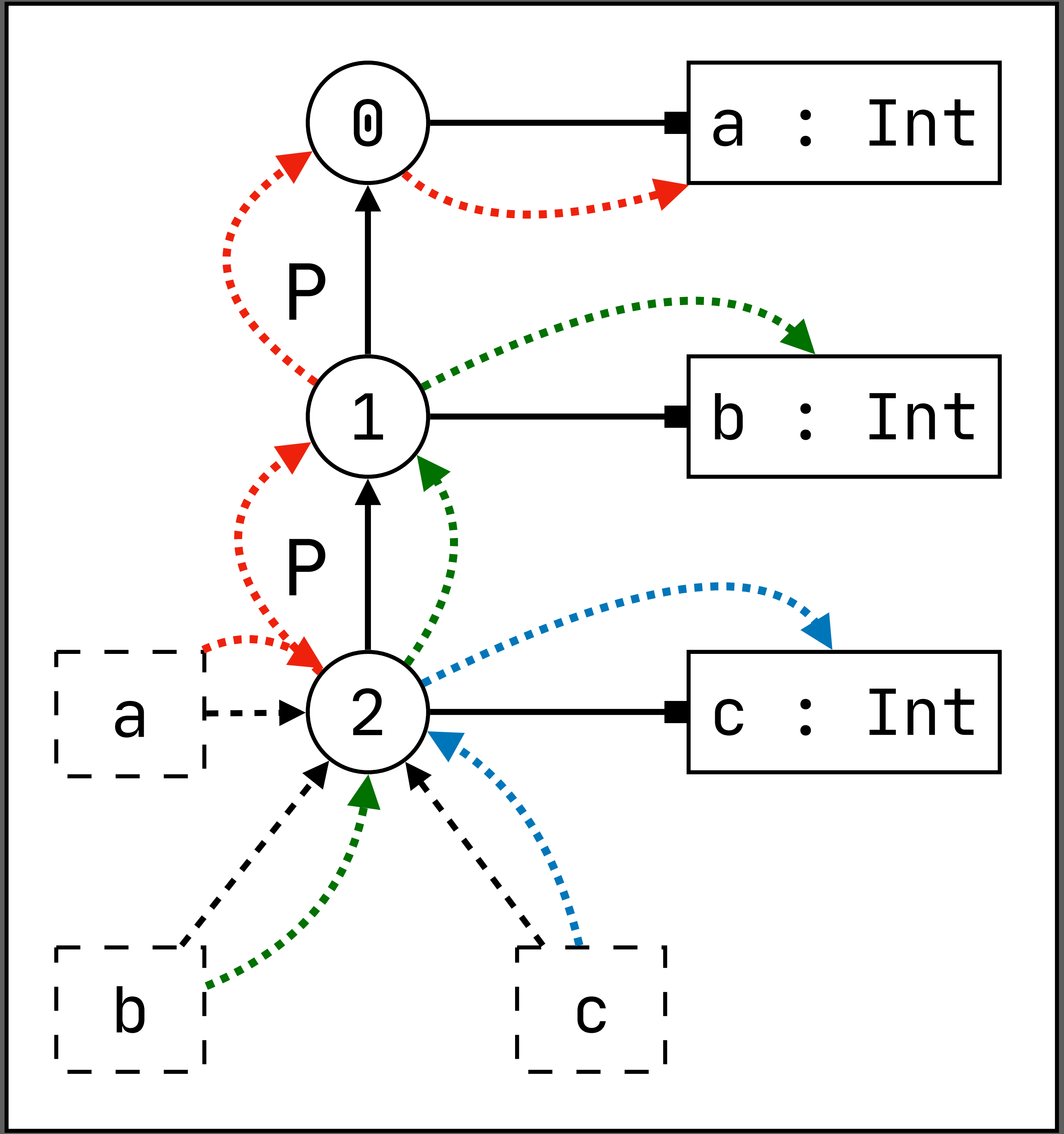
How about shadowing?

Lexical Scope

Labeled Scope Edges: What Scopes are Reachable?

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```



New Scope and Scope Edge Constraints

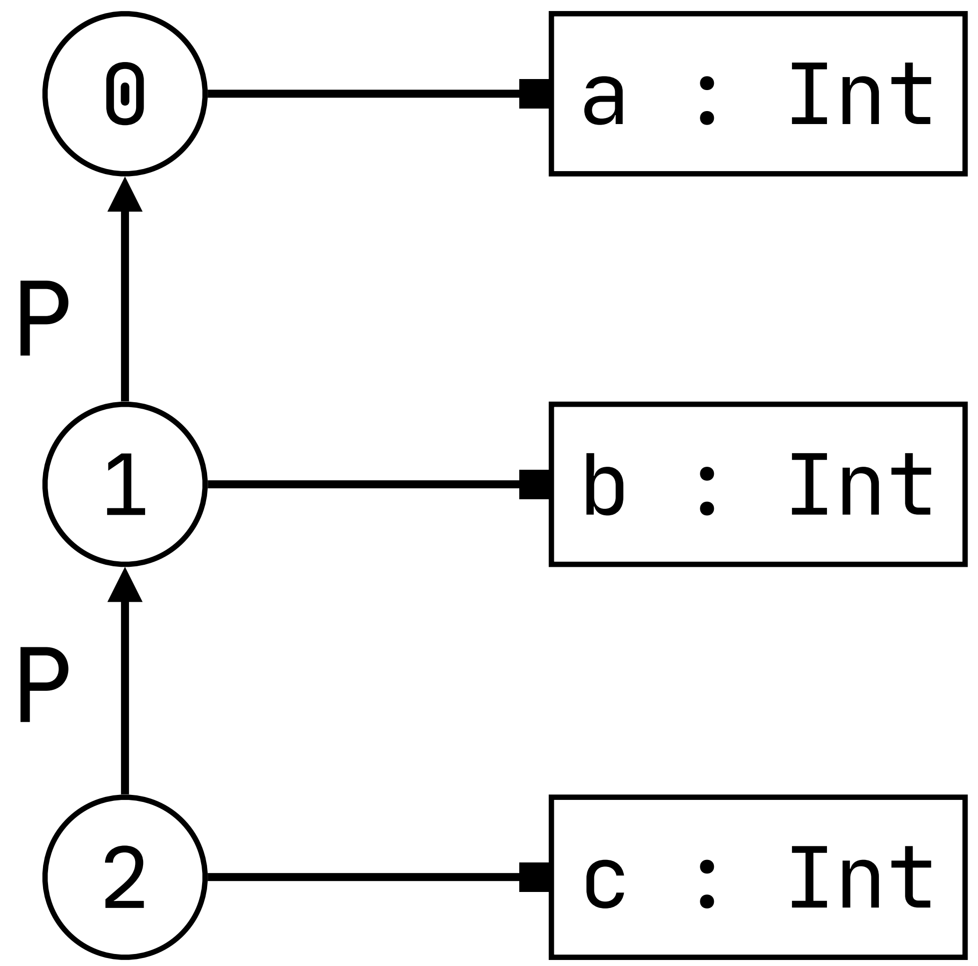
```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P-> s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

scope edge



Path Wellformedness: Reachability

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

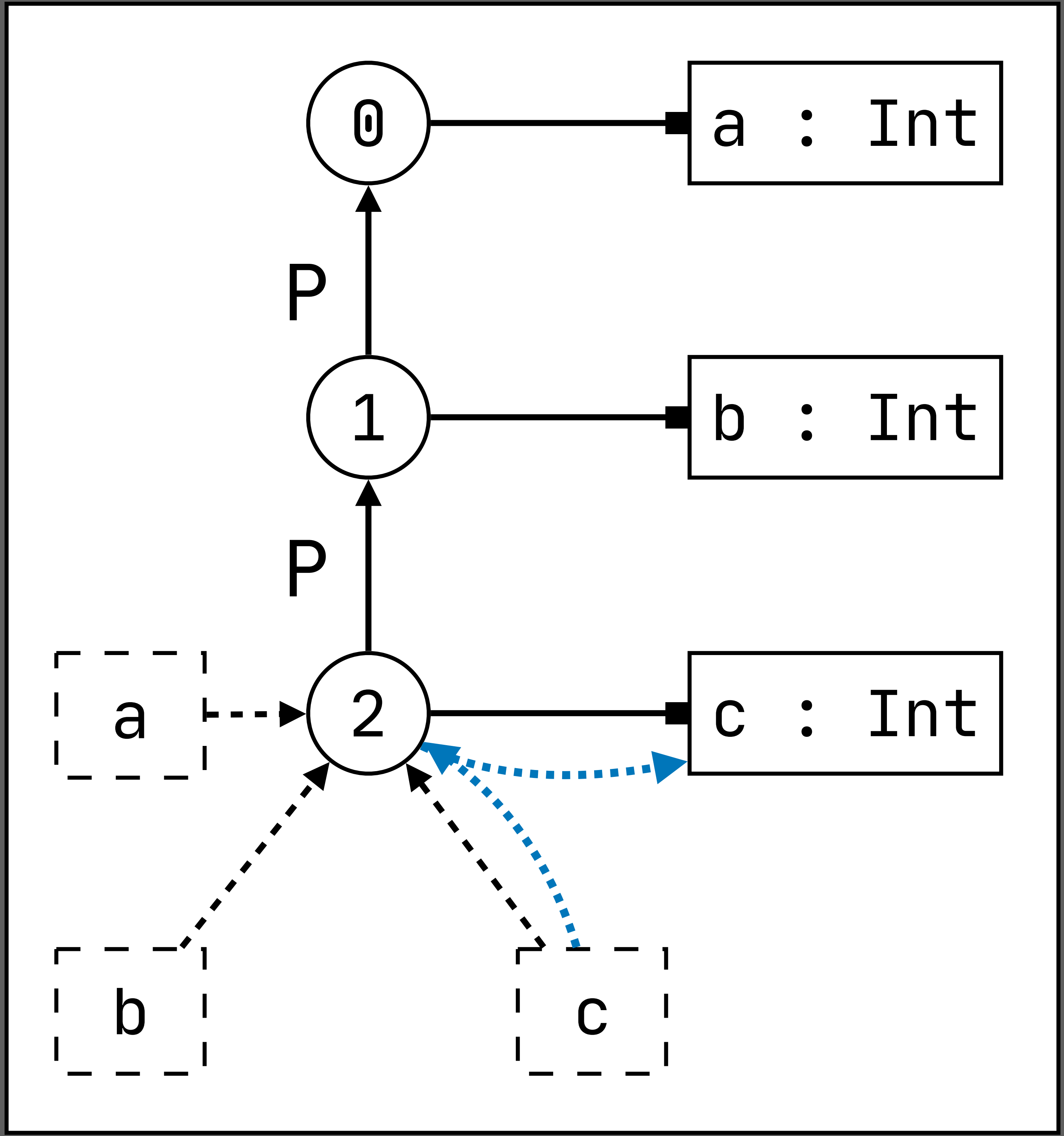
```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

scope edge




```
  resolveVar(s, x) = ps :-
    query var
      filter e and { x' :- x' == x }
        min /* */ and true
         in s ⟼ ps.
```

empty path e only allows resolution in `this' scope

Path Wellformedness: Reachability

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
  name-resolution
    labels P
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

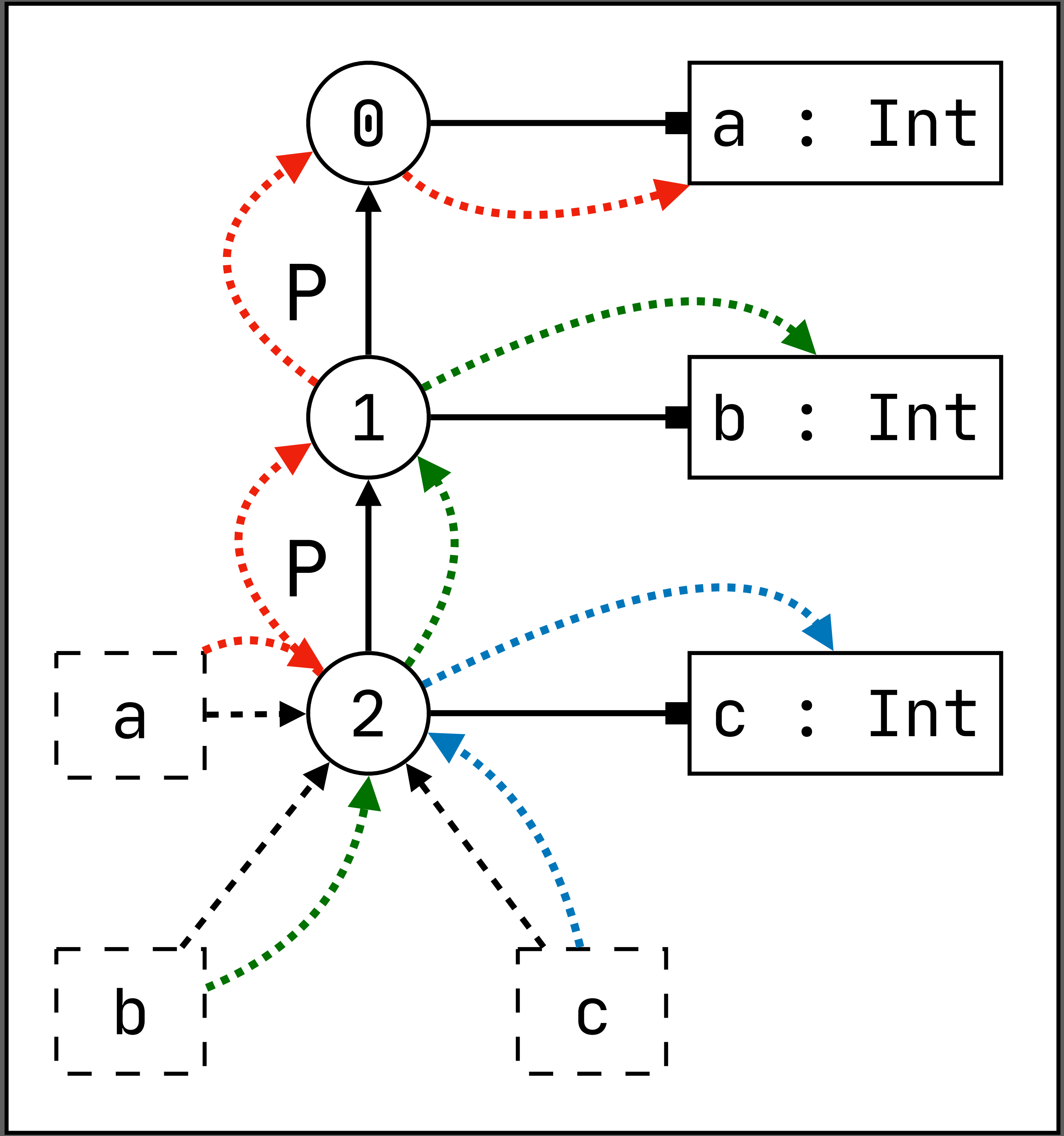
new scope

scope edge

```
  resolveVar(s, x) = ps :-
    query var
      filter P* and { x' :- x' == x }
         min /* */ and true
          in s ↦ ps.
```

path *P*∗ allows resolution through zero or more *P* edges

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```



Duplicate Definitions Revisited

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
  name-resolution
    labels P
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

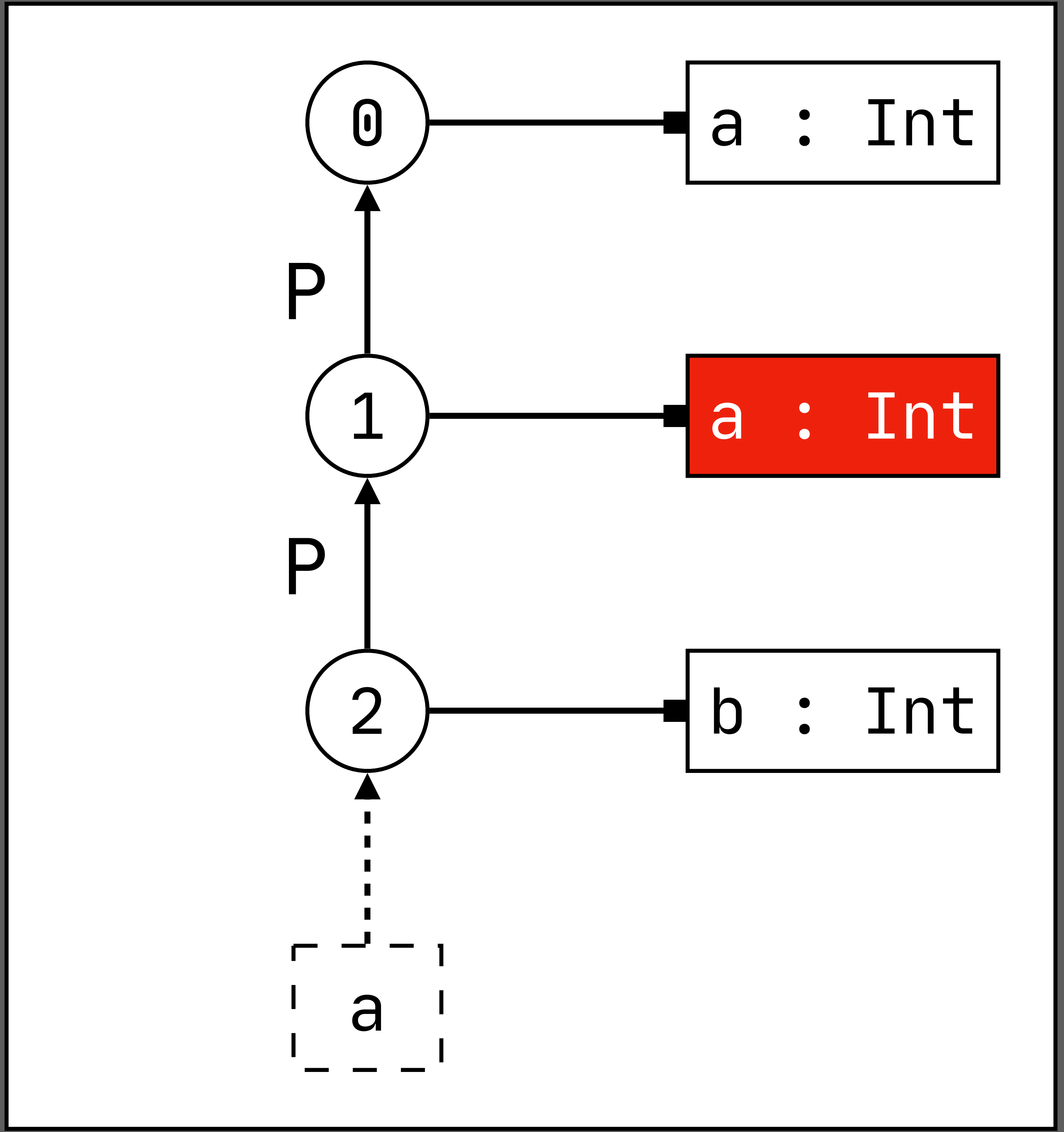
scope edge

```
  resolveVar(s, x) = ps :-
    query var
      filter P* and { x' :- x' == x }
         min /* */ and true
          in s ↦ ps.
```

path *P** allows resolution through zero or more *P* edges

```
let a = 1 in
let a = 2 in
let b = 3 in
  a
```

duplicate definition



Duplicate Definitions Revisited

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
  name-resolution
    labels P
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P-> s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

