Type Checkers from Declarative Type System Specifications in Statix

Eelco Visser

TU Delft

PLDI | ‘London’ | June 16, 2020

Type Checkers from Declarative Type System Specifications in Statix

Eelco Visser

Joint work with Hendrik van Antwerpen, Arjen Rouvoet, Andrew Tolmach, Casper Bach Poulsen, Pierre Néron, …

Goals of Statix Design

Static semantics definition

- Define static semantics of (domain-specific) programming languages

High-level and Understandable

- Can be used as reference documentation

Executable

- Generate type checkers

Declarative

- Understand in terms of declarative

Multi-purpose

- Derive many/all tools from single definition

Correct by Construction

- Implementations sound wrt declarative semantics

Type System Specification in Statix

Constraint-based language with declarative semantics

- Understand type system without algorithmic reasoning

Name binding using scope graphs

- *as part of constraint resolution*

Implementation

- Solver interprets specification as type checker
- Sound wrt declarative semantics
- Scheduling of constraint resolution based on language independent principles

Publications on Statix

Scopes as Types

- Van Antwerpen, Bach Poulsen, Rouvoet, Visser
- OOPSLA 2018

A constraint language for static semantic analysis based on scope graphs

- van Antwerpen, Néron, Tolmach, Visser, Wachsmuth
- PEPM 2016

A Theory of Name Resolution

- Néron, Tolmach, Visser, Wachsmuth
- ESOP 2015

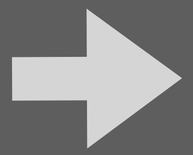
This Tutorial: Statix by Example

- Concrete and abstract syntax
- Modular language definition
- Type predicates
- Declaring and resolving names
- Lexical scope
- Scopes as types
- Modules and imports
- Records
- Incompleteness
- Scheduling queries and critical edges
- Permission to extend

Concrete and Abstract Syntax

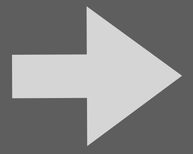
From Concrete Syntax Definition to Abstract Syntax Signature

```
module lang/base/syntax

imports lang/base/lexical

lexical sorts ID INT STRING
sorts Exp Type Val Decl Bind TYPE
context-free syntax
  Exp = <(<Exp>)> {bracket}
  Type = <(<Type>)> {bracket}
```



```
module signatures/lang/base/syntax-sig

imports signatures/lang/base/lexical-sig

signature
  sorts
    ID = string
    INT = string
    STRING = string
    Exp Type Val Decl Bind TYPE
```

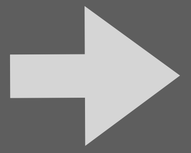
```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax
  Exp.Int    = <<INT>>
  Exp.Min    = [-[Exp]]
  Exp.Add    = <<Exp> + <Exp>> {left}
  Exp.Sub    = <<Exp> - <Exp>> {left}
  Exp.Mul    = <<Exp> * <Exp>> {left}
  Type.IntT = <Int>

context-free priorities
  Exp.Mul > {left: Exp.Add Exp.Sub}
```



```
module signatures/lang/arithmetic/syntax-sig

imports signatures/lang/base/syntax-sig

signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

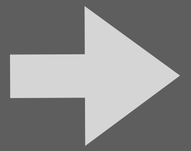
From Concrete Syntax Definition to Abstract Syntax Signature

```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax
  Exp.Int    = <<INT>>
  Exp.Min    = [-[Exp]]
  Exp.Add    = <<Exp> + <Exp>> {left}
  Exp.Sub    = <<Exp> - <Exp>> {left}
  Exp.Mul    = <<Exp> * <Exp>> {left}
  Type.IntT = <Int>

context-free priorities
  Exp.Mul > {left: Exp.Add Exp.Sub}
```



```
module signatures/lang/arithmetic/syntax-sig

imports signatures/lang/base/syntax-sig

signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

```
1 + 2 * 3
```

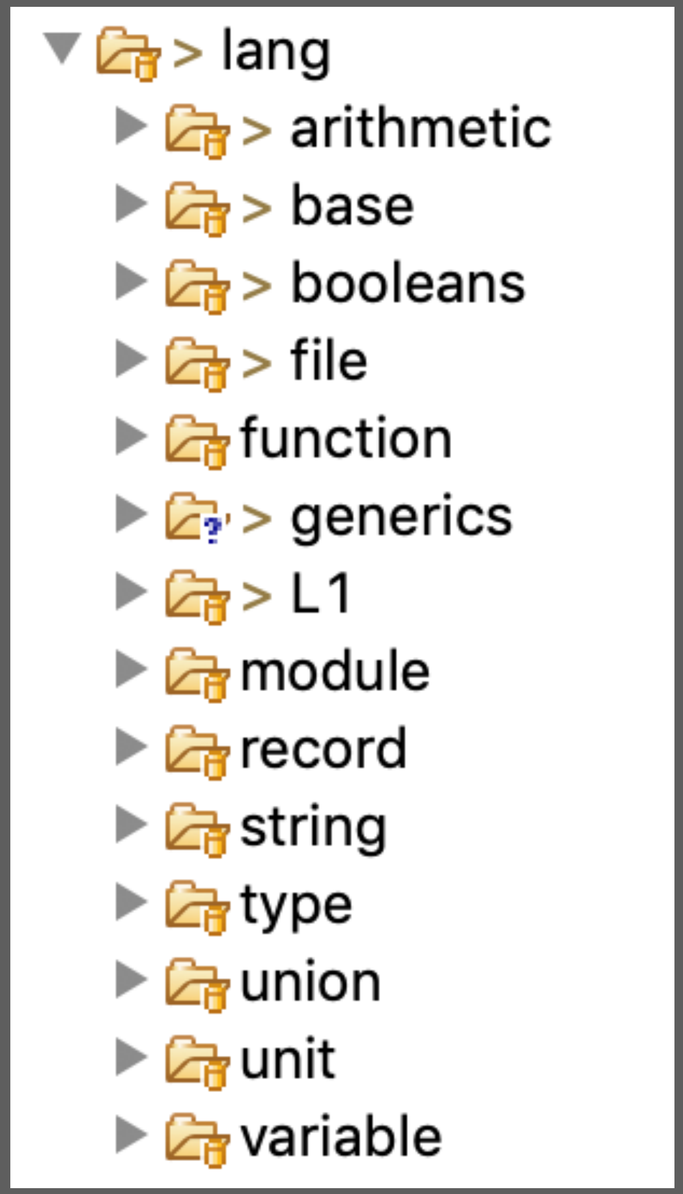


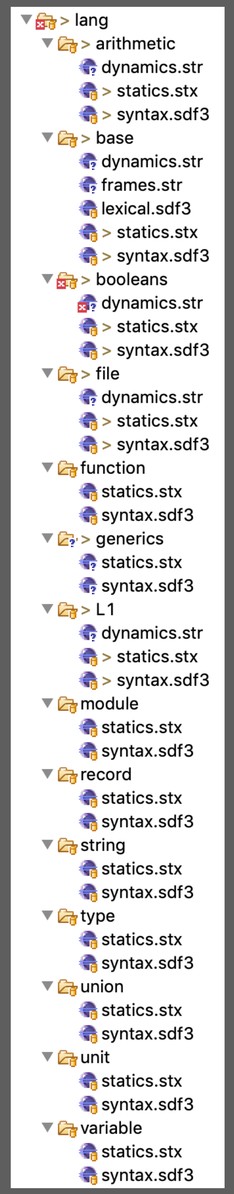
```
Add(
  Int("1"),
  Mul(
    Int(“2"),
    Int(“3")))
```

From here we will use concrete syntax examples and abstract syntax rules

Modular Language Definition

Modular Language Definition: Composing Languages from Components

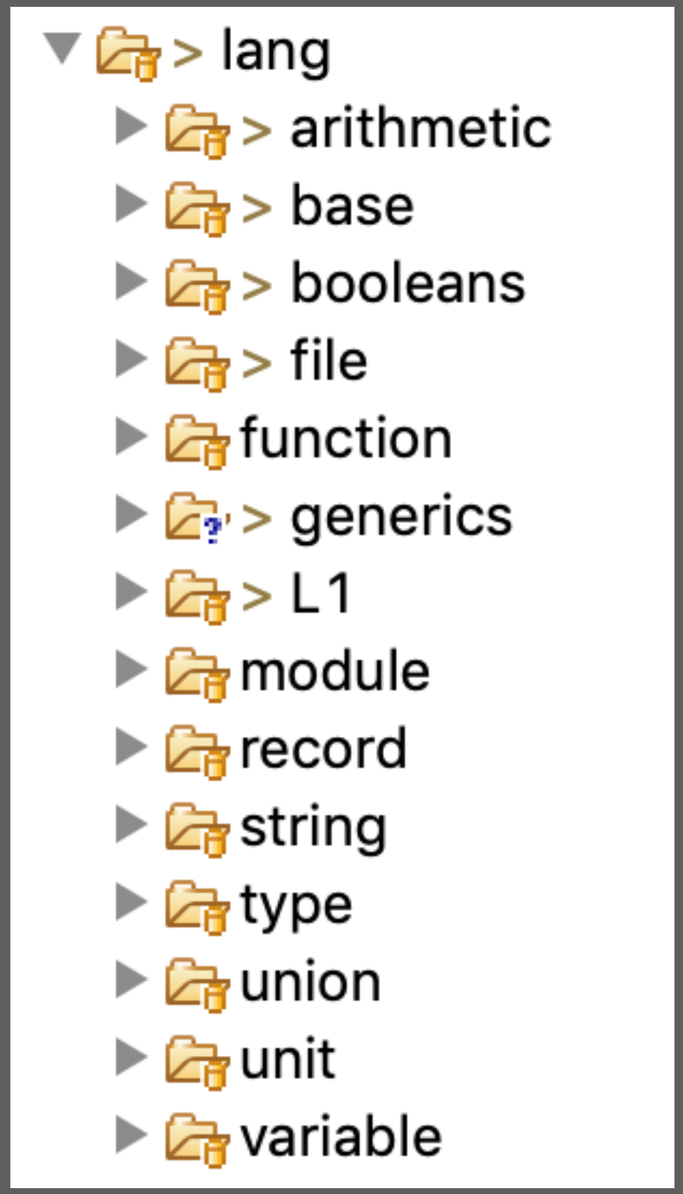


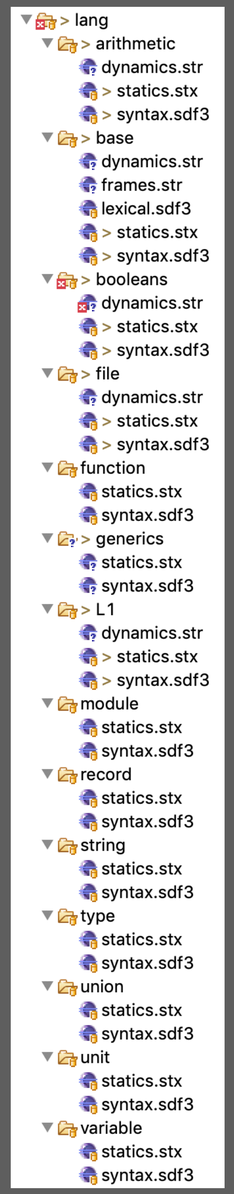



```
module lang/L1/statics

imports lang/base/statics

imports lang/arithmetic/statics
imports lang/booleans/statics
imports lang/string/statics
imports lang/unit/statics
//imports lang/union/statics

//imports lang/generics/statics

imports lang/record/statics
imports lang/function/statics

imports lang/variable/statics
imports lang/let/statics
imports lang/type/statics
imports lang/module/statics
//imports lang/module-simple/statics
//imports lang/module-seq/statics

imports lang/file/statics
```

Easily experiment with alternative language definitions

Modular Language Definition: Extending Common Base Types






```
module signatures/lang/base/syntax-sig

imports
  signatures/lang/base/lexical-sig

signature
  sorts
    ID     = string
    INT    = string
    STRING = string
    Type   // syntactic types
    TYPE   // semantic types
    Exp    // expressions
    Decl   // declarations
    Bind   // binding
    Val    // values
```

Work in progress: combining language projects

Predicates

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE

rules // well-typedness of ...

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

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Use `maps` to apply a predicate to all elements of a list

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

`typeOfExp(s, e) == T`
expression `e` has type `T` in scope `s`

`typeOfType(s, t) == T`
syntactic type `t` has semantic type `T` in scope `s`

`declOk(s, d)`
declaration `d` is well-defined (Ok) in scope `s`

Functional Notation vs Predicate Notation

```
rules

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

`typeOfExp(s, e) == T`

expression e has type T in scope s

One expression has one type

(Solver does not match on type argument)

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

`typeOfExp(s, e, T)`

expression e has type T in scope s

One expression can have multiple types

Predicates are Defined by Rules

Predicate

```
typeOfType : scope * Type → 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

Declarative Reading vs Operational Reading

Predicate

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

Rule

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

Head

Premises

Declarative Names	Operational Names
`typeOfExp(e) == T`	`typeCheck(e) = T`
The type of expression `e` is `T`	Type checking expression `e` produces type `T`
Type system defines a (functional) relation	Type checking is a process

Syntax-Directed Definitions: One Rule per Language Construct

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules

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

```
module signatures/lang/arithmetic/syntax-sig

imports signatures/lang/base/syntax-sig

signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

```
module lang/arithmetic/statics

imports lang/base/statics
imports signatures/lang/arithmetic/syntax-sig

signature
  constructors
    INT : TYPE

rules
  typeOfType(s, IntT()) = INT().

rules
  typeOfExp(s, Int(i)) = INT().

  typeOfExp(s, Min(e)) = INT() :-
    typeOfExp(s, e) == INT().

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

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

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

From Now: No Module Headers

```
rules

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

```
signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

```
signature
  constructors
    INT : TYPE

rules
  typeOfType(s, IntT()) = INT().

rules
  typeOfExp(s, Int(i)) = INT().

  typeOfExp(s, Min(e)) = INT() :-
    typeOfExp(s, e) == INT().

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

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

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

Types Are Just Terms

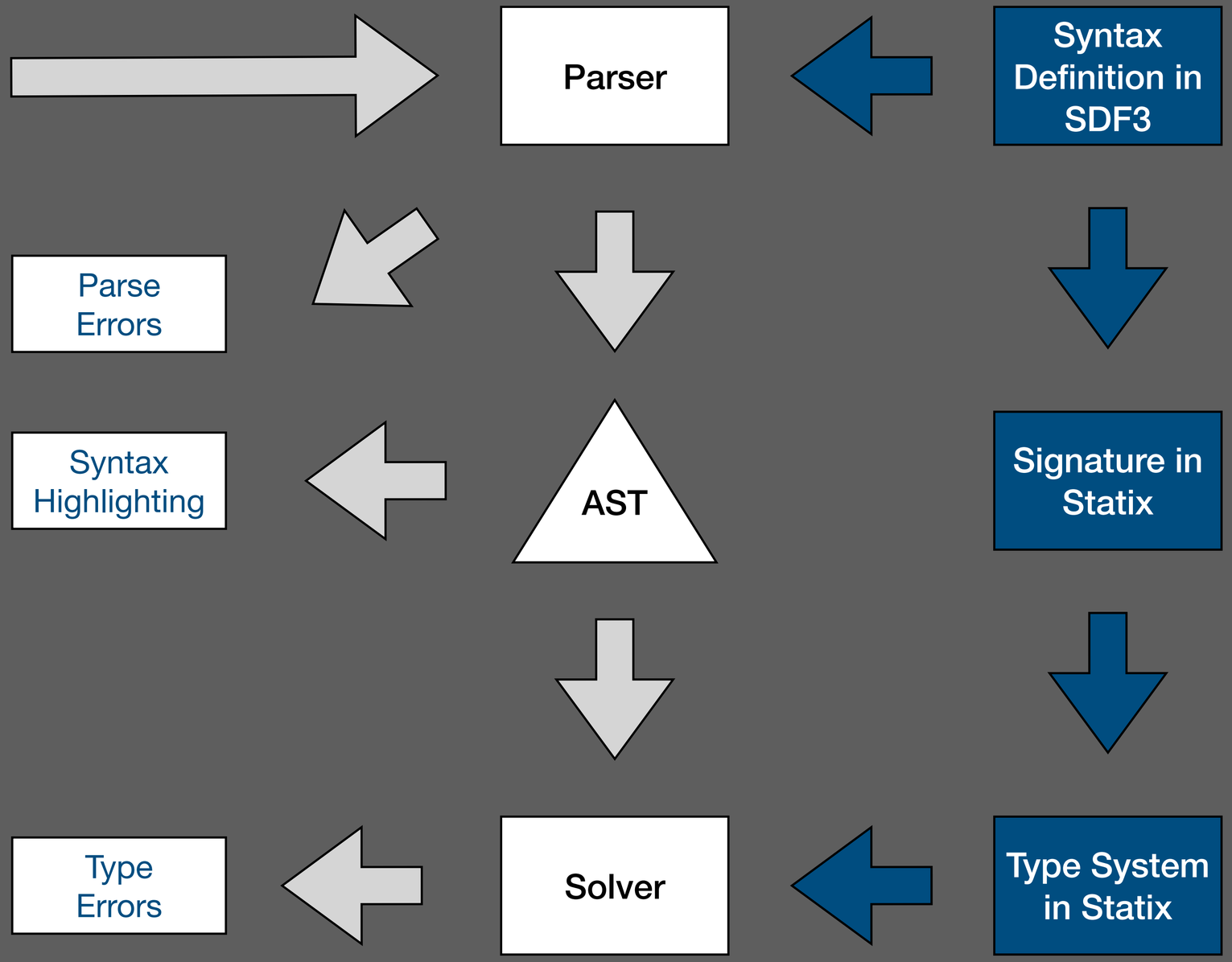
```
signature
  constructors
    BoolT     : Type
    BOOL      : TYPE
    True      : Exp
    False     : Exp
    Not       : Exp → Exp
    And       : Exp * Exp → Exp
    Or        : Exp * Exp → Exp
    If        : Exp * Exp * Exp → Exp
    Eq        : Exp * Exp → Exp
```