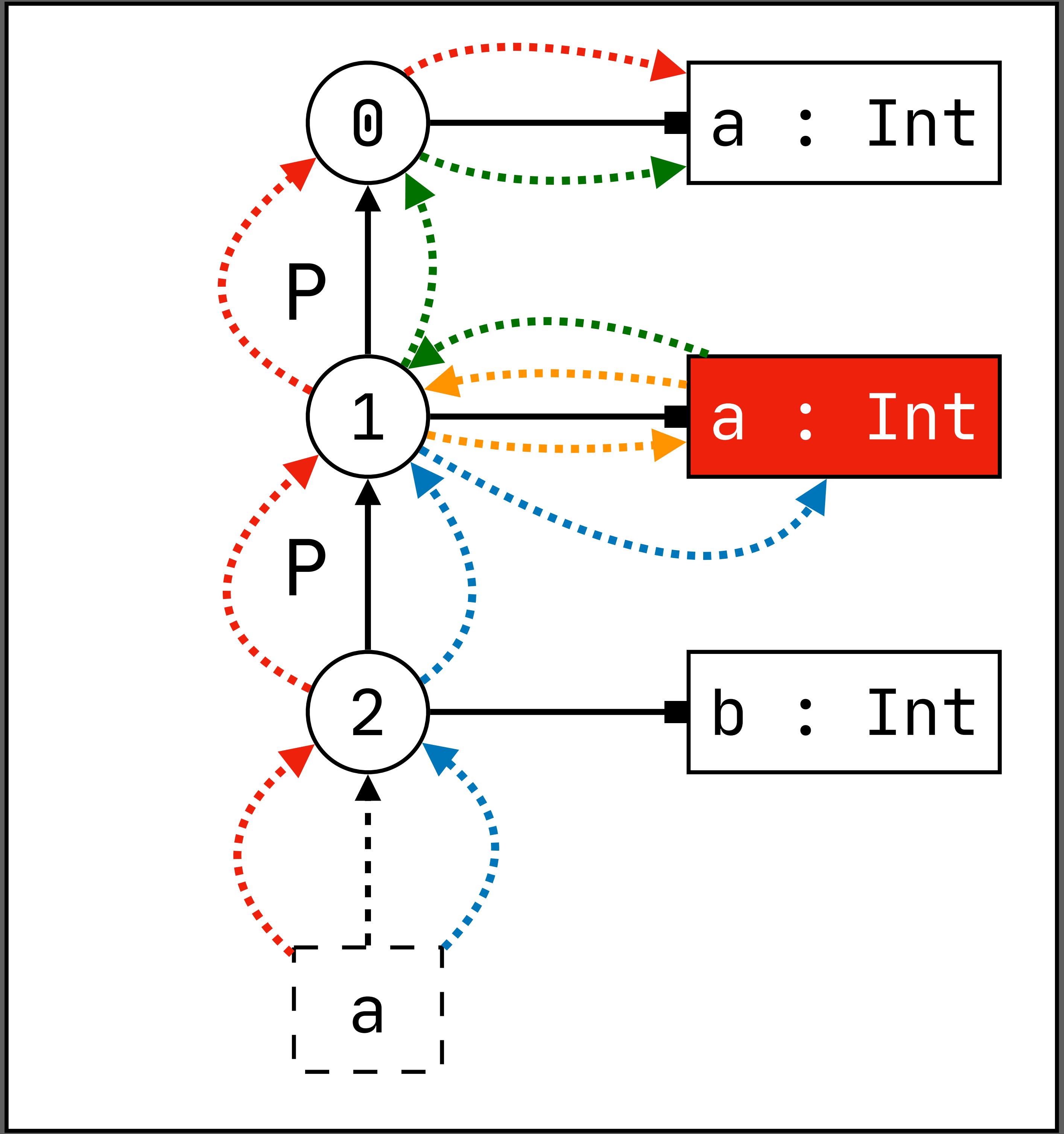
scope edge

```
  resolveVar(s, x) = ps :-
    query var
      filter P* and { x' :- x' == x }
         min /* */ and true
          in s |-> ps.
```

path *P** allows resolution through zero or more *P* edges

```
let a = 1 in
let a = 2 in
let b = 3 in
  a
```

duplicate definition



Path Specificity: Visibility (Shadowing)

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
  name-resolution
    labels P
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P-> s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

scope edge

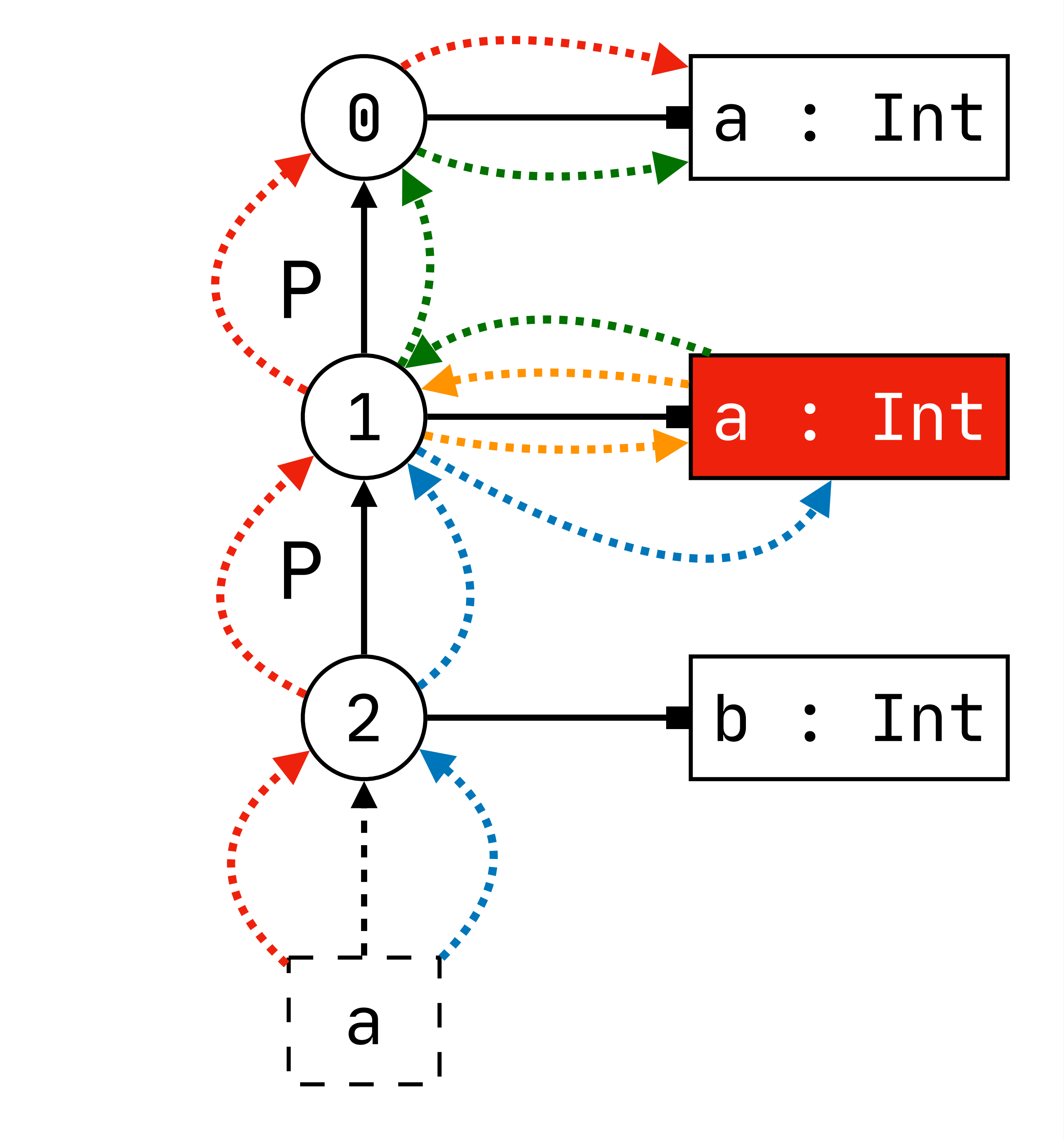
```
  resolveVar(s, x) = ps :-
    query var
      filter P* and { x' :- x' == x }
         min /* */ and true
          in s |-> ps.
```

path *P** allows resolution through zero or more *P* edges

prefer local scope ($) over parent scope (*P*)

```
let a = 1 in
let a = 2 in
let b = 3 in
  a
```

duplicate definition



prefer blue path over red path

prefer orange path over green path

Path Specificity: Visibility (Shadowing)

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
  name-resolution
    labels P
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let, s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

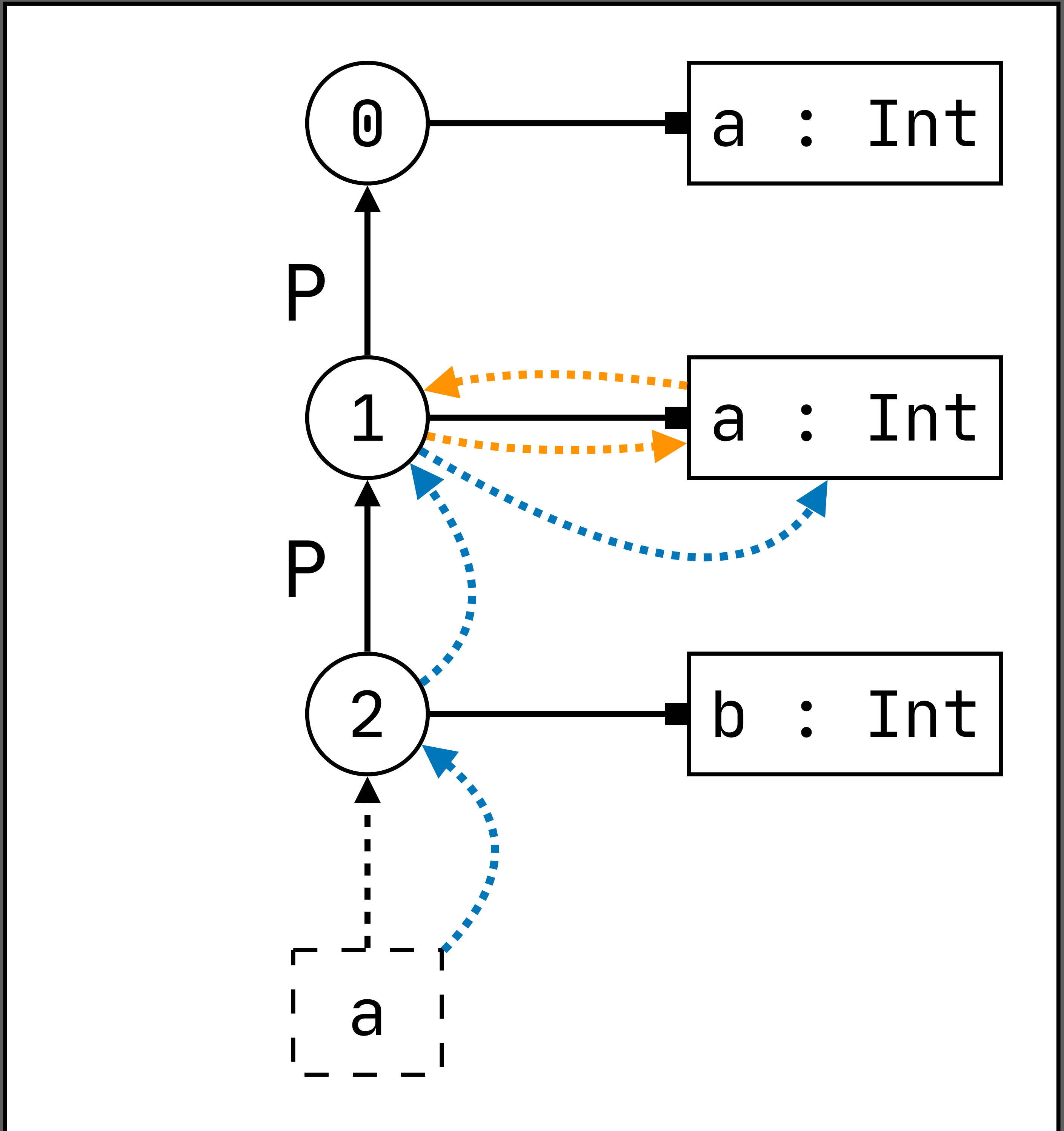
scope edge

```
  resolveVar(s, x) = ps :-
    query var
      filter P* and { x' :- x' == x }
         min $ < P and true
          in s ⟼ ps.
```

path P* allows resolution through zero or more P edges

prefer local scope ($) over parent scope (P)

```
let a = 1 in
let a = 2 in
let b = 3 in
  a
```



Statix Queries

```
query RELATION filter REGEX and MATCH min LABELORD and SHADOW in SCOPE ⟼ RESULT
```

`RELATION`

– the relation we are querying

`filter`

– filter applied to individual paths to rule out ‘non-wellformed’ paths

`min`

– a filter applied to pairs of paths to select the ‘minimal’ paths from a set of paths

`SCOPE`

– the source scope of the query

`RESULT`

– the list of query results

Statix Queries

`query RELATION filter REGEX and MATCH min LABELORD and SHADOW in SCOPE ⟼ RESULT`

RELATION

- Can be a custom relation name, `decl`, or `()` which only looks at the reachable scopes

REGEX

- Specifies path well-formedness using a regular expression (e.g. `P* I*`) over edge labels

MATCH

- Specifies which data in the relation to match
- To match on a name x, the match is an anonymous rule `{x' :- x' == x}` which is tested against all reachable declarations

LABELORD

- Determines the edge label order using inequalities (e.g. `$ < P, I < P`) over edge labels

SHADOW

- Determines which declarations shadow each other
- `true`: all declarations shadow each other and we only get the declarations reached via the shortest path
- `false`: none of the declarations shadow (which could be used to find all reachable declarations)
- Anonymous rule (e.g. `{x, x′ :- x' == x})`: only shadow between things with the same name

How about non-lexical bindings?

Scopes as Types
/
Modules

Modules: Scopes as Types

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

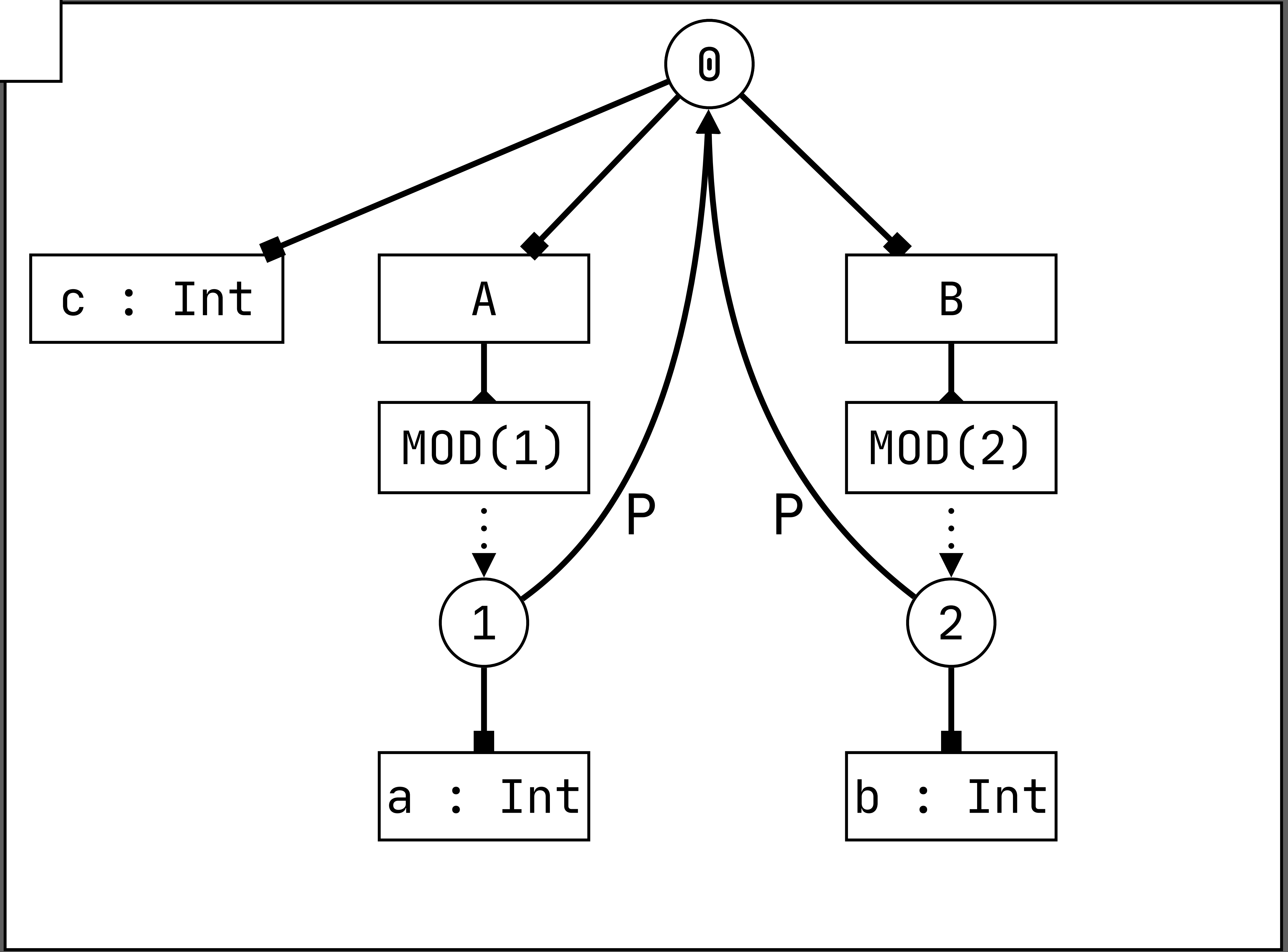
scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).
```

lexical scope

scope as type

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Resolving Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

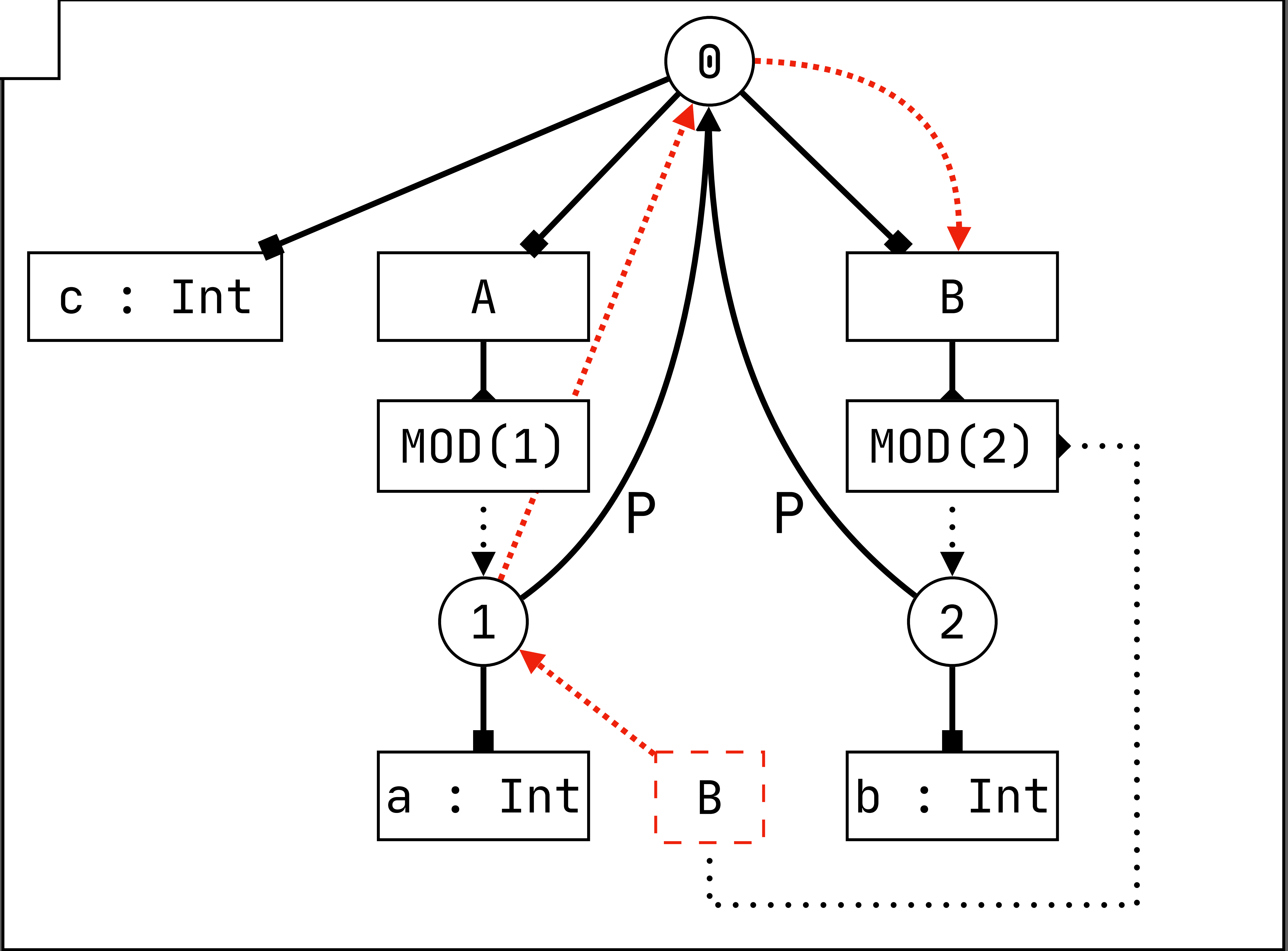
scope as type

resolve import