```
rules // operations on types

  subtype  : Exp * TYPE * TYPE
  equitype : TYPE * TYPE
  lub      : TYPE * TYPE → TYPE

  subtype(_, T, T).
  equitype(T, T).
  lub(T, T) = T.
```

```
rules

  typeOfType(s, BoolT()) = BOOL().

rules

  typeOfExp(s, True()) = BOOL().

  typeOfExp(s, False()) = BOOL().

  typeOfExp(s, And(e1, e2)) = BOOL() :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == BOOL().

  typeOfExp(s, If(e1, e2, e3)) = lub(T1, T2) :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == T1,
    typeOfExp(s, e3) == T2,
    equitype(T1, T2).

  typeOfExp(s, Eq(e1, e2)) = BOOL() :- {T1 T2}
    typeOfExp(s, e1) == T1,
    typeOfExp(s, e2) == T2,
    equitype(T1, T2).
```

From Declarative Definition to Type Checker

```
1> 1 + 2 * 3
2
3> true && false
4
5> 1 ^ 2
6
7> true + 4
8
9> 1 && (true || false)
10
11> if 1 == 1 then
12    true
13  else
14    1 == 3
15
16> if 1 == 1 then
17    true
18  else
19    2
```



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

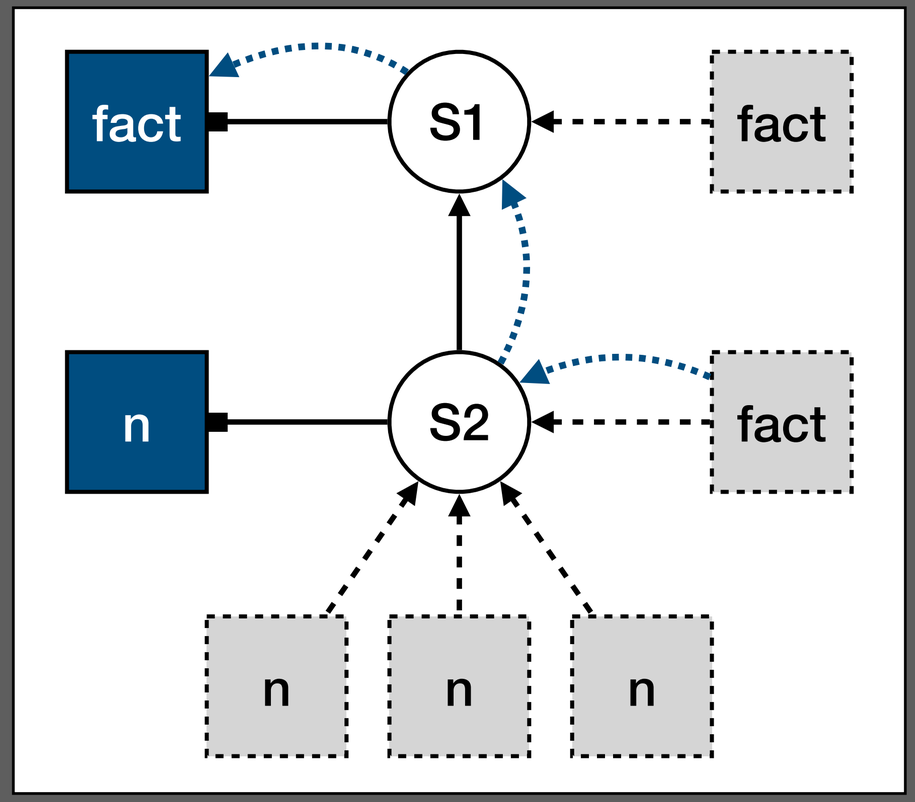
A systematic, uniform approach to name resolution?

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
- Records
- Modules
- Permission to Extend
- Scheduling Resolution

Declaring and Resolving Names

Declarations and References

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

```
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

```
rules

  typeOfExp(s, Var(x)) = typeOfVar(s, x).

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

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

  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(e, T2, T1).
```

```
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

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

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE
```

Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

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

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE
```

Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

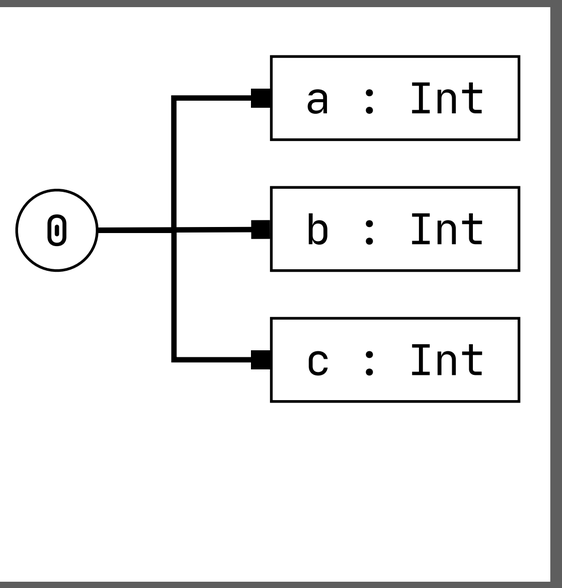
declaration relation

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

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T.
```

variable x is declared in scope s with type T



Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

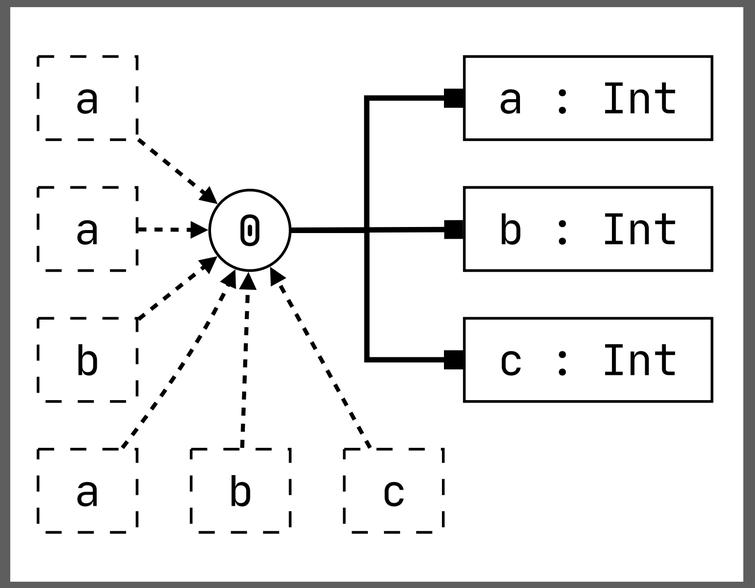
```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T.

  typeOfVar(s, x) = T :- {x'}
    typeOfDecl of Var{x} in s ⟼ [(_, (Var{x'}, T))].
```

variable x is declared in scope s with type T

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



Undefined Variable

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence ⟶ TYPE
```

namespace

resolution policy

declaration relation

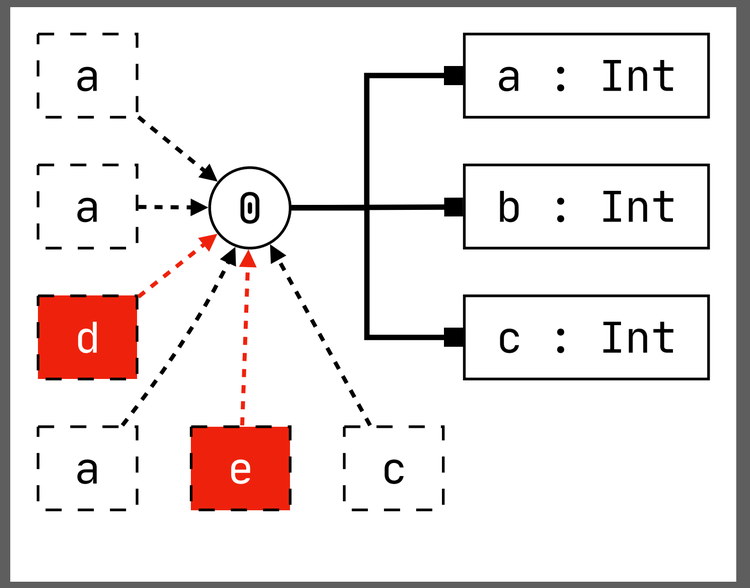
```
def a = 0
def b = a + 1
def c = a + d
> a + e + c
```

undefined variable

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string ⟶ TYPE

  declareVar(s, x, T) :-
    s ⟶ Var{x} with typeOfDecl T.

  typeOfVar(s, x) = T :- {x'}
    typeOfDecl of Var{x} in s ⟼ [(_, (Var{x'}, T))].
```

variable x is declared in scope s with type T

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



resolution query is not satisfiable

Duplicate Definition

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence ⟶ TYPE
```

namespace

resolution policy

declaration relation

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

what we want

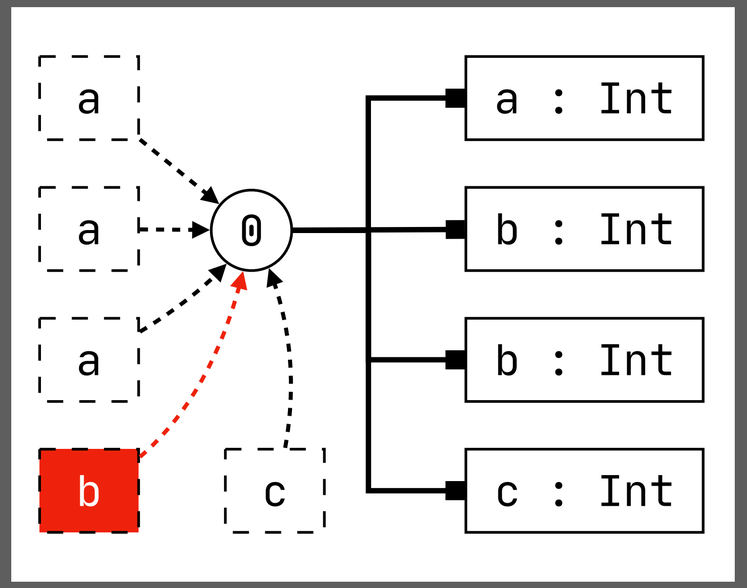
```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

what we get

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string ⟶ TYPE

  declareVar(s, x, T) :-
    s ⟶ Var{x} with typeOfDecl T.

  typeOfVar(s, x) = T :- {x'}
    typeOfDecl of Var{x} in s ⟼ [(_, (Var{x'}, T))].
```

variable x is declared in scope s with type T

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



Duplicate Definition: Permissive Resolution

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

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

duplicate definition

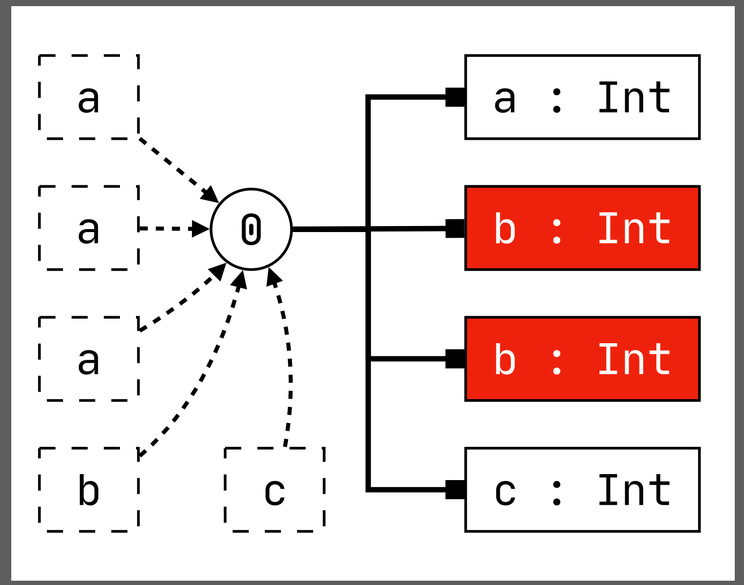
```
def a = 0
def b = a + 1
def b = 2 + a
def c = 3
> a + b + c
```

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T,
    typeOfDecl of Var{x} in s ⟼ [(_, (_, T))]
    | error $[Duplicate definition of variable [x]].
    // declaration is distinct

  typeOfVar(s, x) = T :- {x'}
    typeOfDecl of Var{x} in s ⟼ [(_, (Var{x'}, T))|_]
    | error $[Variable [x] not defined],
    // permissive lookup to cope with double declaration
    @x.ref := x'.
```

there should only be one declaration

there should be at least one matching declaration

record link from reference to declaration



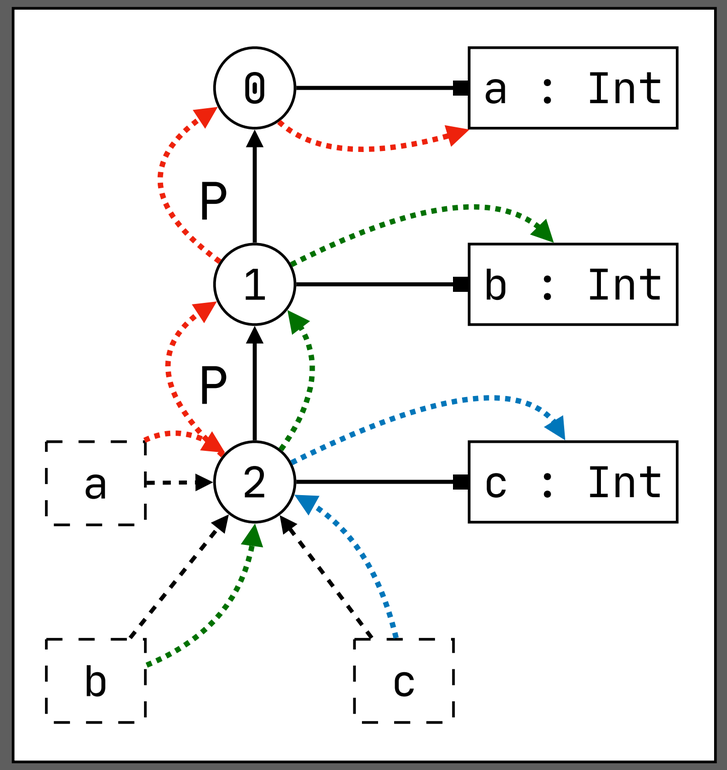
How about shadowing?

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

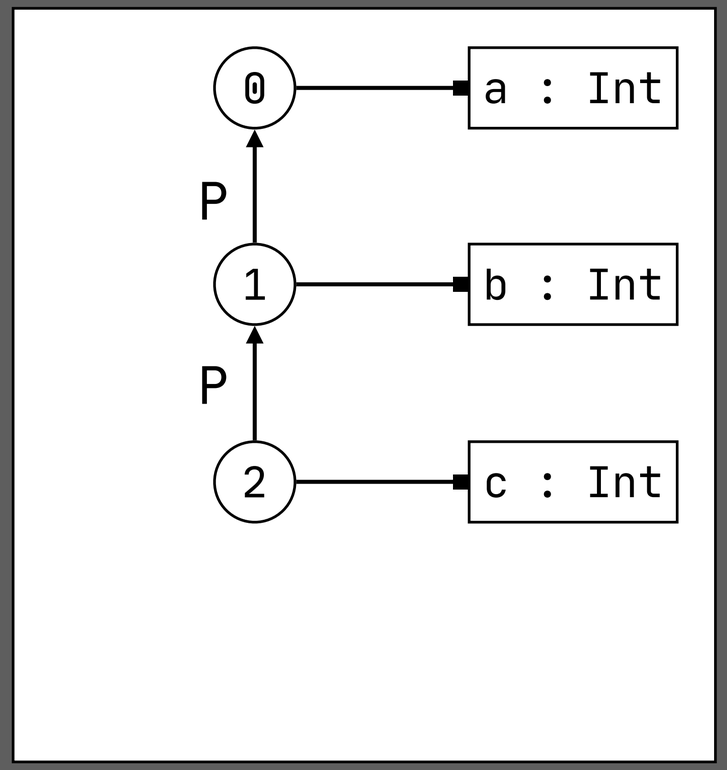
```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
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

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



Path Wellformedness: Reachability

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