```
  resolveMod(s, x) = ps :-
    query mod
      filter P* and { x' :- x' == x }
        min $ < P and true
         in s ⟼ ps.
```

resolve module name

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Import Edge

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

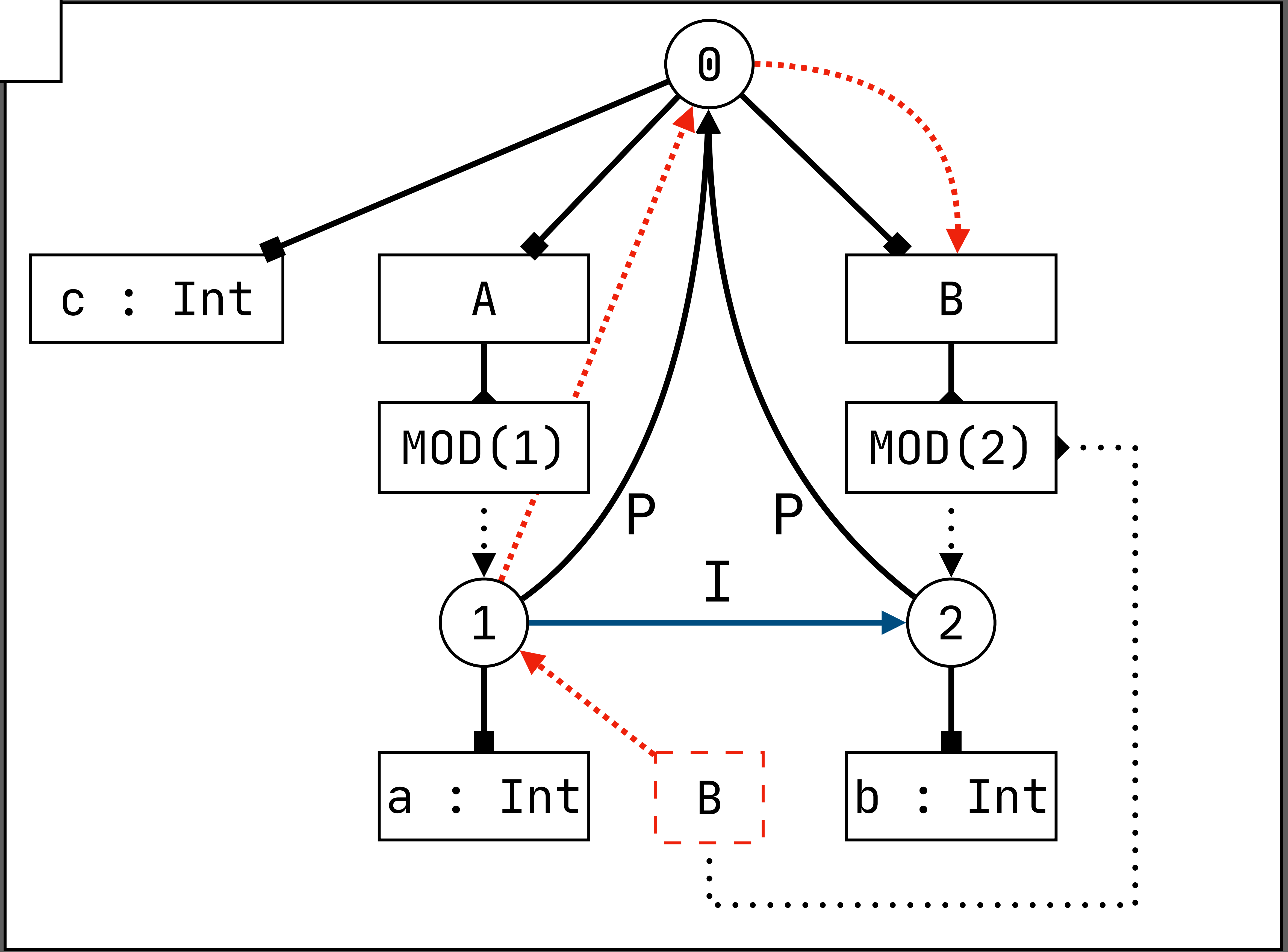
resolve import

import edge

```
  resolveMod(s, x) = ps :-
    query mod
      filter P* and { x' :- x' == x }
        min $ < P and true
          in s ⟼ ps.
```

resolve module name

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Resolving Variable through Import Edge

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

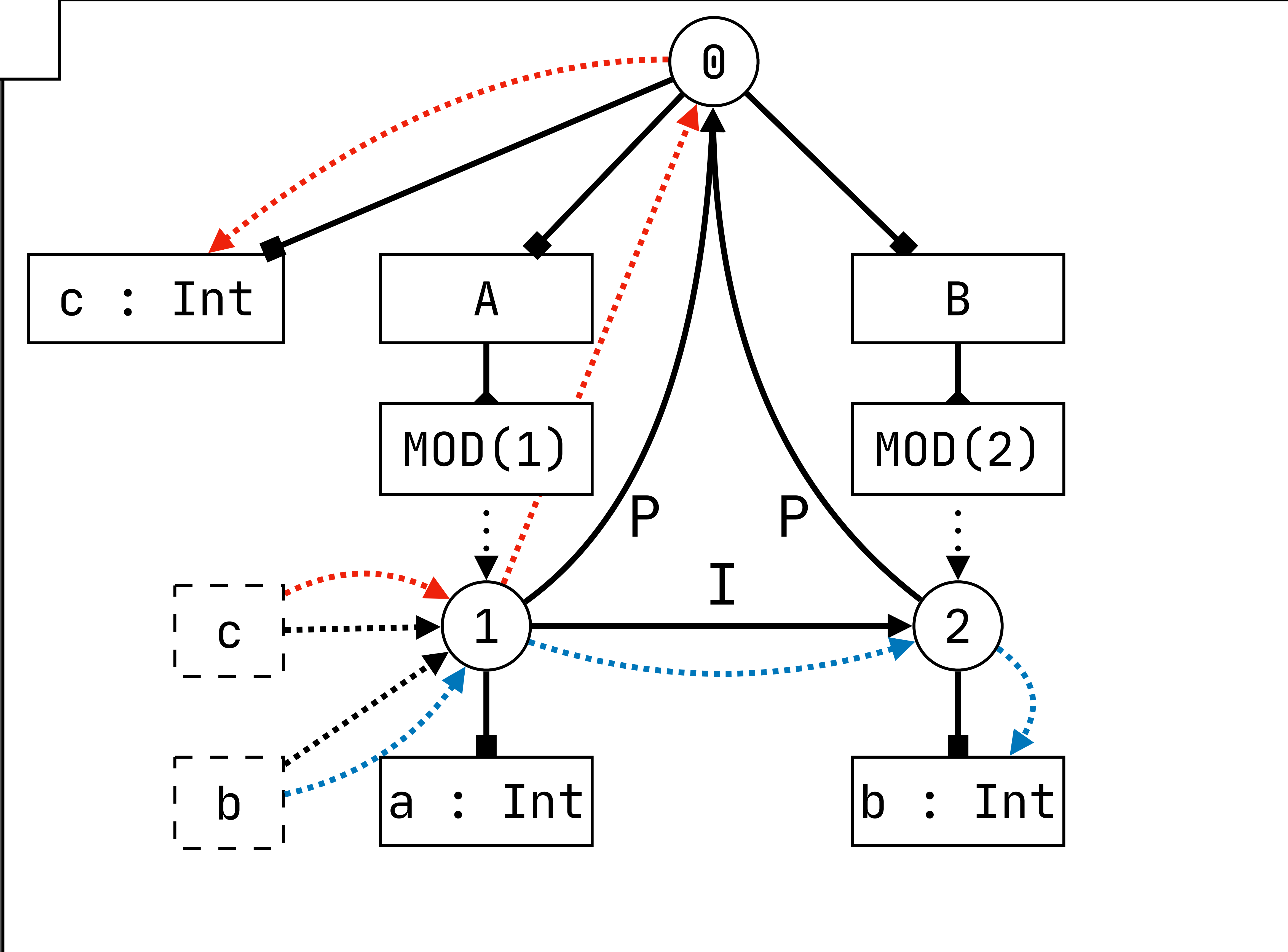
resolve import

import edge

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ⟼ ps.
```

resolve variable through import edges

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Import Shadows Parent (Lexical Context)

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

resolve import

import edge

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
          in s ↦ ps.
```

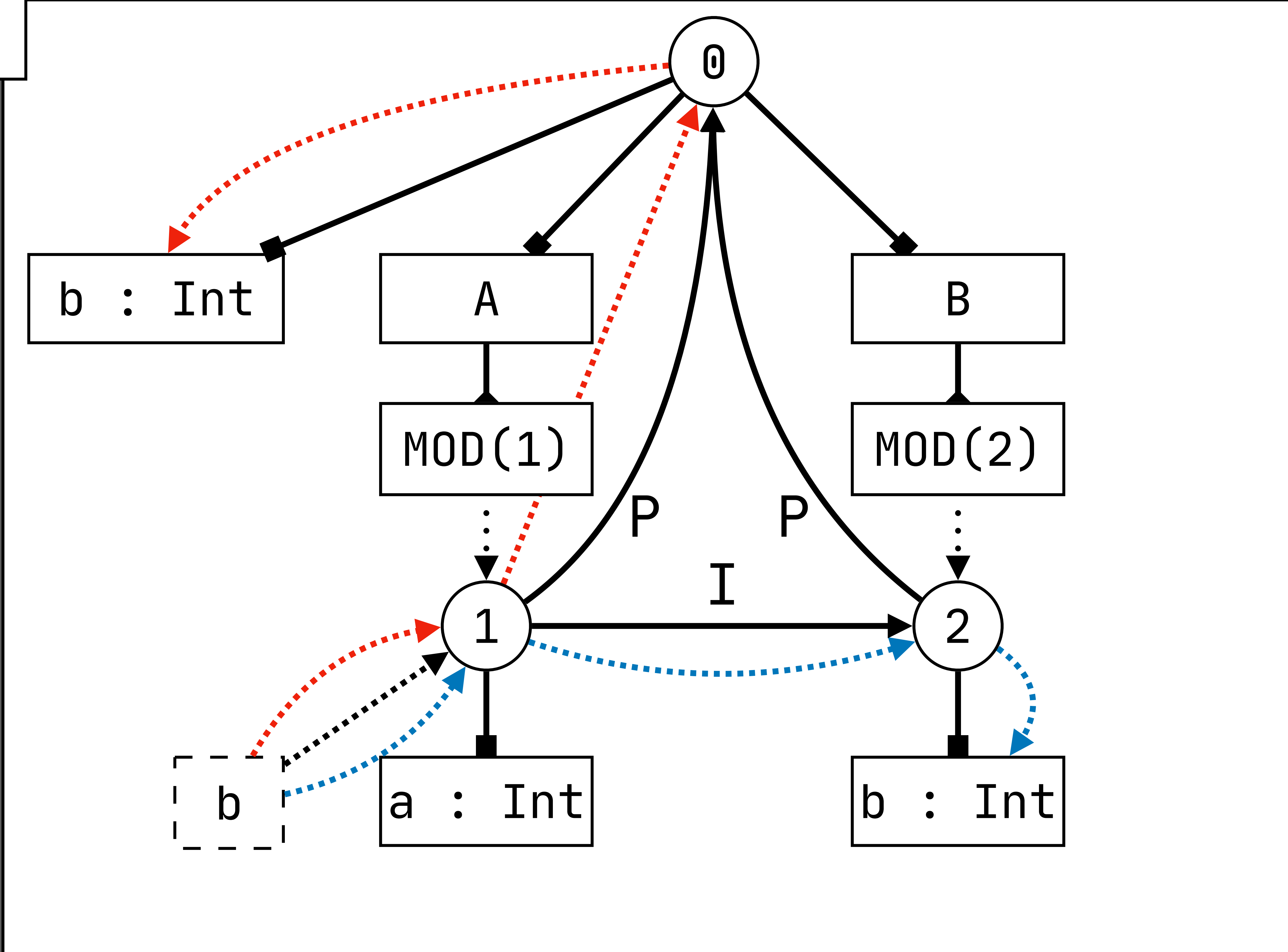
import after parent

prefer import

resolve variable through import edges

```
def b = 0
module A {
  import B
  def a = b
}
module B {
  def b = 2
}
```

prefer blue path over red path



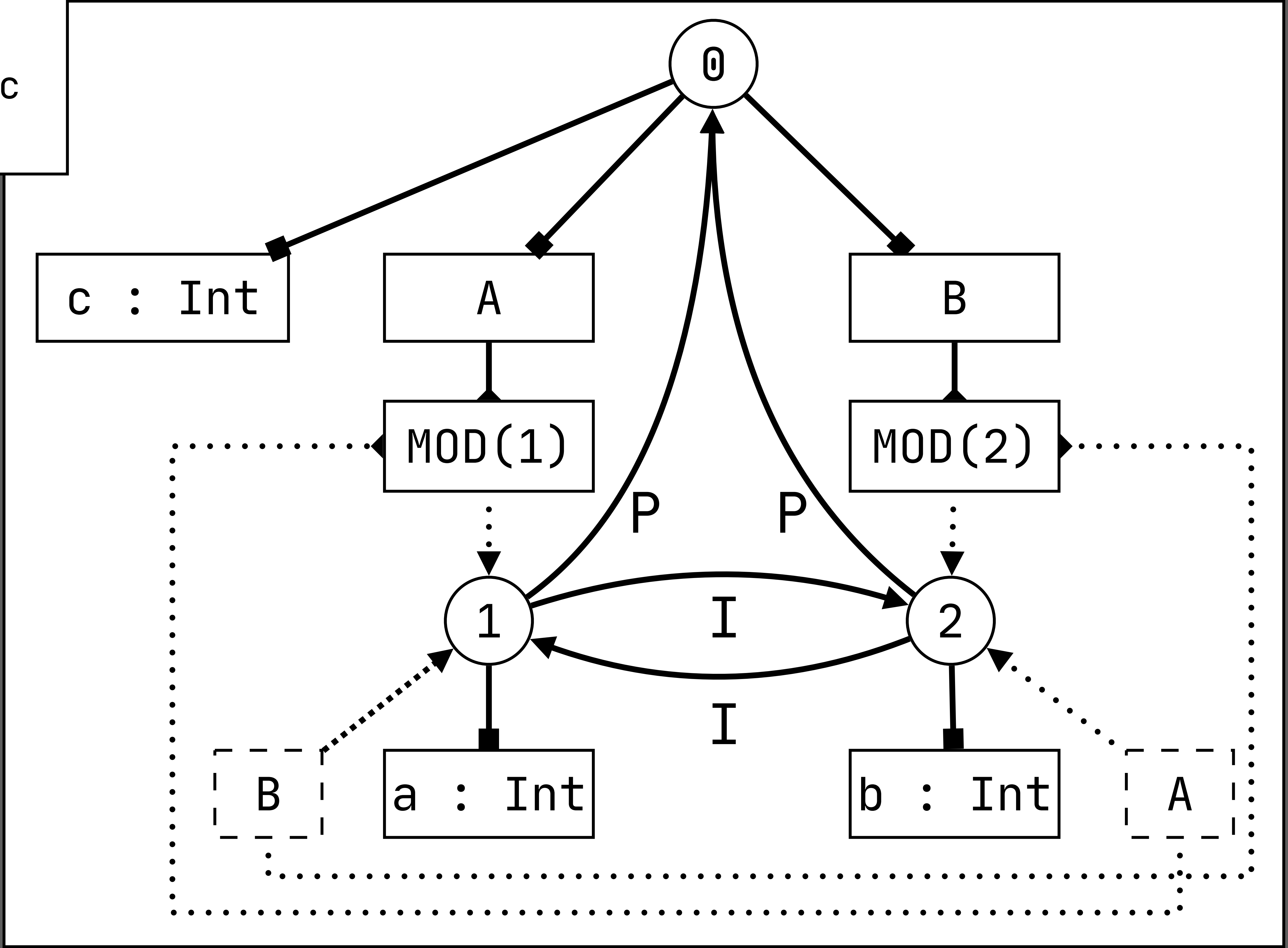
Mutually Recursive Imports

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ⟼ ps.
```