```
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

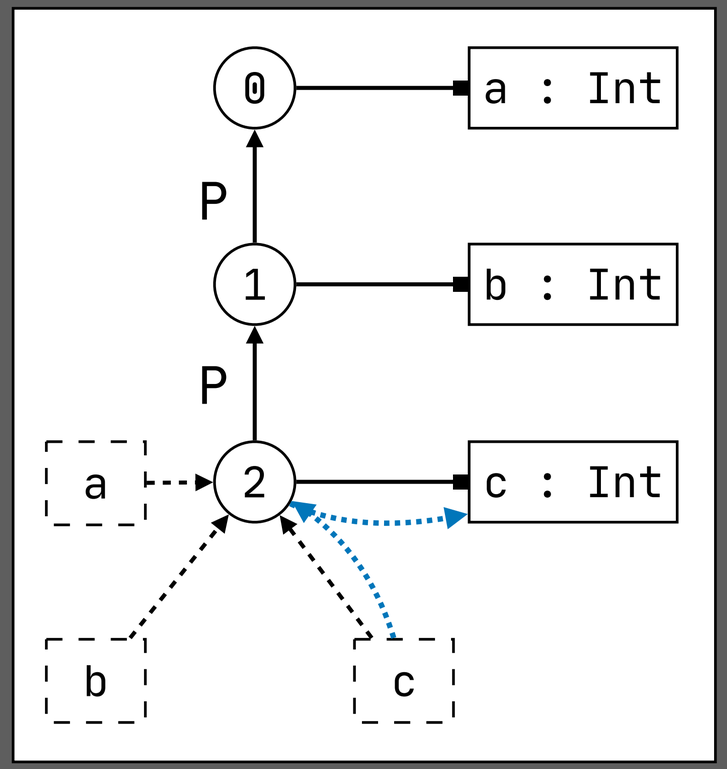
scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
```

empty path e only allows resolution in `this' scope

```
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
```



Path Wellformedness: Reachability

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

```
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.
```

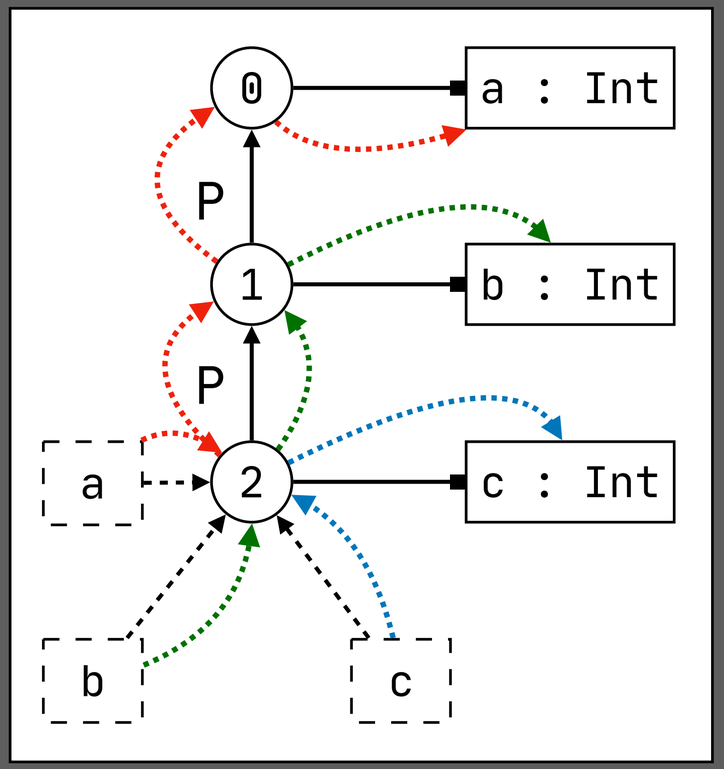
new scope

scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

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
```

```
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

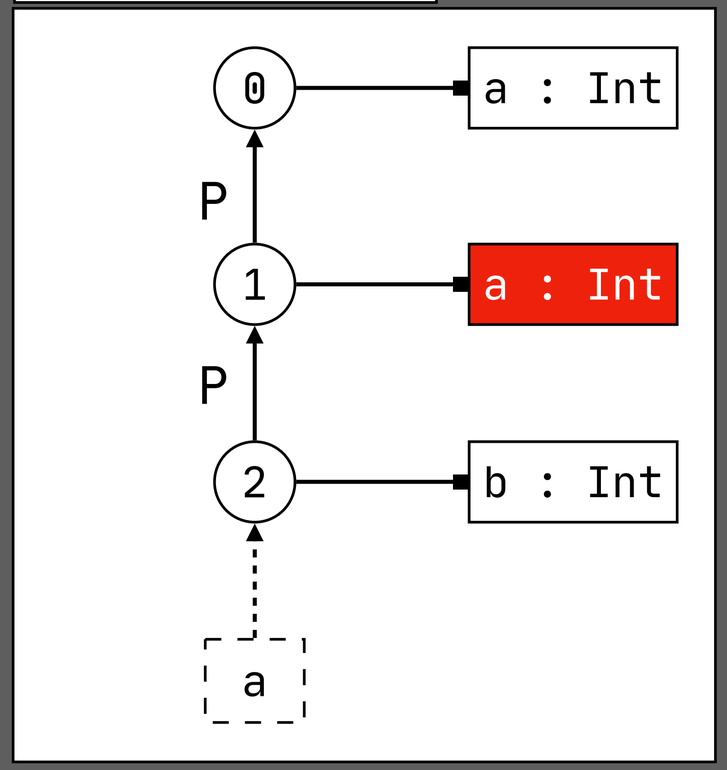
scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

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
```

```
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

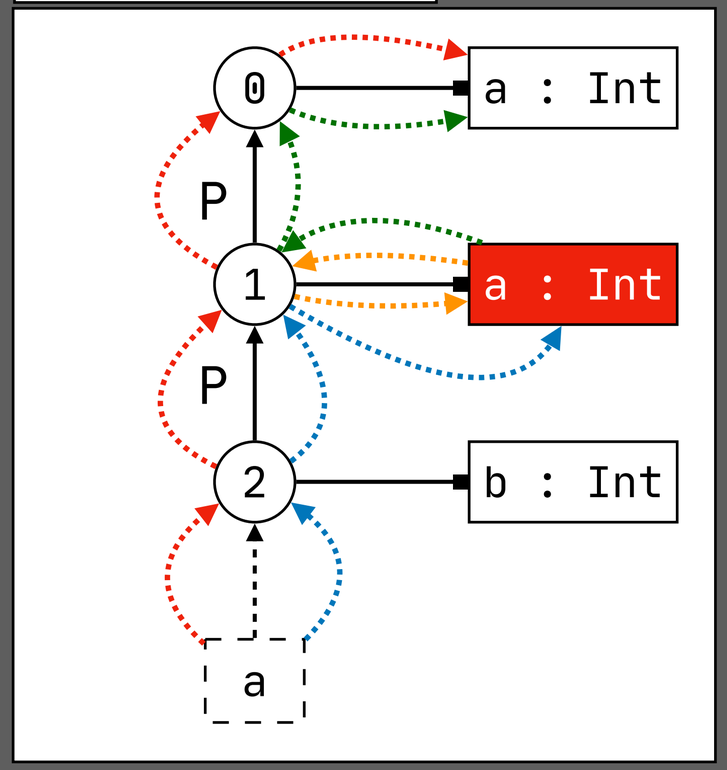
scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

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
```

```
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

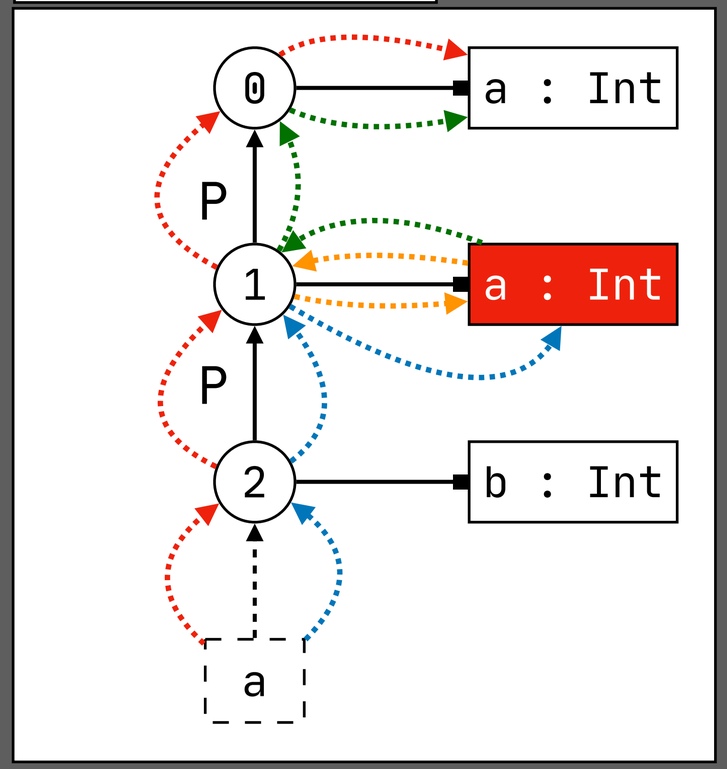
```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

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
```

```
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

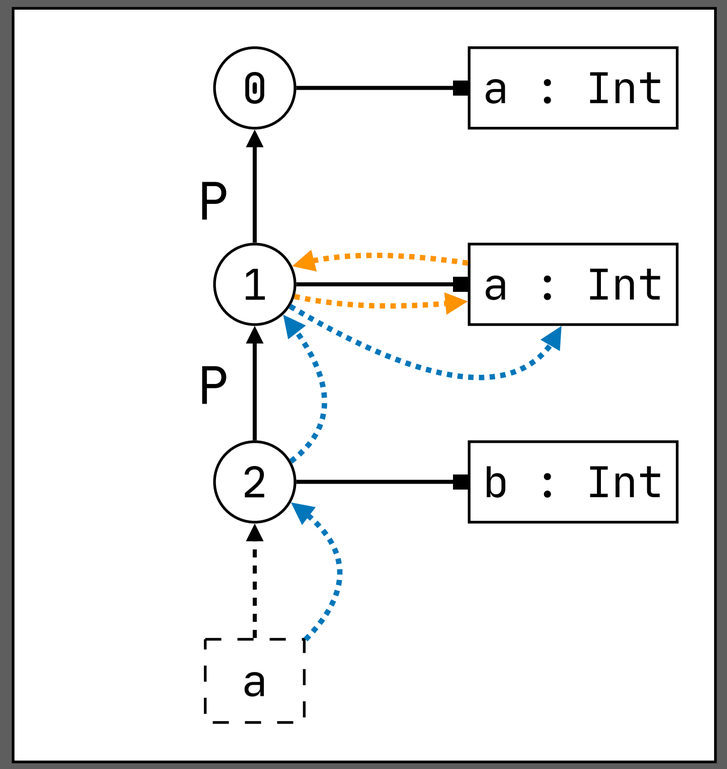
scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P* min $ < P
```

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
```



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

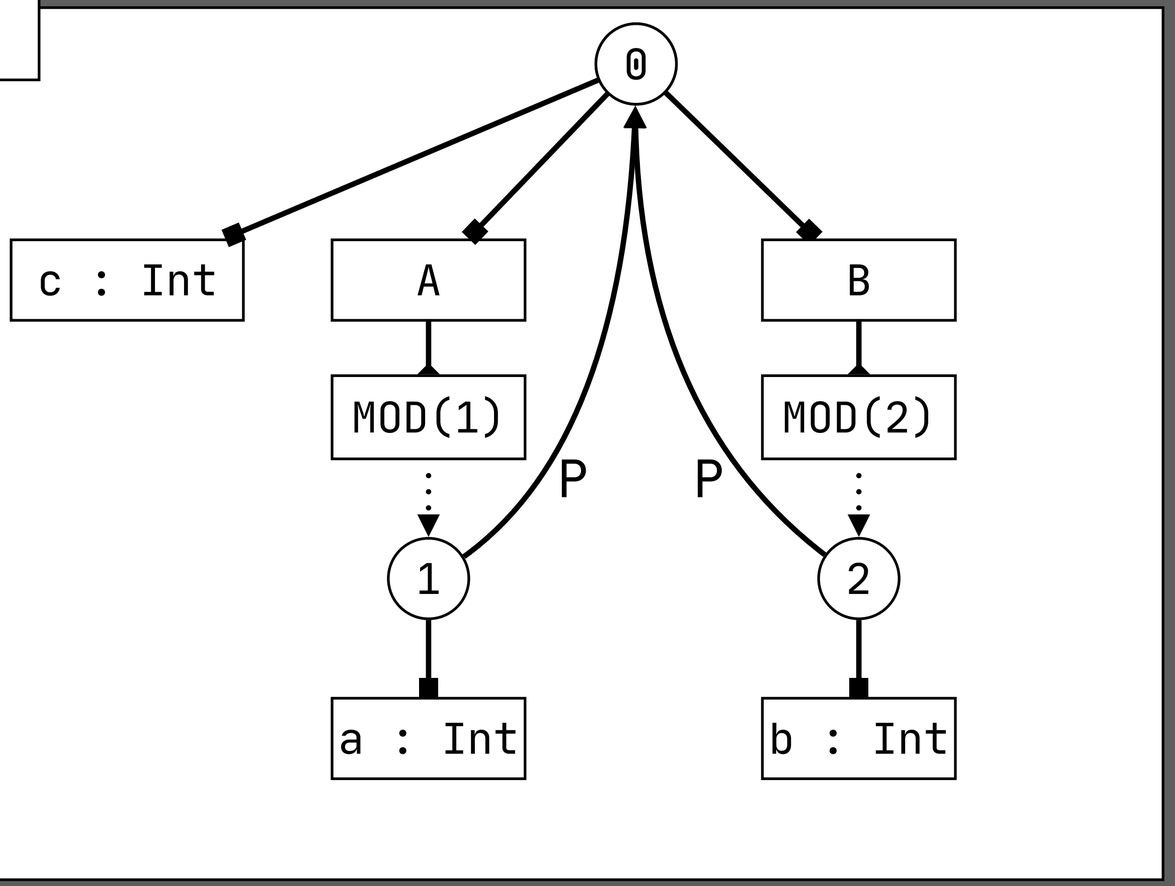
```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```

```
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

```
signature
  namespaces
    Mod  : string
```



Resolving Import

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

scope as type

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

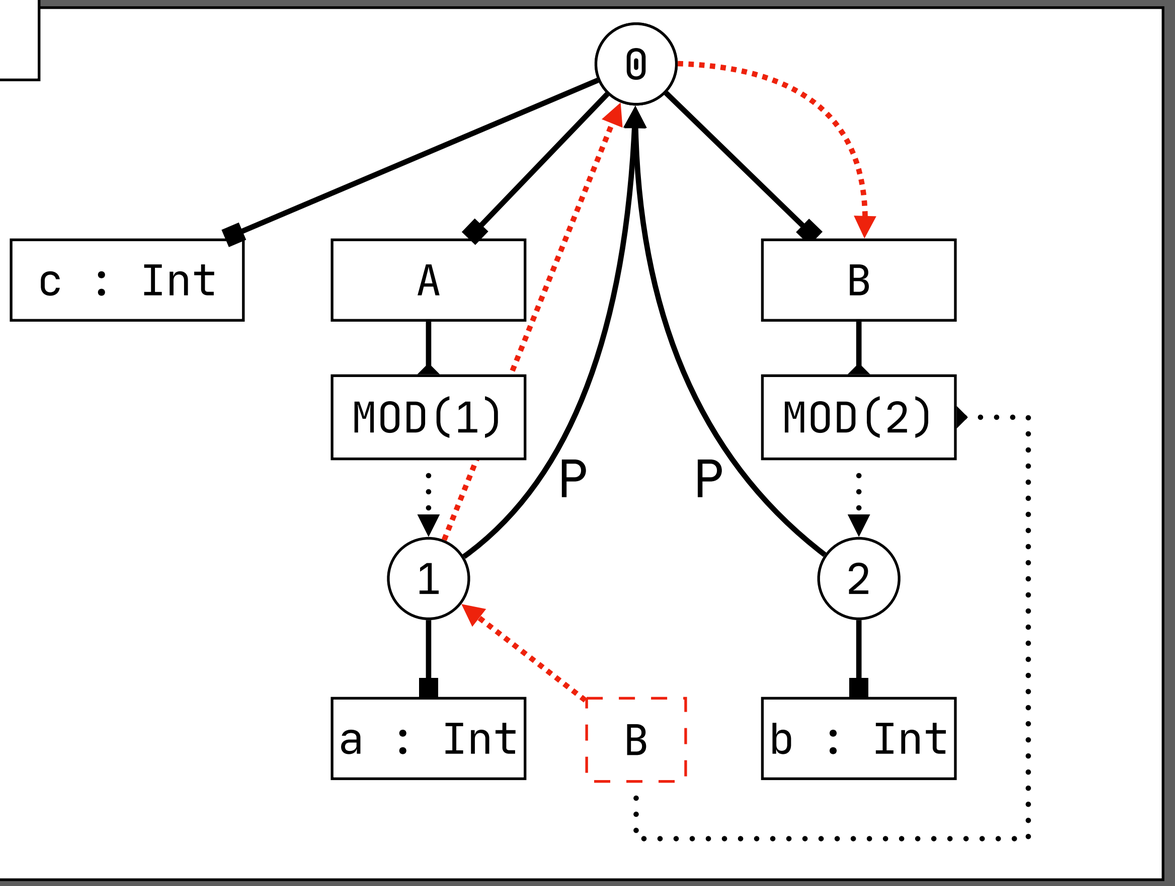
```
rules

  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

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P*
      min $ < I, $ < P, I < P
```