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```



Mutually Recursive Imports

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

scope as type

resolve import

import edge

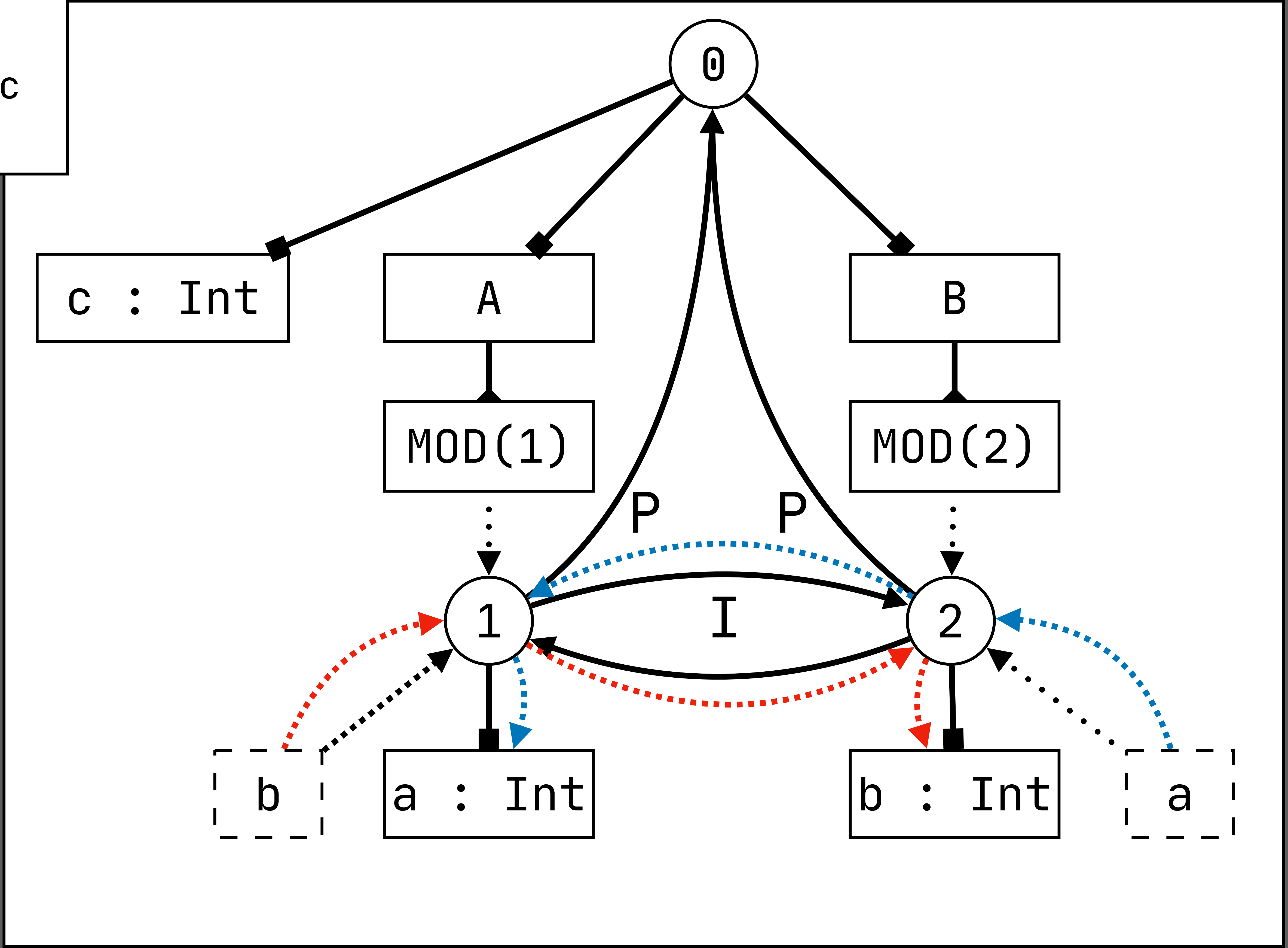
```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ↦ ps.
```

import after parent

prefer import

resolve variable through import edges

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```



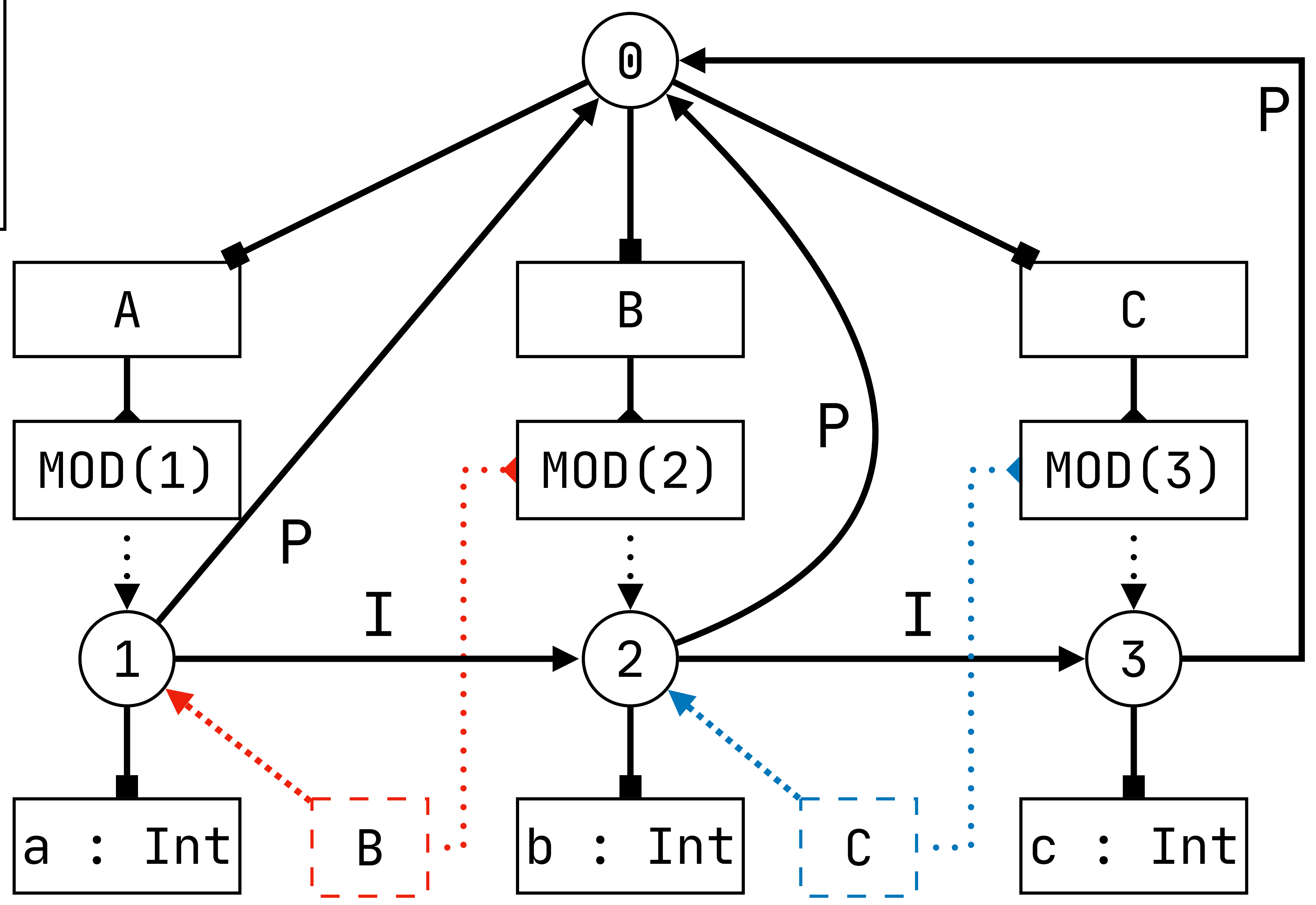
Transitive Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ↦→ ps.
```

```
module A {
  import B
  def a = b + c
}
module B {
  import C
  def b = c + 2
}
module C {
  def c = 1
}
```



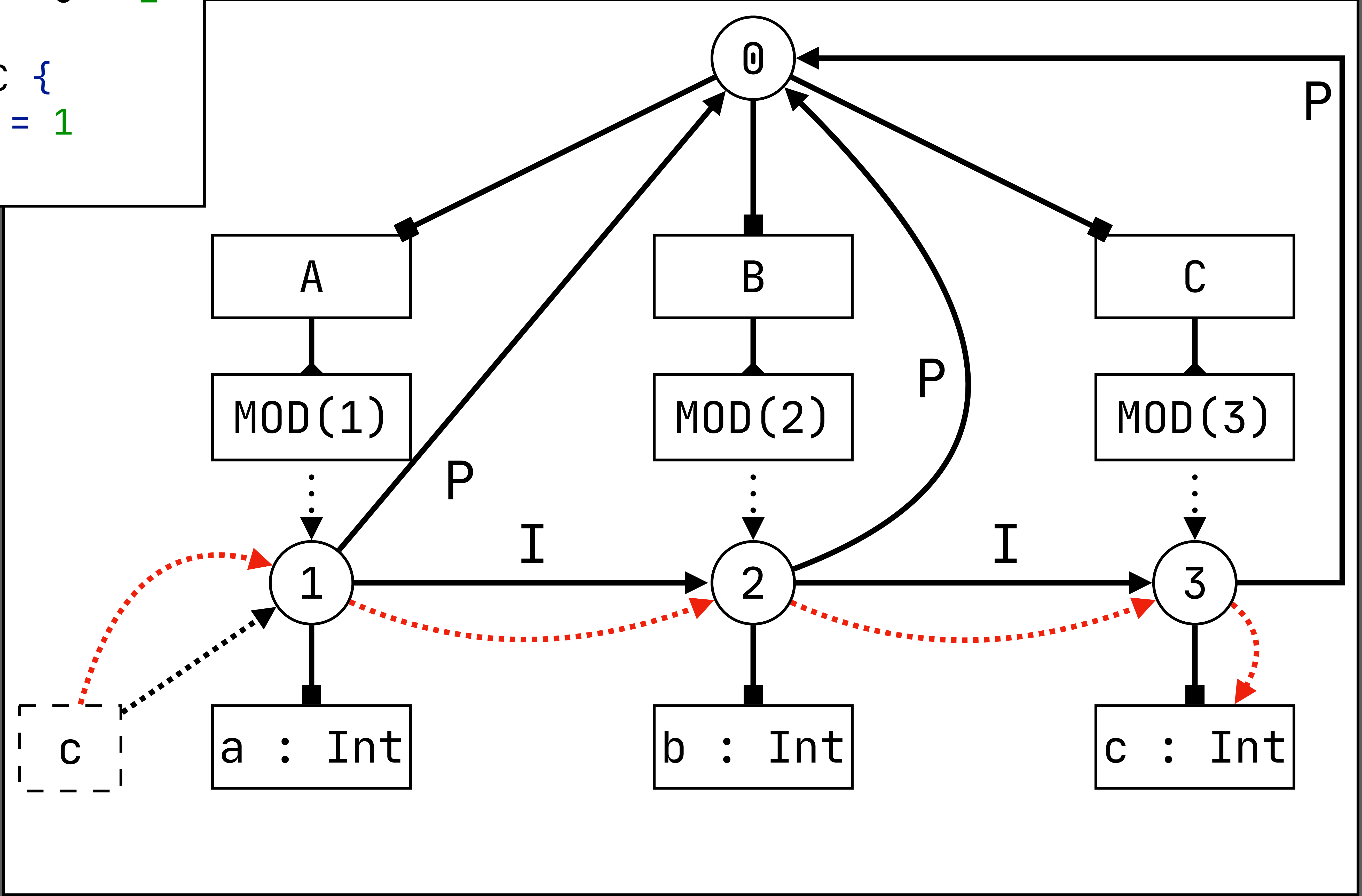
Transitive Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ⟼ ps.
```

```
module A {
  import B
  def a = b + c
}
module B {
  import C
  def b = c + 2
}
module C {
  def c = 1
}
```



Changing Query Outcomes

(is not allowed)

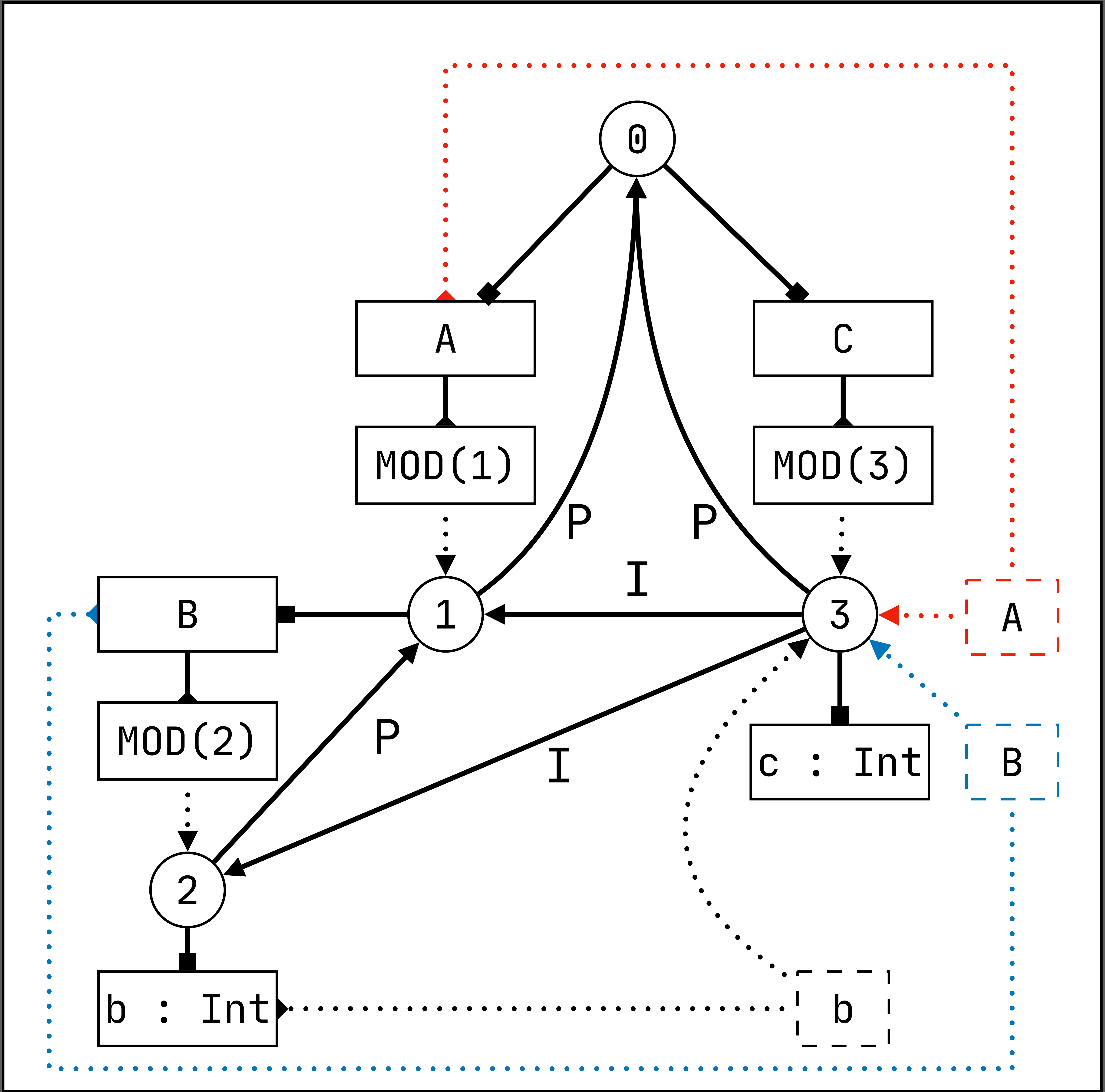
Nested Modules

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
module A {
  module B {
    def b = 1
  }
}

module C {
  import A
  import B
  def c = b
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P-> s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I-> s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s |-> ps.
```



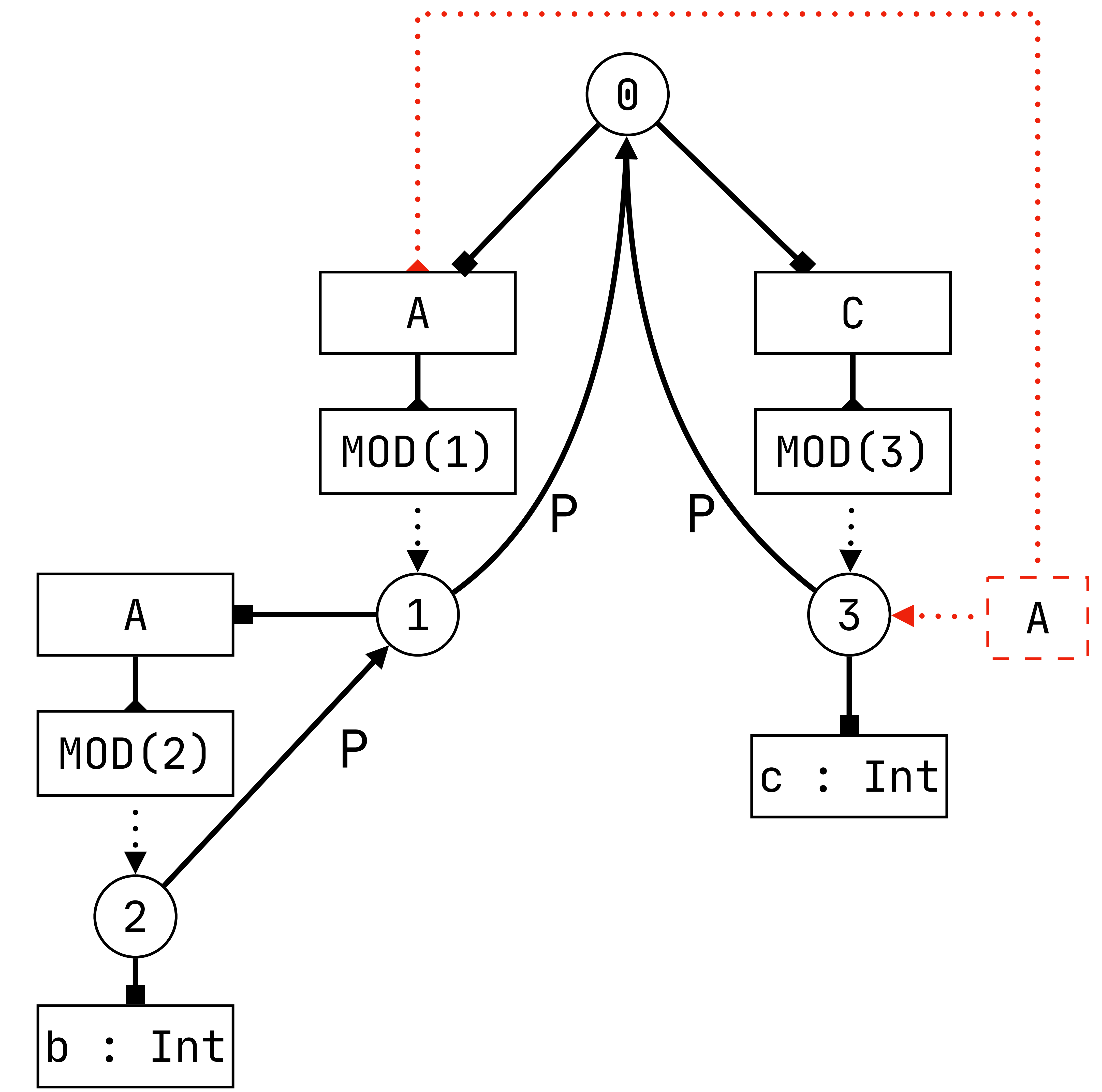
Changing Result of Query

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
module A {
  module A {
    def b = 1
  }
}

module C {
  import A
  import A
  def c = b
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ↦ ps.
```



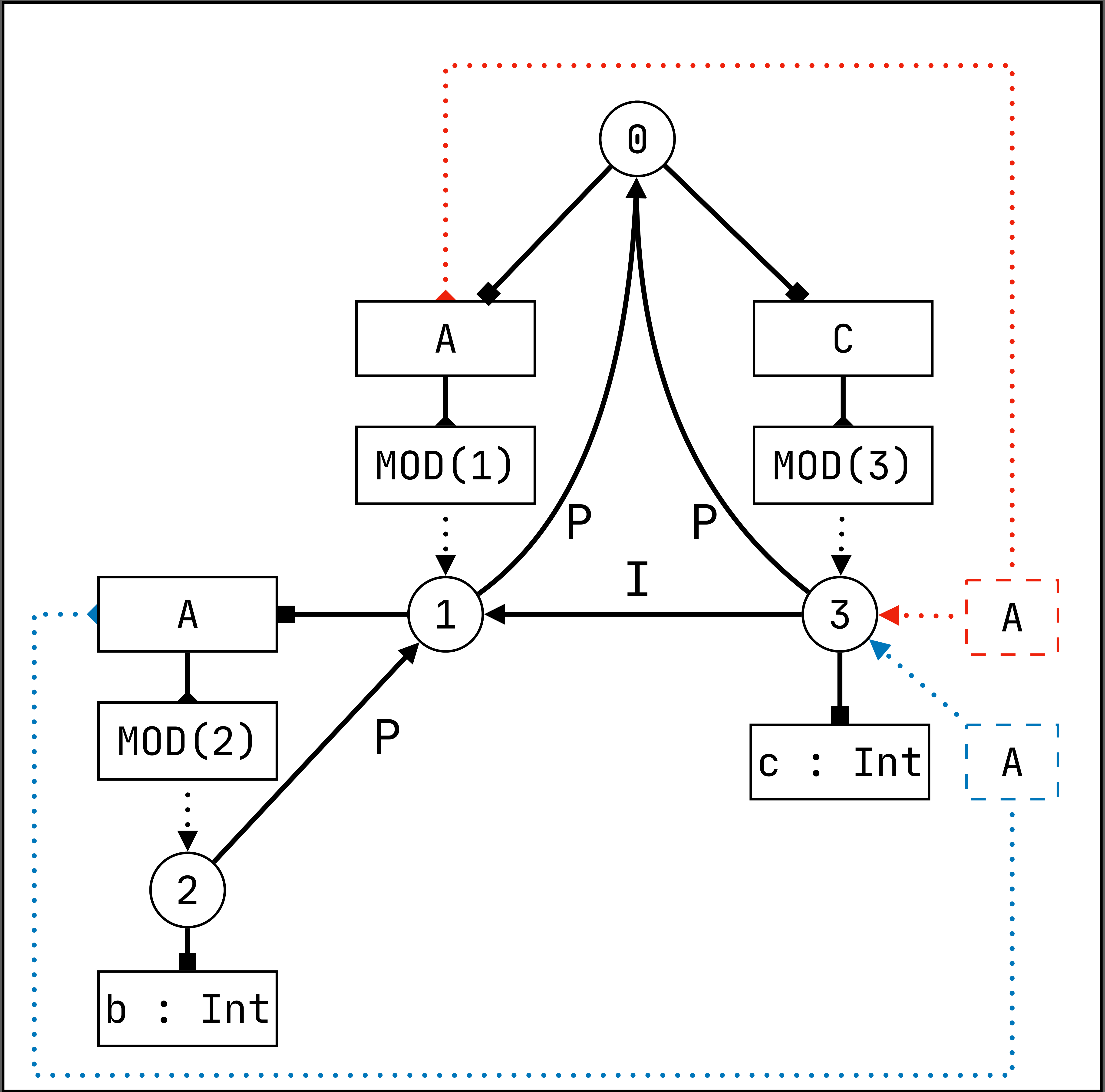
Changing Result of Query

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < I, I < P and true
         in s ⟼ ps.
```

```
module A {
  module A {
    def b = 1
  }
}

module C {
  import A
  import A
  def c = b
}
```



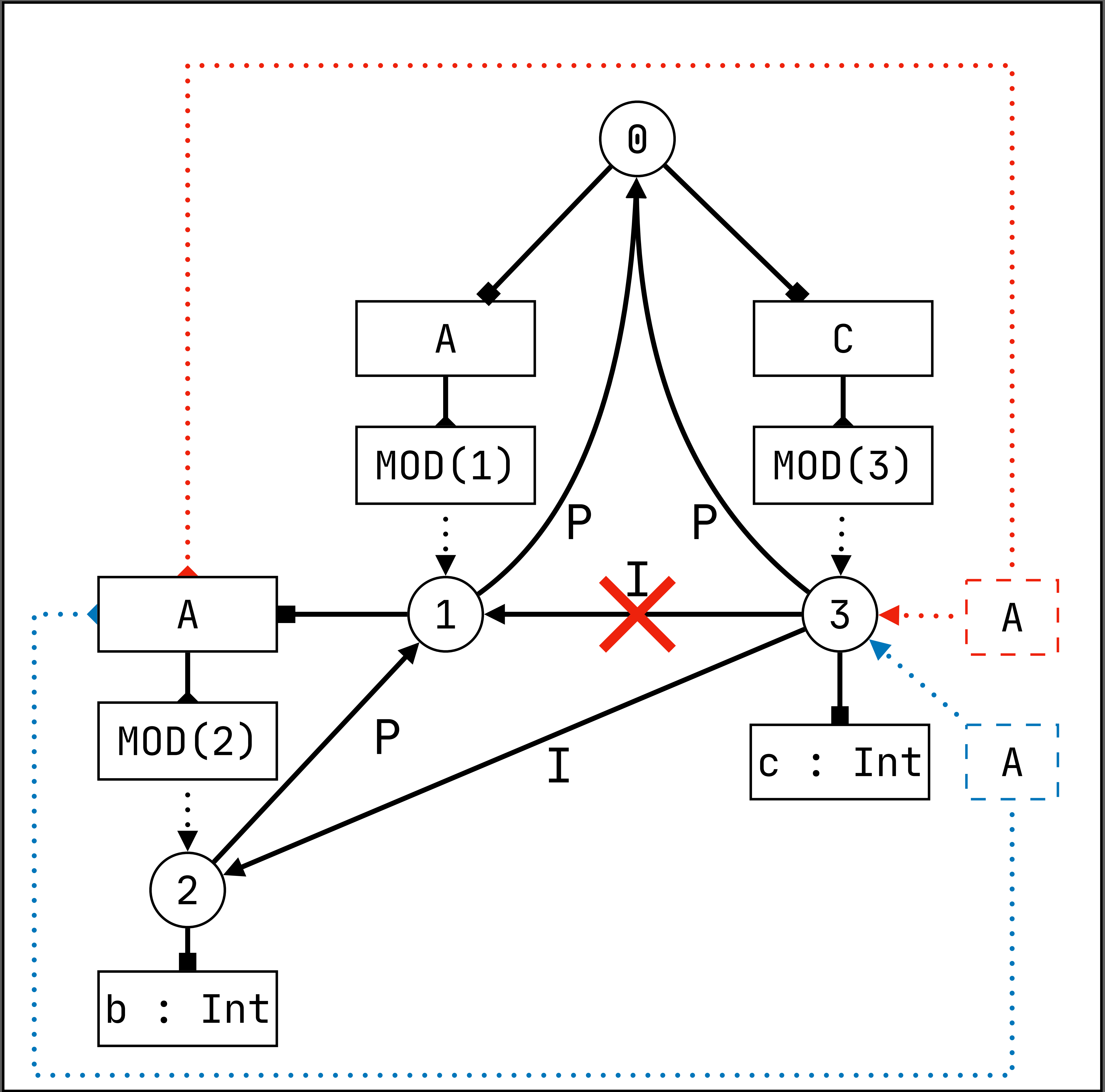
Changing Result of Query

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* I* and { x' :- x' == x }
        min $ < P, $ < P, I < P and true
         in s ↦ ps.
```

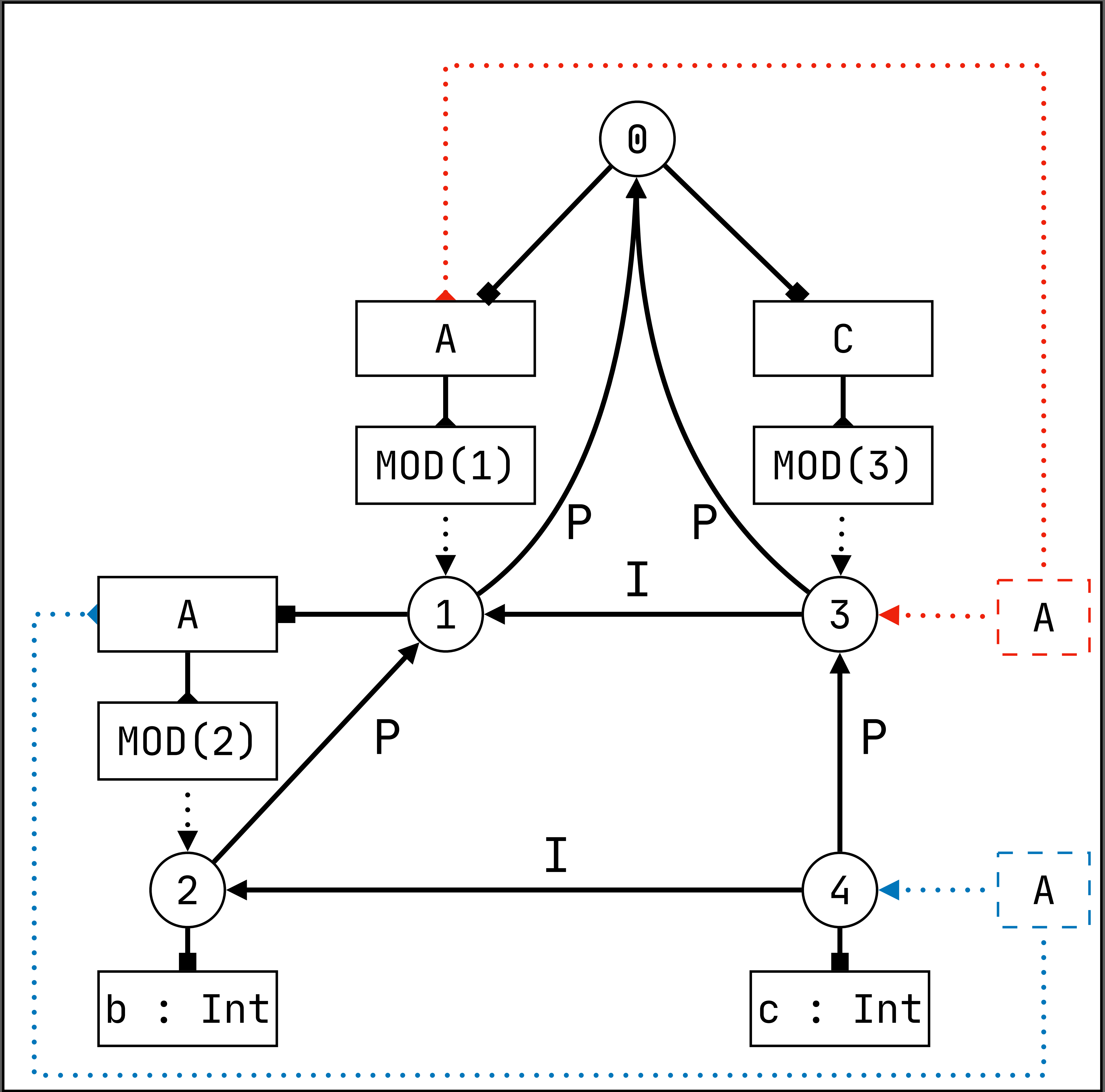
```
module A {
  module A {
    def b = 1
  }
}

module C {
  import A
  import A
  def c = b
}
```



Alternative Encoding: Scoped Imports

```
signature
  sorts DecGroups
  constructors
    MOD     : scope → TYPE
    Module  : ID * DecGroups → Decl
    Import  : ID → Decl
    ModRef  : ID * ID → Exp

    Decs    : list(Decl) → DecGroups
    Seq     : list(Decl) * DecGroups
              → DecGroups
```

```
module A {
  module A {
    def b = 1
  }
}

module C {
  import A;
  import A
  def c = b
}
```

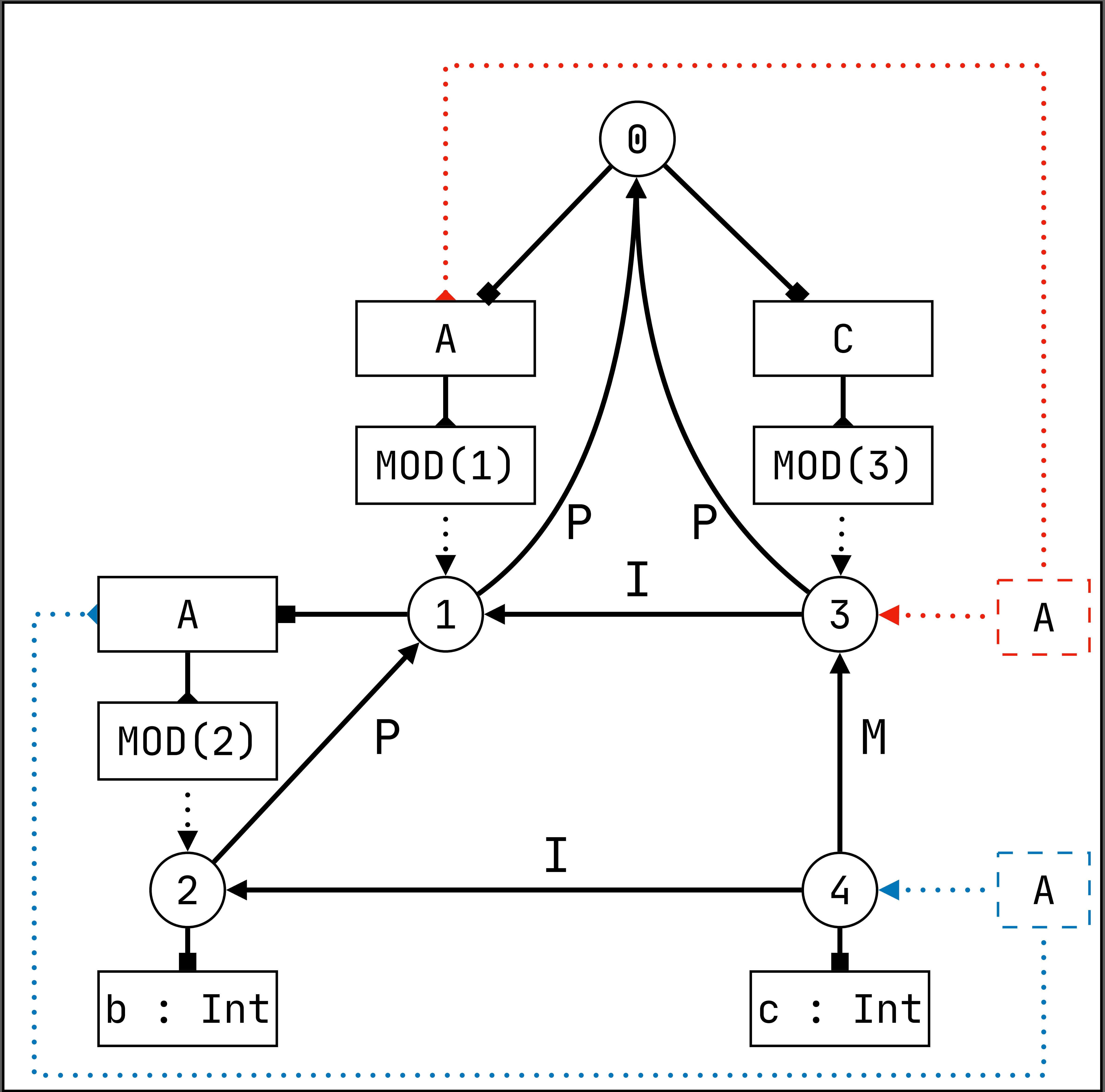



```
resolveVar(s, x) = ps :-
  query var
    filter P+ I* and { x' :- x' == x }
      min $ < P, $ < I, I < P and true
       in s ↦ ps.
```

Alternative Encoding: Scoped Imports — M Edge Label

```
signature
  sorts DecGroups
  constructors
    MOD    : scope → TYPE
    Module : ID * DecGroups → Decl
    Import : ID → Decl
    ModRef : ID * ID → Exp

    Decs   : list(Decl) → DecGroups
    Seq    : list(Decl) * DecGroups
             → DecGroups
```

```
module A {
  module A {
    def b = 1
  }
}

module C {
  import A;
  import A
  def c = b
}
```




```
resolveVar(s, x) = ps :-
  query var
    filter (P | M) P* (I | M)*
      and { x' :- x' == x }
      min $ < P, $ < I, I < P and true
       in s ⟼ ps.
```

Permission to Extend

Permission to Extend

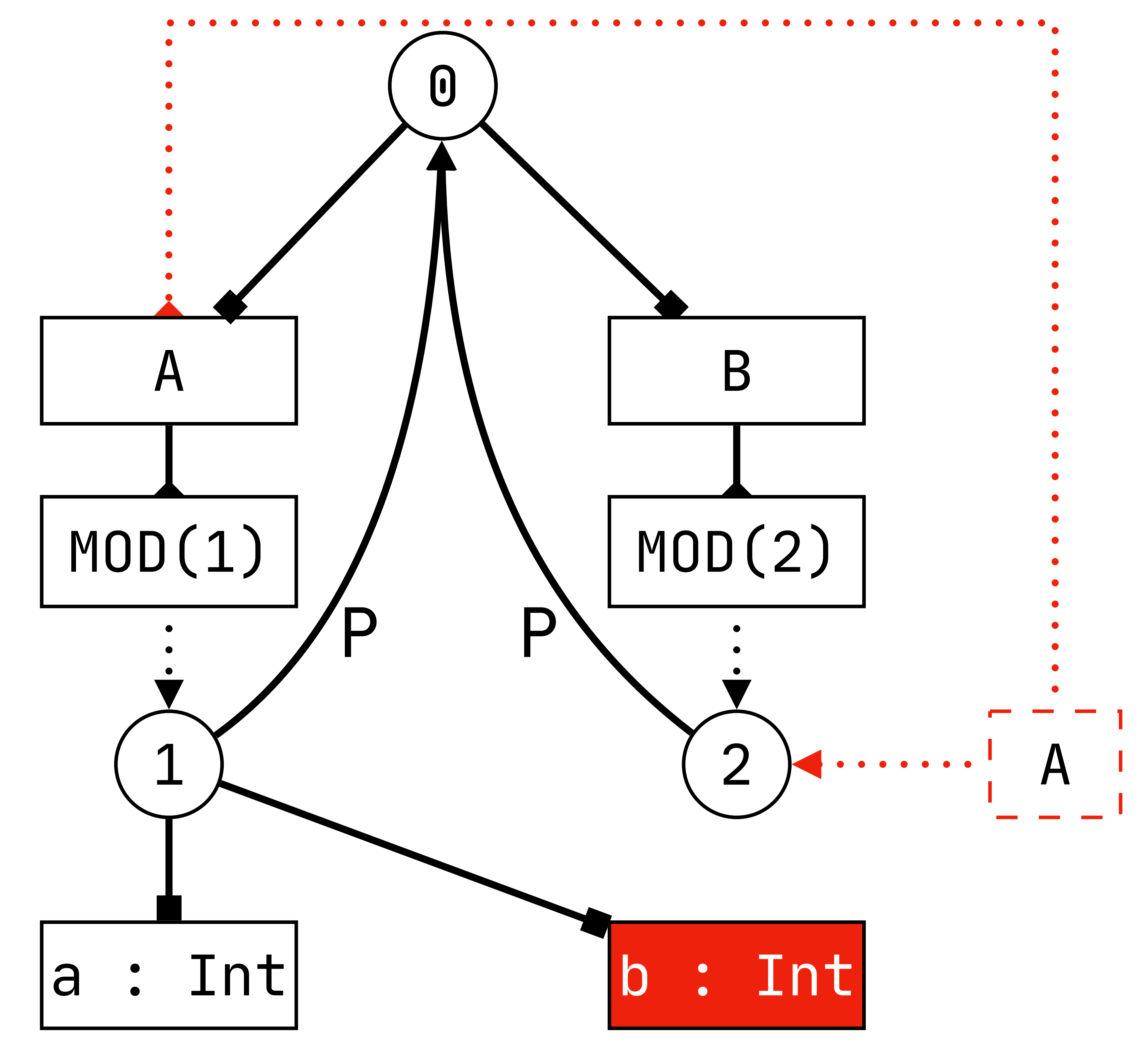
```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
    ExtendRemote : ID * ID * Exp → Decl
```

```
module A {
  def a = b
}
module B {
  def A.b := 2
}
```

extend remote: `def M.x := e`
extend module `M` with declaration of `x`

```
rules // extend remote

  declOk(s, ExtendRemote(m, x, e)) :- {s_mod T}
    typeOfModRef(s, m) == MOD(s_mod),
    typeOfExp(s, e) == T,
    declareVar(s_mod, x, T).
    // no permission to extend
```