Import Edge

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

scope as type

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

```
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.
```

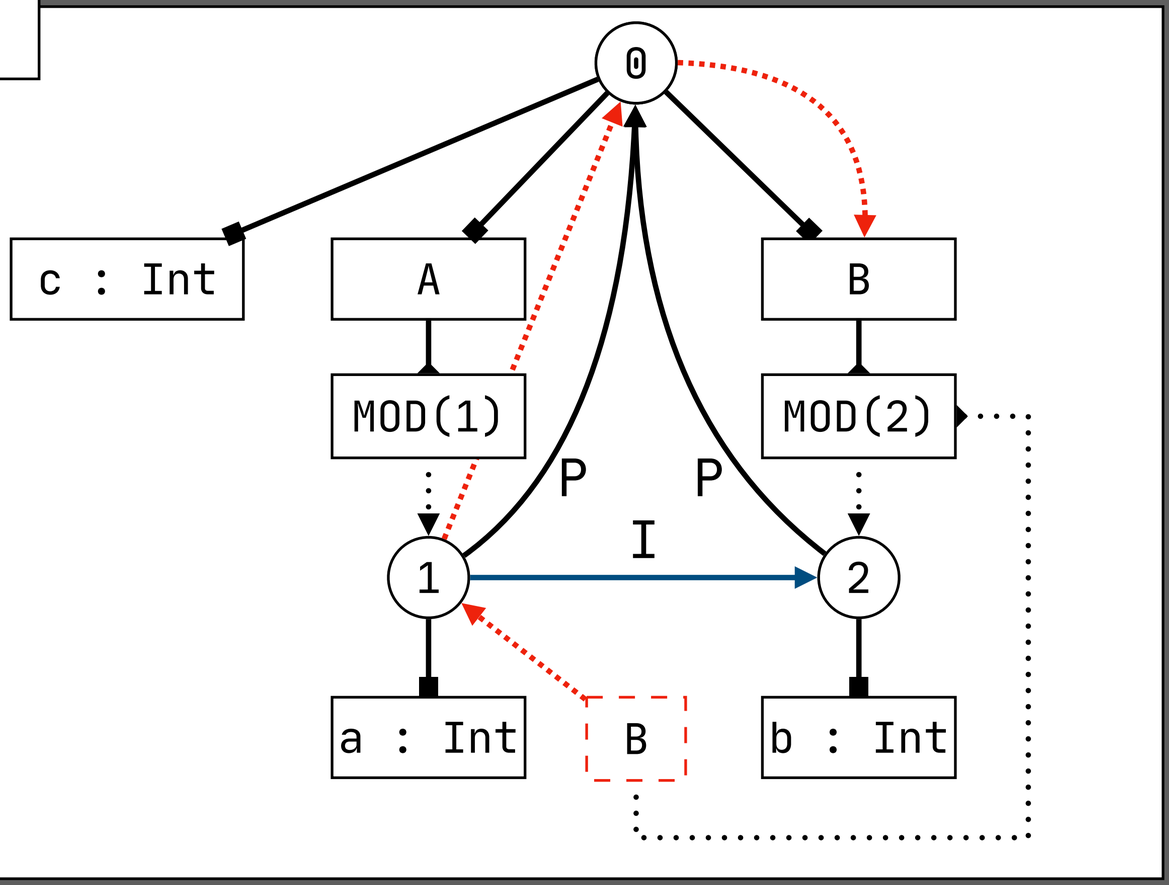
lexical scope

scope as type

resolve import

import edge

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P*
      min $ < I, $ < P, I < P
```



Resolving through Import Edge

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

scope as type

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