Add declaration to scope obtained through a query



This is not allowed in Statix

A predicate can only extend scopes over which its has ownership, i.e. that it creates or gets passed down as an argument

Type-Dependent Name Resolution / Records

Records

```
signature
  constructors
    REC    : scope → TYPE
    Record : ID * list(FDecl) → Decl
    FDecl  : ID * Type → FDecl
    New    : ID * list(FBind) → Exp
    FBind  : ID * Exp → FBind
    Proj   : Exp * ID → Exp
```

```
record Point { x : Int, y : Int }

def p = Point{ x = 1, y = 2 }

> p.y
```

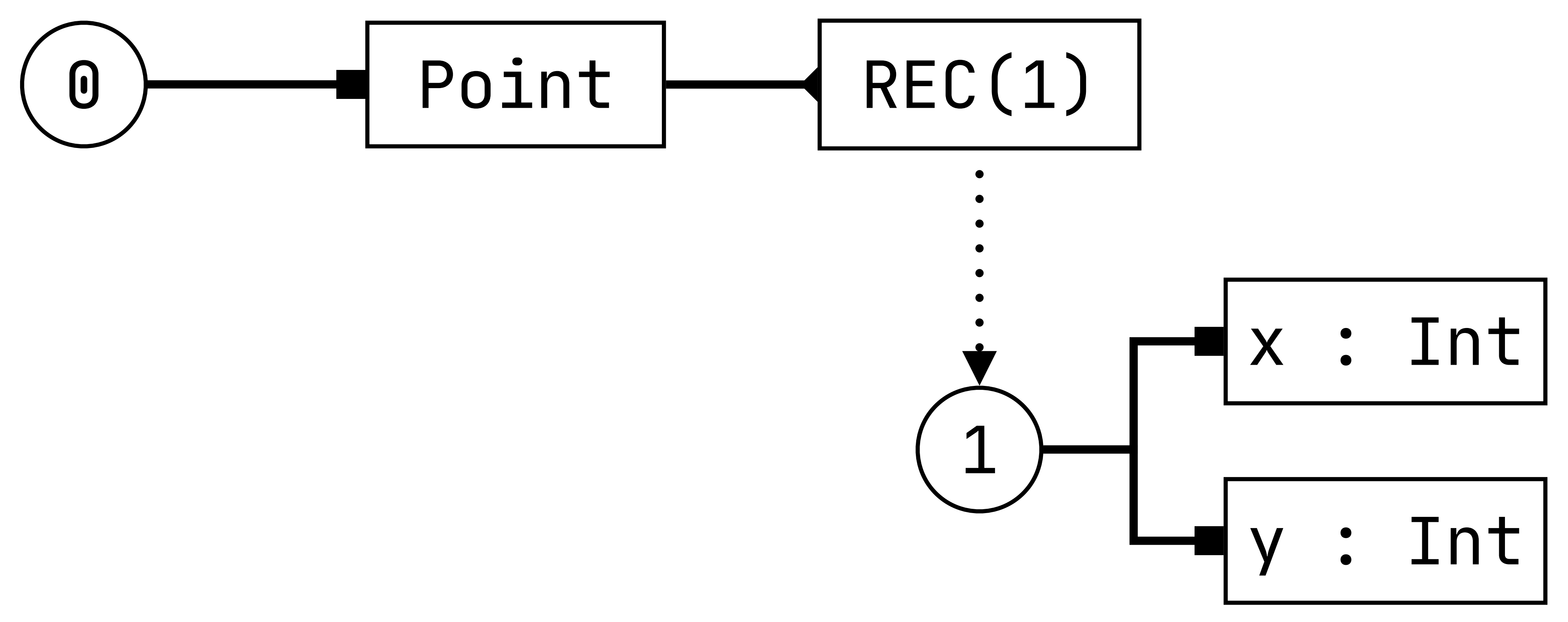
Record Type: Scope as Type

```
signature
  constructors
    REC    : scope → TYPE
    Record : ID * list(FDecl) → Decl
    FDecl  : ID * Type → FDecl
    New    : ID * list(FBind) → Exp
    FBind  : ID * Exp → FBind
    Proj   : Exp * ID → Exp
```

scope as type

```
rules // record type

  declOk(s, Record(x, fdecls)) :- {s_rec}
     new s_rec,
     fdeclsOk(s_rec, s, fdecls),
     declareType(s, x, REC(s_rec)).

  fdeclOk(s_bnd, s_ctx, FDecl(x, t)) :- {T}
    typeOfType(s_ctx, t) == T,
    declareVar(s_bnd, x, T).
```

```
record Point { x : Int, y : Int }

def p = Point{ x = 1, y = 2 }

> p.y
```

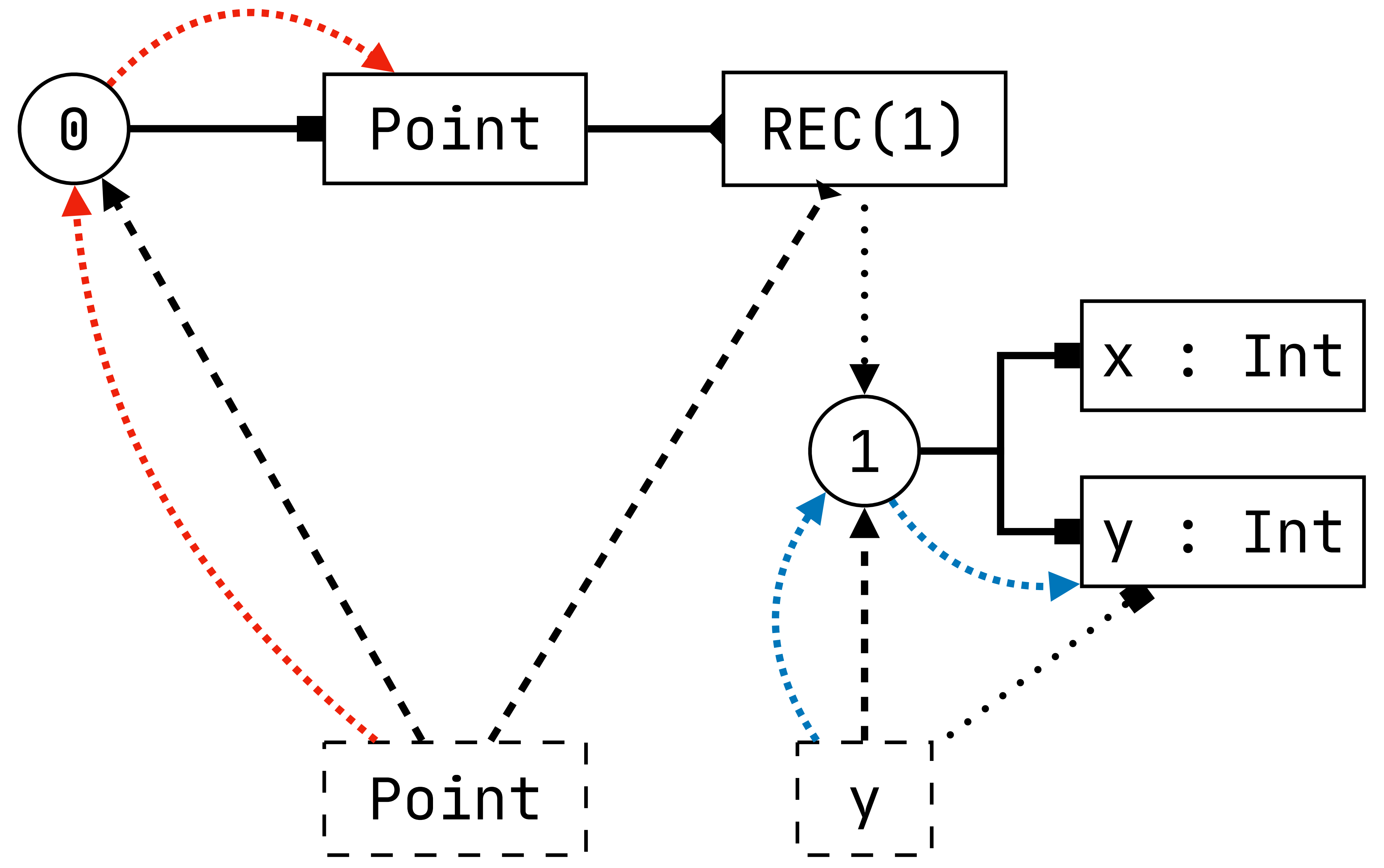


Record Construction & Initialization

```
signature
  constructors
    REC    : scope → TYPE
    Record : ID * list(FDecl) → Decl
    FDecl  : ID * Type → FDecl
    New    : ID * list(FBind) → Exp
    FBind  : ID * Exp → FBind
    Proj   : Exp * ID → Exp
```

```
rules // record construction

  typeOfExp(s, New(x, fbinds)) = REC(s_rec) :- {p d}
    typeOfTypeRef(s, x) == REC(s_rec),
    fbindsOk(s, REC(s_rec), fbinds).

  fbindOk(s, T_rec, FBind(x, e)) :- {T1 T2}
    typeOfExp(s, e) == T1,
    proj(T_rec, x) == T2,
    subtype(e, T1, T2).
```

```
record Point { x : Int, y : Int }

def p = Point{ x = 1, y = 2 }

> p.y
```

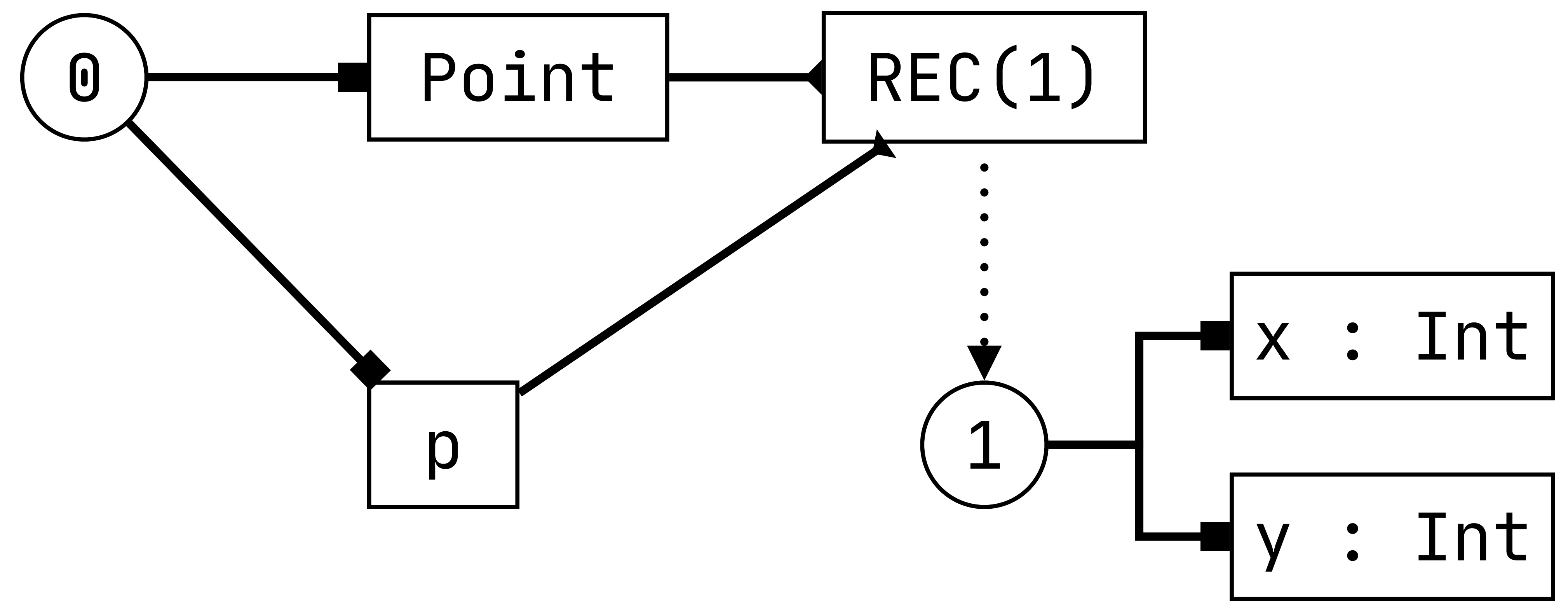


Type-Dependent Name Resolution

```
signature
  constructors
    REC    : scope → TYPE
    Record : ID * list(FDecl) → Decl
    FDecl  : ID * Type → FDecl
    New    : ID * list(FBind) → Exp
    FBind  : ID * Exp → FBind
    Proj   : Exp * ID → Exp
```

```
rules // record construction

  typeOfExp(s, New(x, fbinds)) = REC(s_rec) :- {p d}
    typeOfTypeRef(s, x) == REC(s_rec),
    fbindsOk(s, REC(s_rec), fbinds).

  fbindOk(s, T_rec, FBind(x, e)) :- {T1 T2}
    typeOfExp(s, e) == T1,
    proj(T_rec, x) == T2,
    subtype(e, T1, T2).
```

```
record Point { x : Int, y : Int }

def p = Point{ x = 1, y = 2 }

> p.y
```

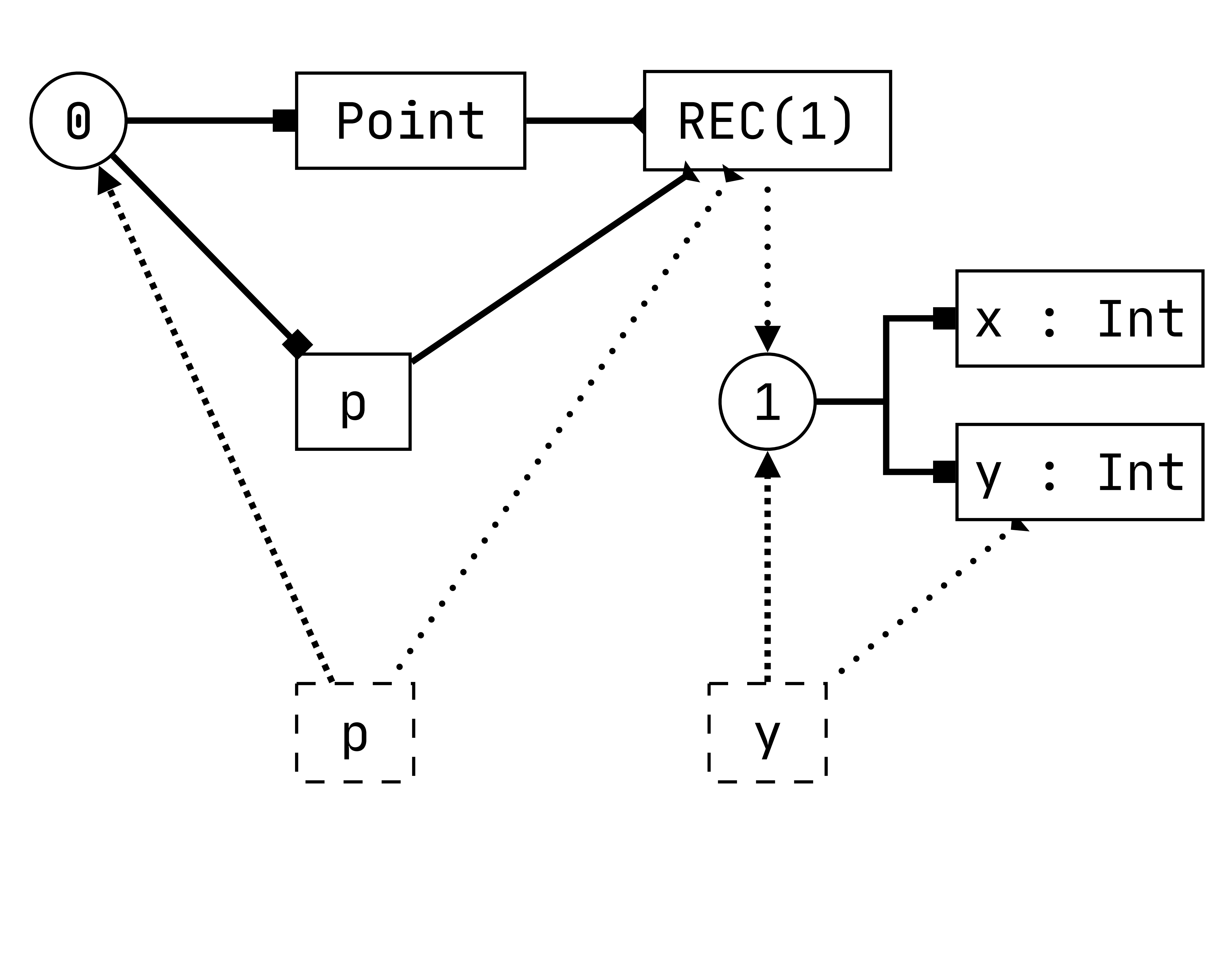


Type-Dependent Name Resolution

```
signature
  constructors
    REC    : scope → TYPE
    Record : ID * list(FDecl) → Decl
    FDecl  : ID * Type → FDecl
    New    : ID * list(FBind) → Exp
    FBind  : ID * Exp → FBind
    Proj   : Exp * ID → Exp
```

```
rules // record projection

  typeOfExp(s, Proj(e, x)) = T :- {p d s_rec S}
    typeOfExp(s, e) == REC(s_rec),
    typeOfVar(s_rec, x) == T.
```

```
record Point { x : Int, y : Int }

def p = Point{ x = 1, y = 2 }

> p.y
```



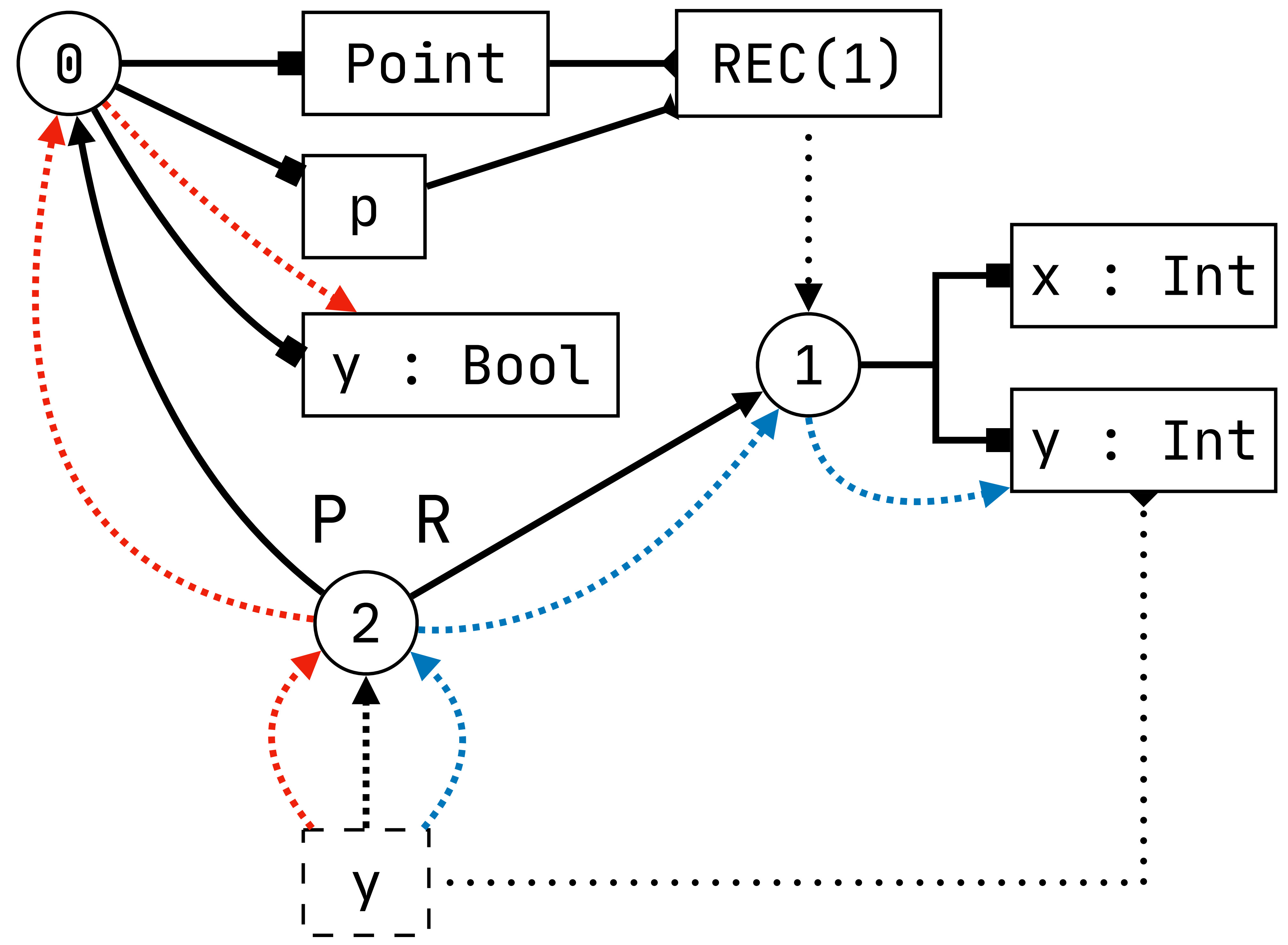
With

```
signature
  constructors
    REC    : scope → TYPE
    Record : ID * list(FDecl) → Decl
    FDecl  : ID * Type → FDecl
    New    : ID * list(FBind) → Exp
    FBind  : ID * Exp → FBind
    Proj   : Exp * ID → Exp
```

```
rules // with record value

  typeOfExp(s, With(e1, e2)) = T :- {s_with s_rec}
    typeOfExp(s, e1) == REC(s_rec),
    new s_with, s_with -P→ s, s_with -R→ s_rec,
    typeOfExp(s_with, e2) == T.
```

```
  resolveVar(s, x) = ps :-
    query var
      filter P* R* and { x' :- x' == x }
         min $ < P, R < P and true
          in s ↦ ps.
```

```
record Point { x : Int, y : Int }

def p = Point{x = 1, y = 2}

def y = true

> with p do y
```



Scheduling Constraint Resolution

Scheduling in Type Checkers

Type checker constructs scope graph

- Module, variable declarations
- Module imports
- Scopes

Type checker queries scope graph

- Type of variable reference

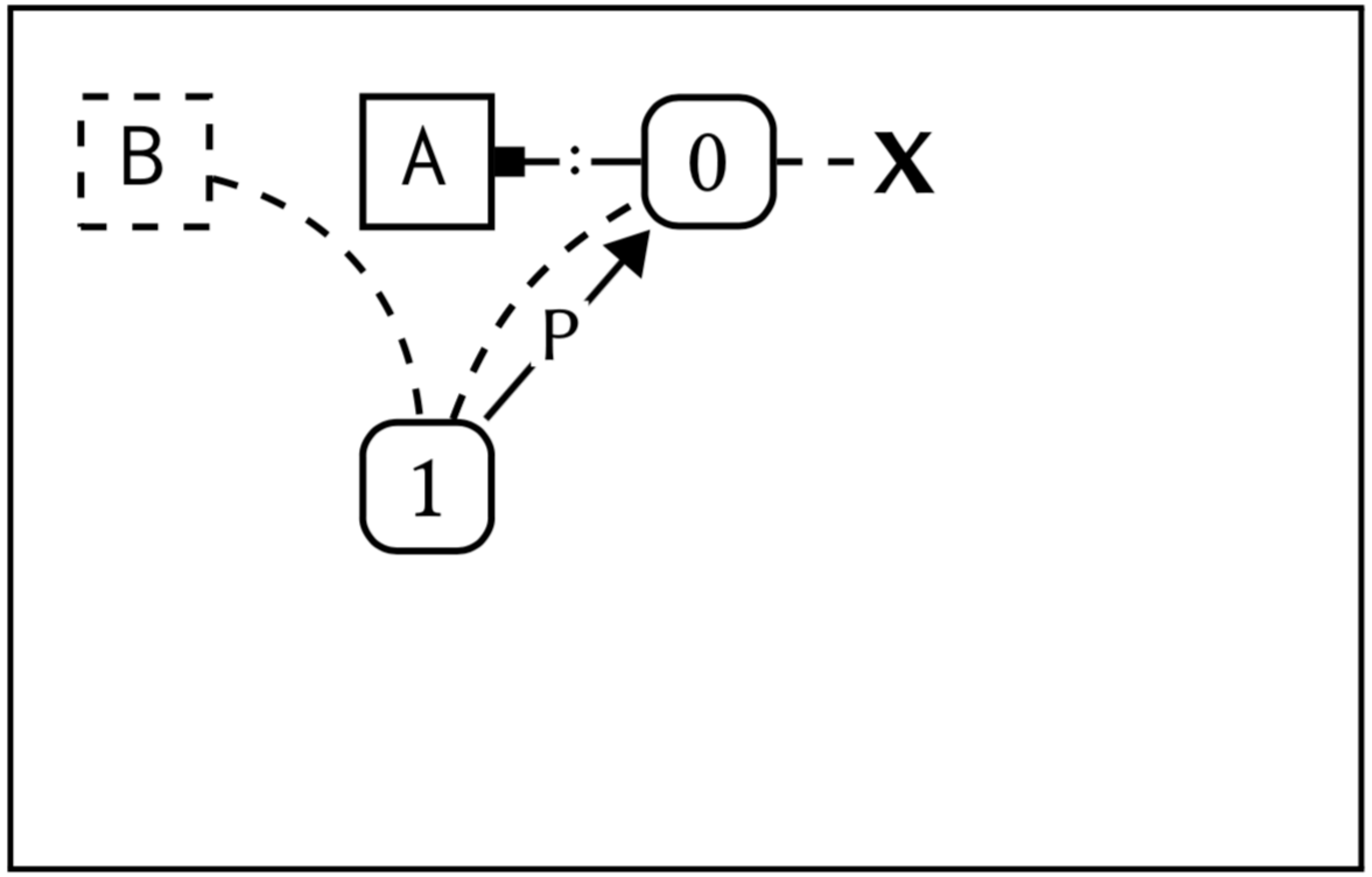
Scope graph construction depends on queries

- Imports require name resolution of module name

When is it safe to query the scope graph?

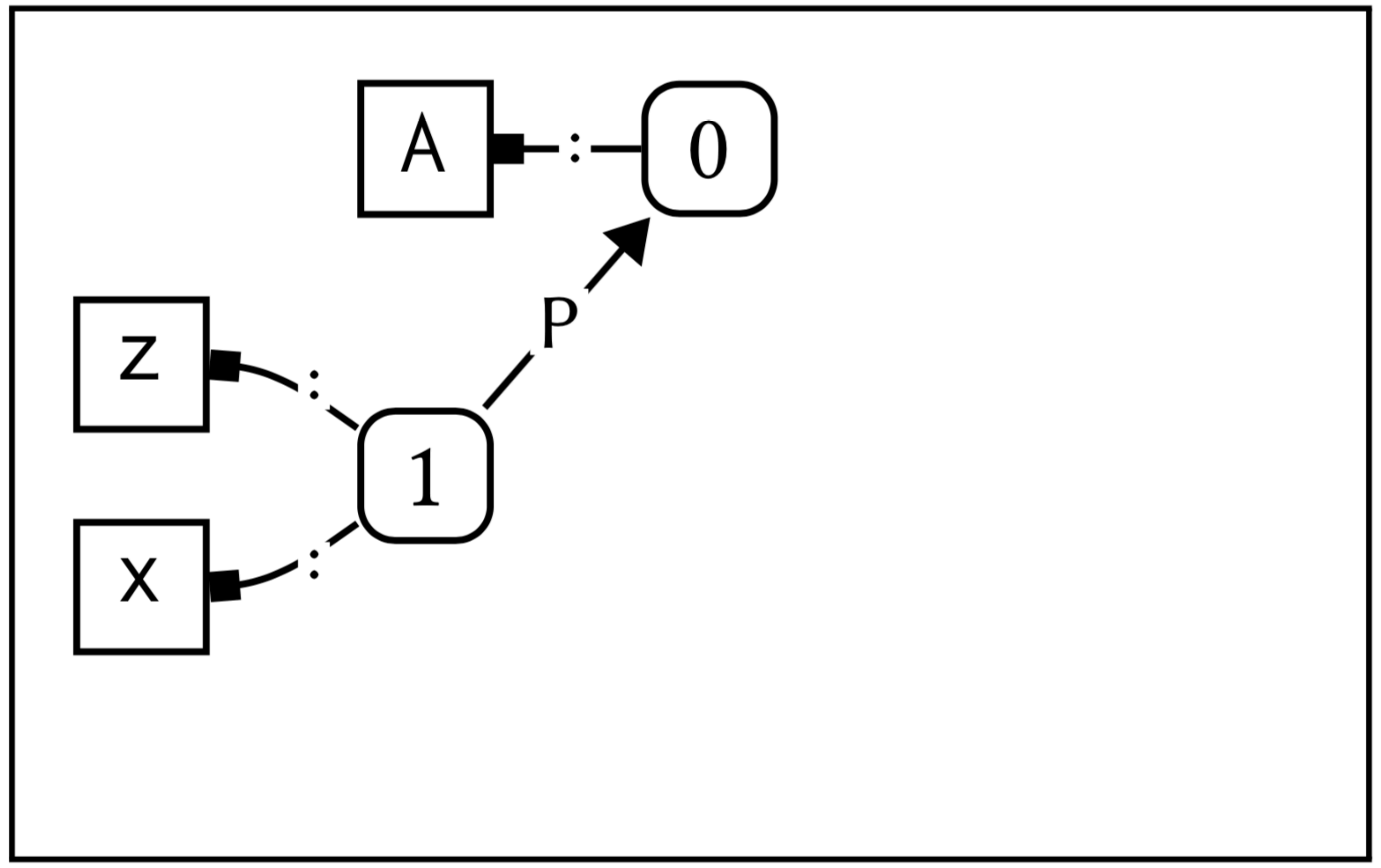
- In what order should type checker perform construction, querying?

A Single Stage Type Checker (Fails)

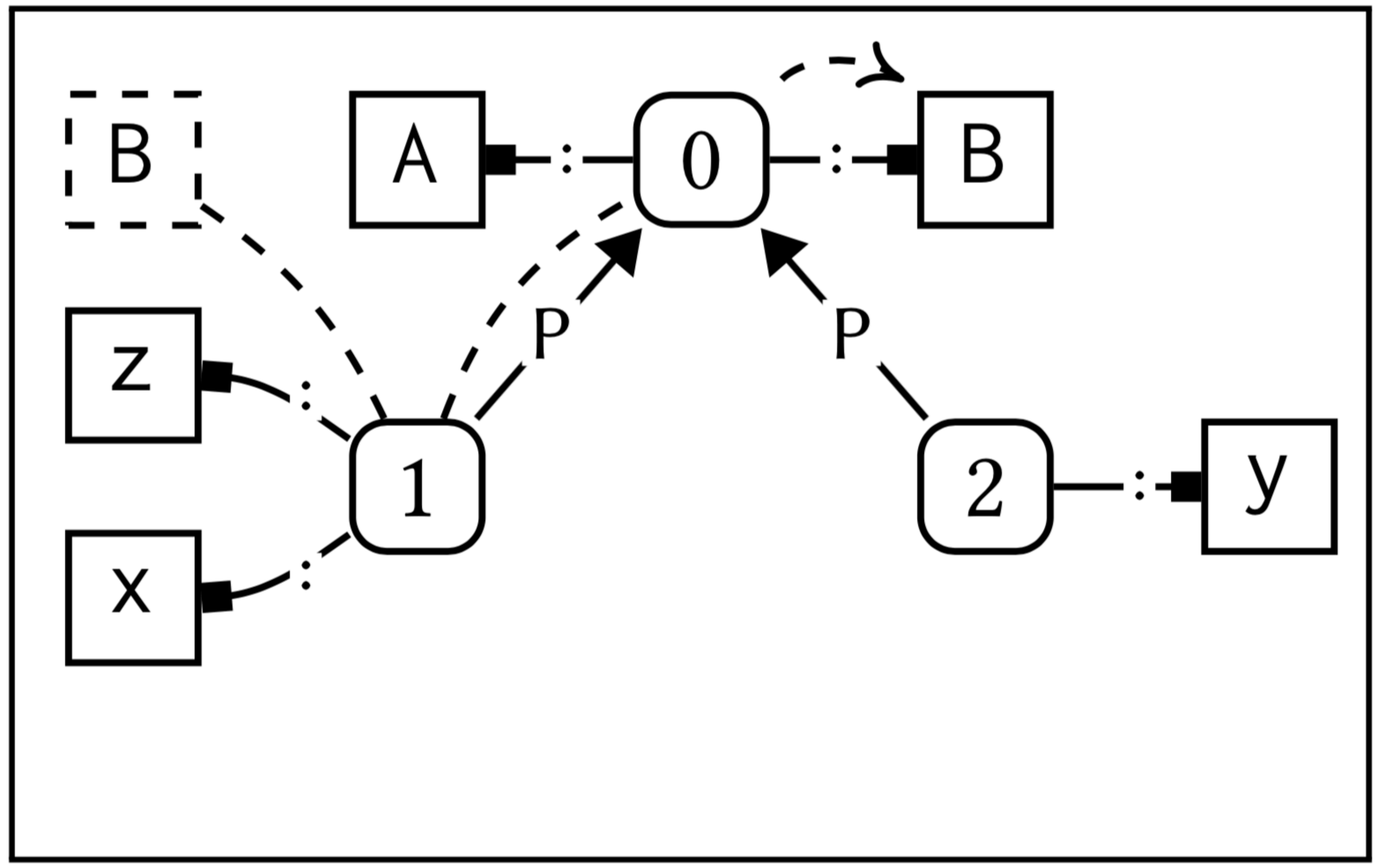



```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

A Two Stage Type Checker: Stage 1 (Build Module Table)



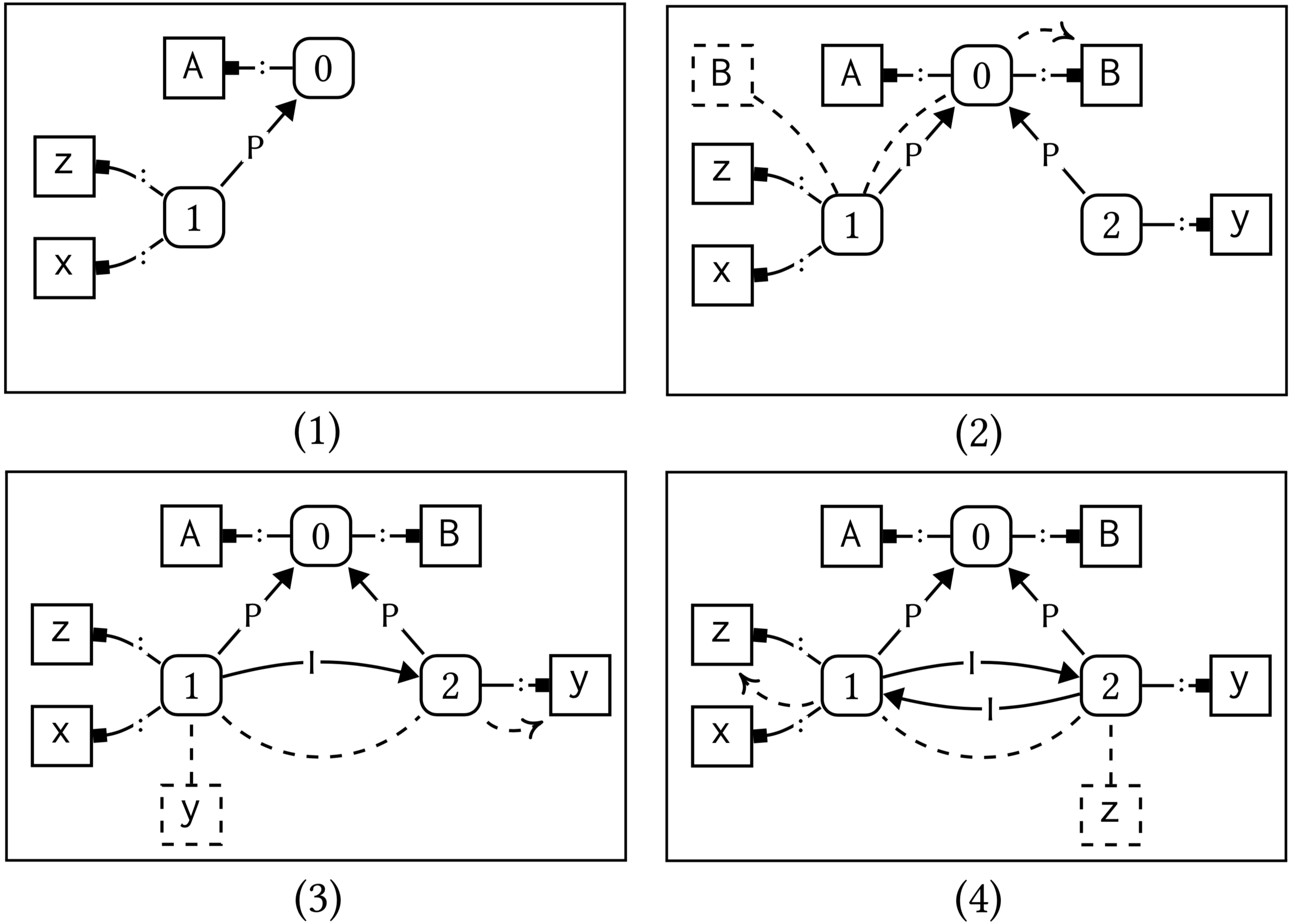
(1)



(2)

```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

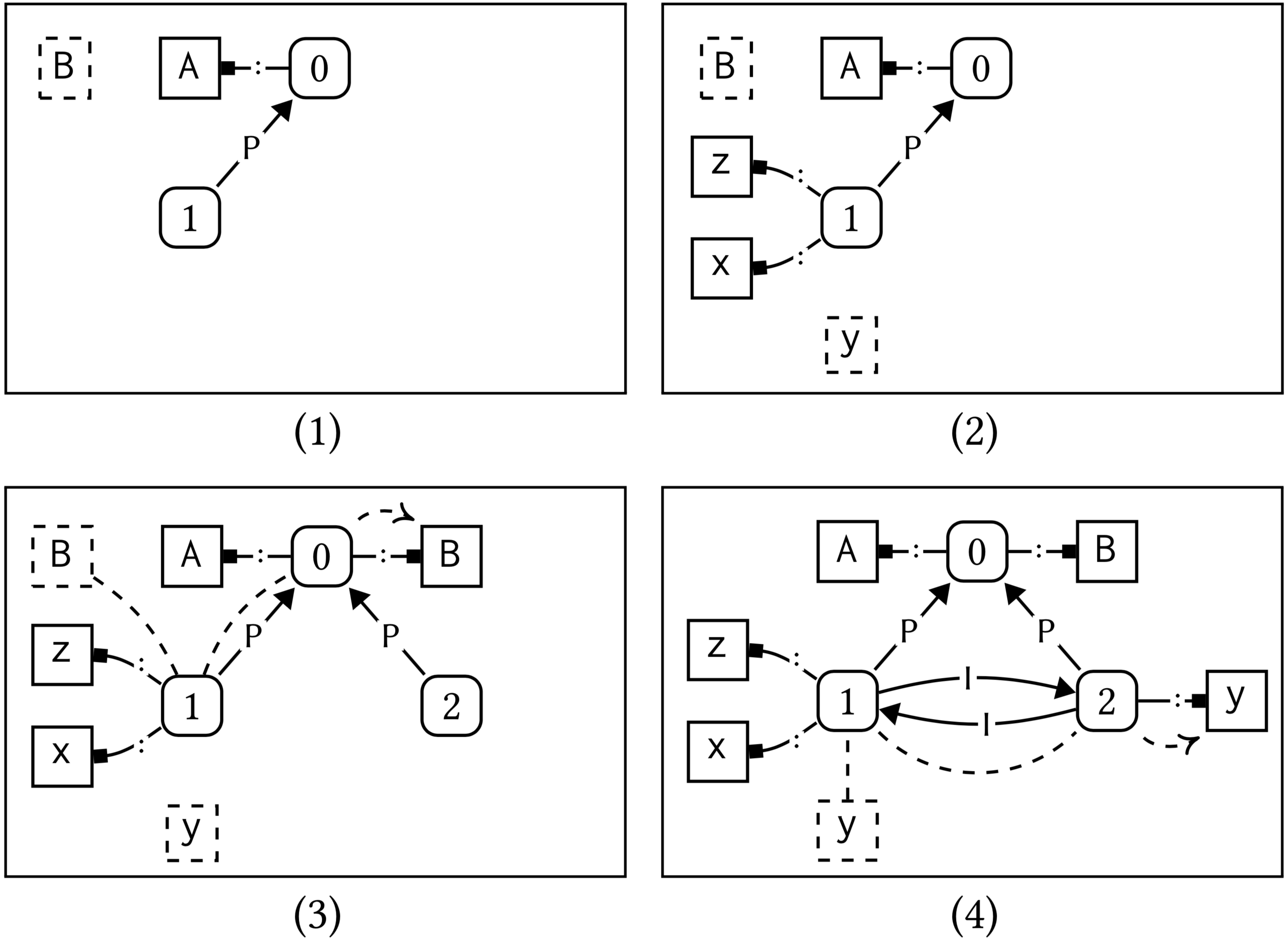
A Two Stage Type Checker: Stage 2 (Check Modules)



Requires that imports are resolved before variable references

```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

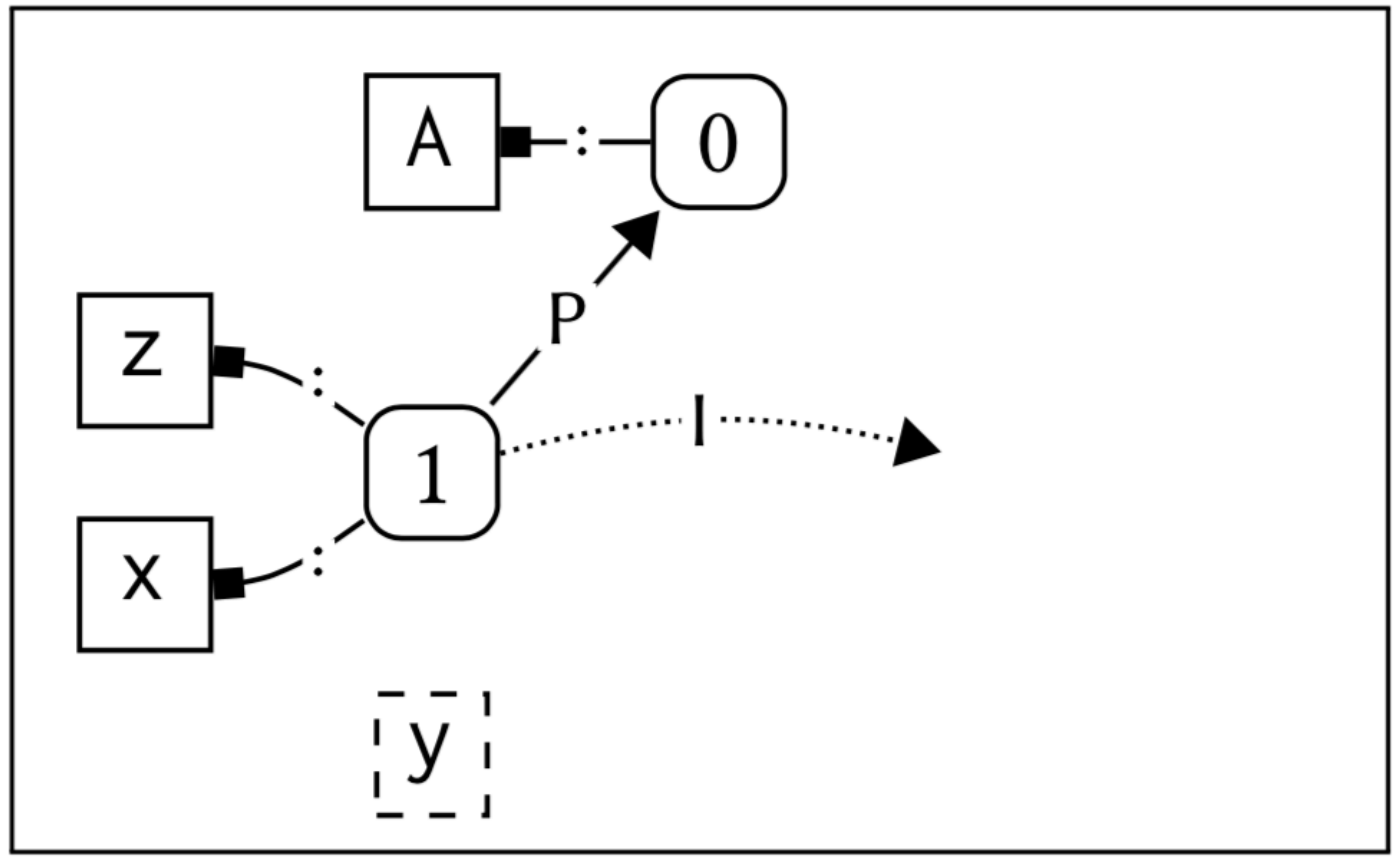
Dynamic



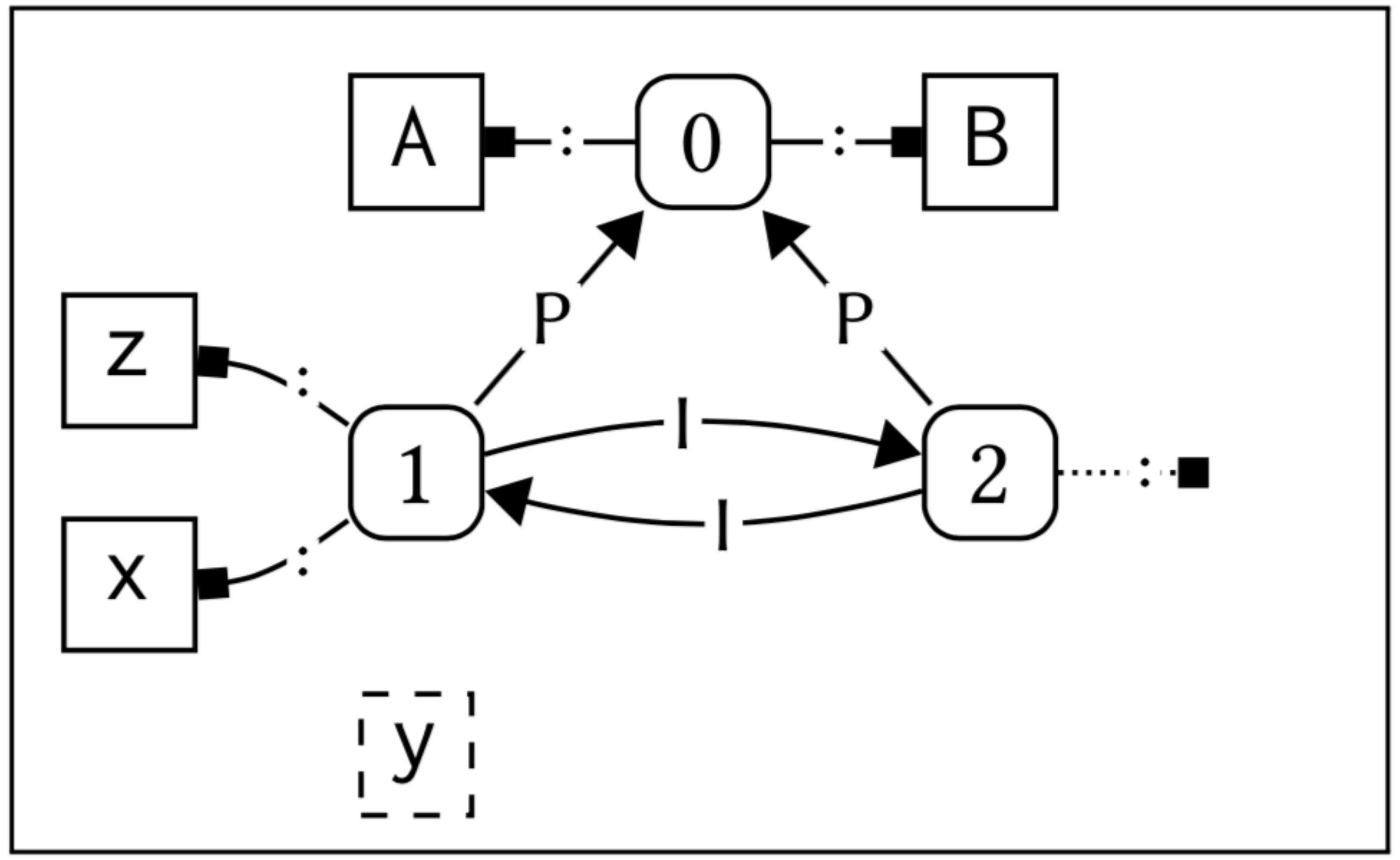
When do we have sufficient information to answer a query?

```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

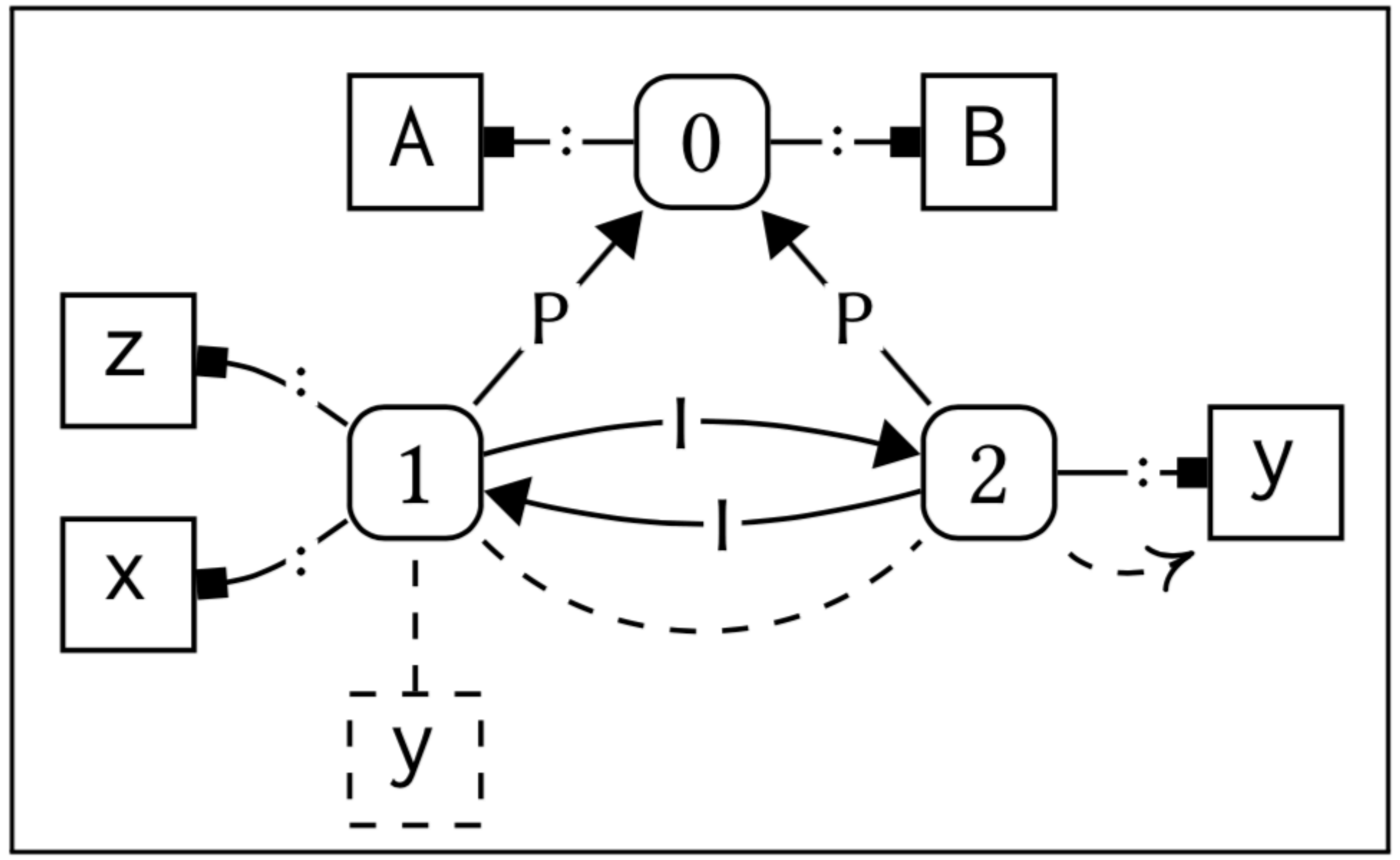
Critical Edges



(a) Intermediate scope graph



(b) Intermediate scope graph



(c) Final scope graph

```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

(Weakly) Critical Edges

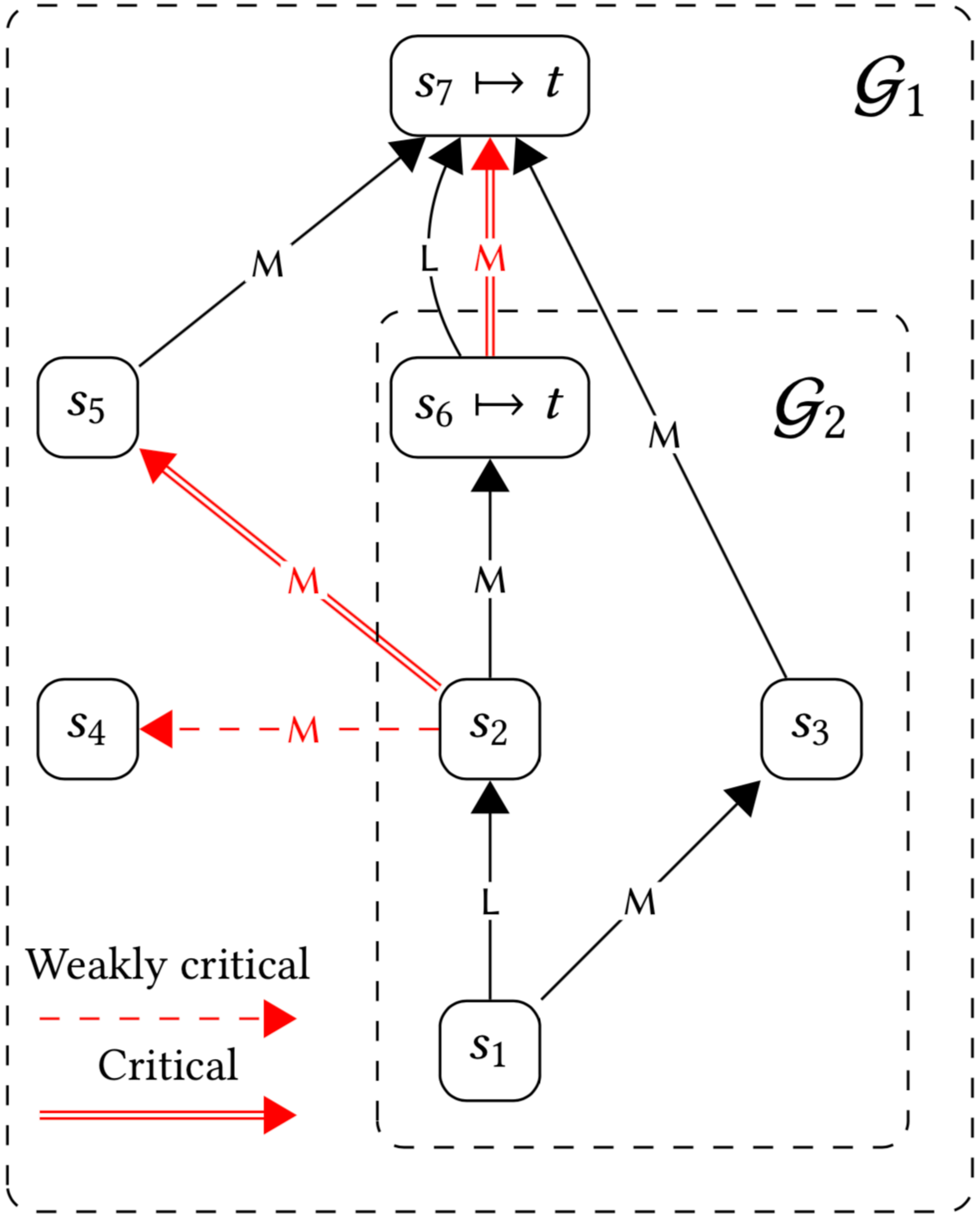


Fig. 11. (Weakly) critical edges for the query $s_1 \xrightarrow{LM*}\!\!\twoheadrightarrow_{\mathrm{R}} D$, assuming $t \in D$

Automatically Scheduling Constraint Resolution

Scope graph represents context information

- Type checker constructs scope graph
- Type checker queries scope graph
- Scope graph construction depends on queries

When is it safe to query the scope graph?

- When there are no more critical edges *for this query*

Conclusion

Statix

Modeling Name Binding with Scope Graphs

- Scopes + declarations + edges (reachability)
- Queries to resolve references
- Visibility policies = path disambiguation
  - path well-formedness + path specificity
- Model wide range of name binding policies

Scheduling Constraint Resolution [OOPSLA’20]

- Declarative: no explicit scheduling / staging / stratification of traversal
- Only perform queries when outcome will not be changed (capture)
- Don’t extend scopes ‘remotely’ (permission to extend)

Examples in this lecture: [ESOP’15] + [PEPM’16] in Statix

Scopes as Types [OOPSLA’18]

Applications

- Structural (sub)typing (records)
- Parametric polymorphism (System F)
- Nominal subtyping (FJ)
- Generic classes (FGJ)

Under investigation

- Make those encodings less clunky
- Hindley-Milner: inference supported, but how to generalize?

Ongoing Work

Incremental multi-file analysis

- Given a change, which files need to be reanalyzed?

Code completion [vision: ECOOP 2019]

- Given a hole, what can be filled in?
- Expressions, but also declarations, …

Refactoring

- Renaming, inlining, …

Other editor services

- Quick fixes, …

Random term generation

- Generate program that is well-typed and well-bound