```
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

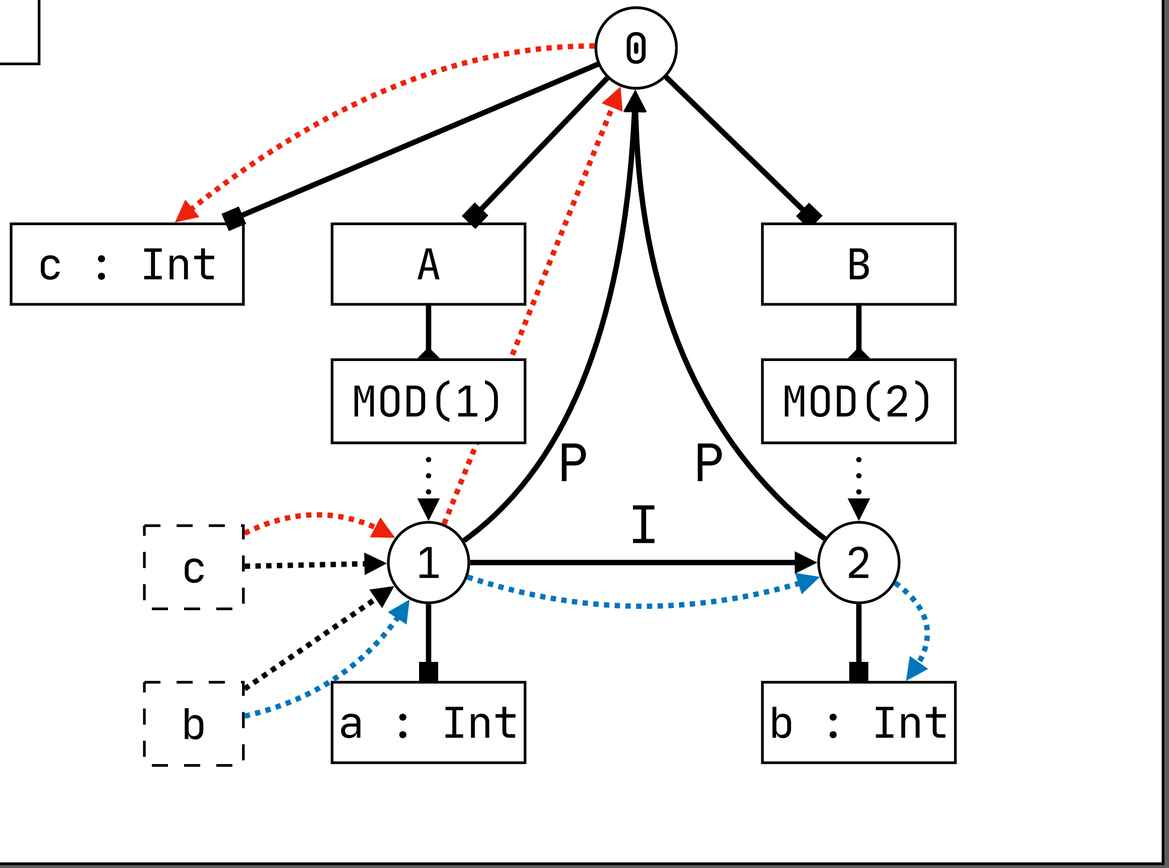
scope as type

resolve import

import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges



Import vs Parent

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

scope as type

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

```
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

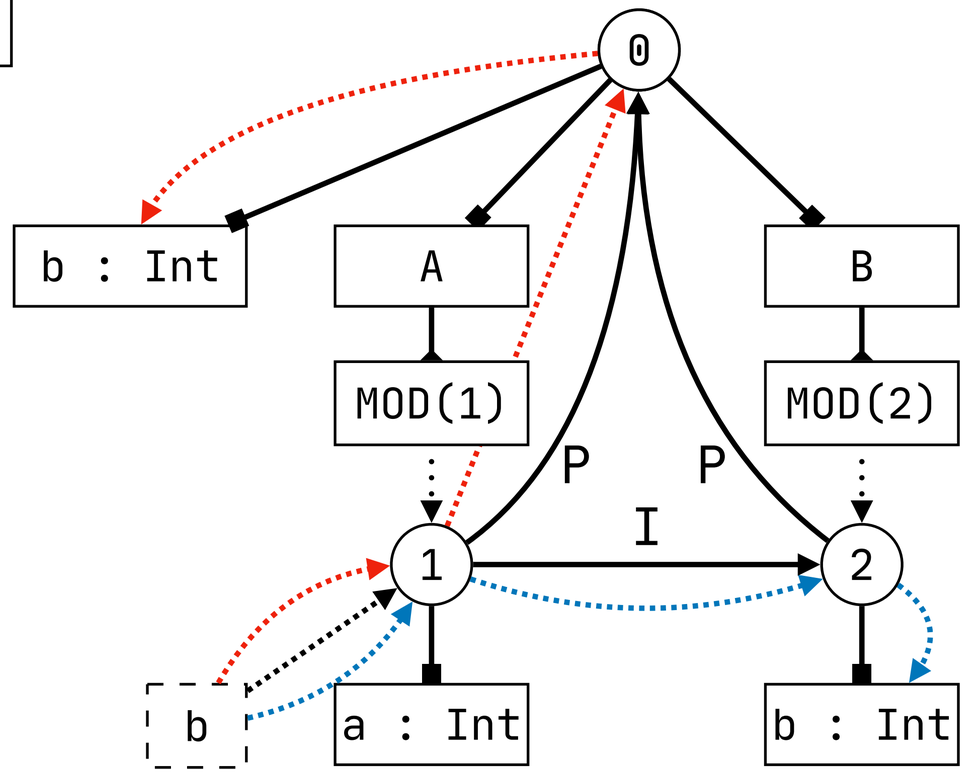
import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

prefer import

prefer blue path over red path



Mutual 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

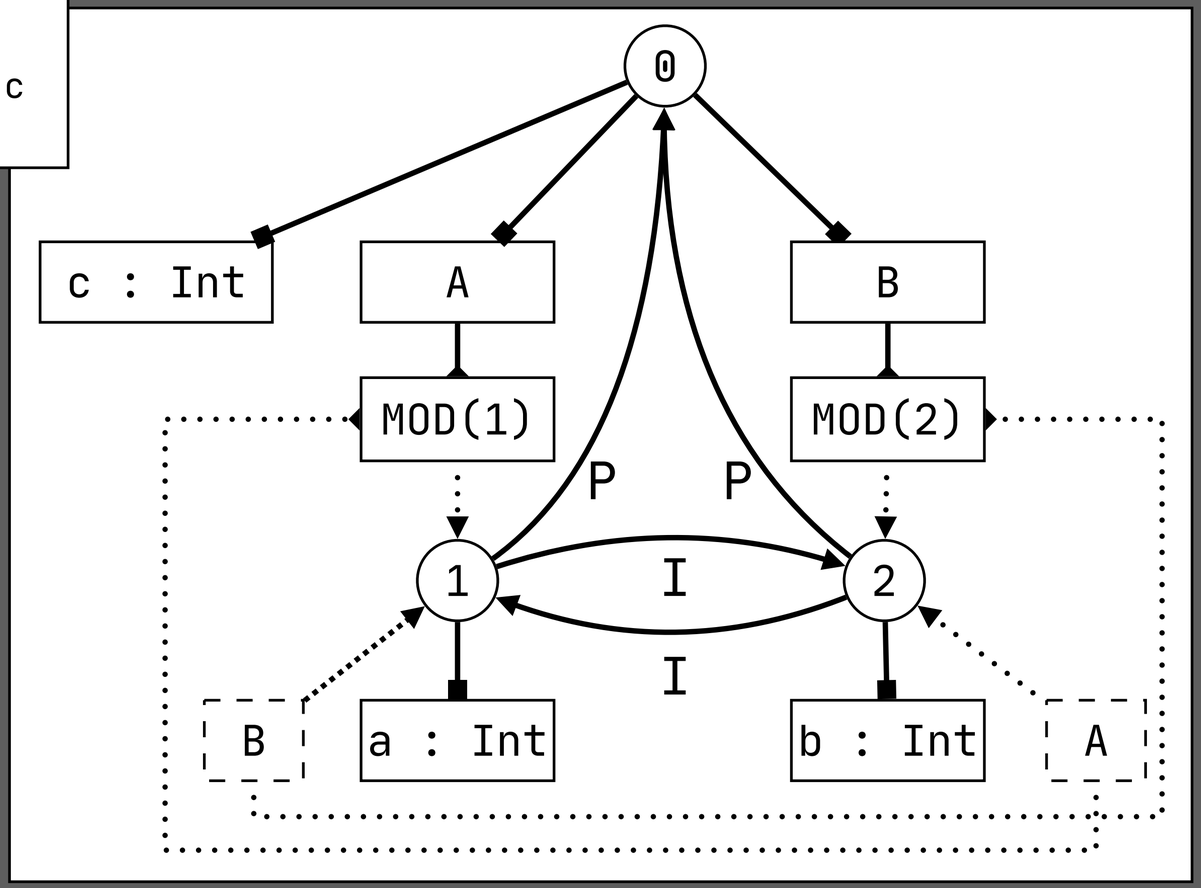
import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

import after parent

prefer import

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



Mutual Imports

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

scope as type

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

```
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

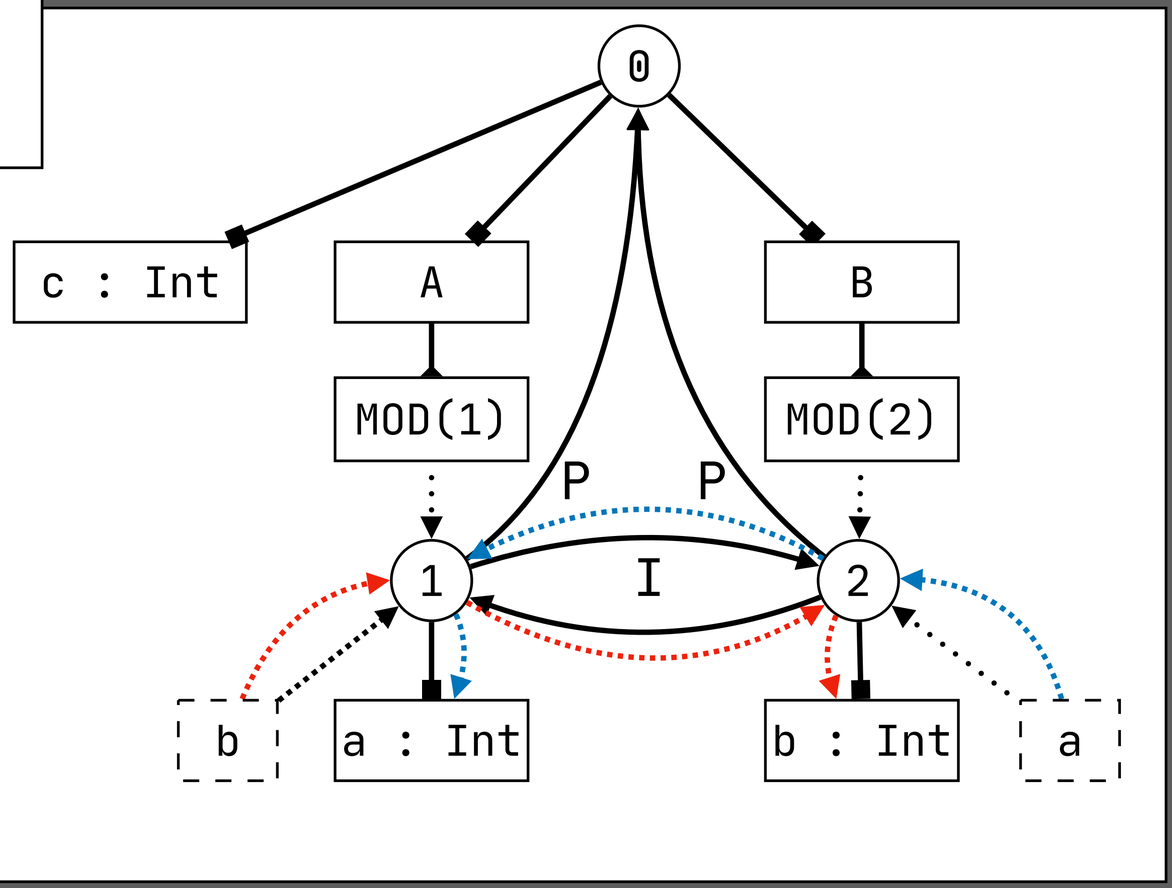
resolve import

import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

prefer import



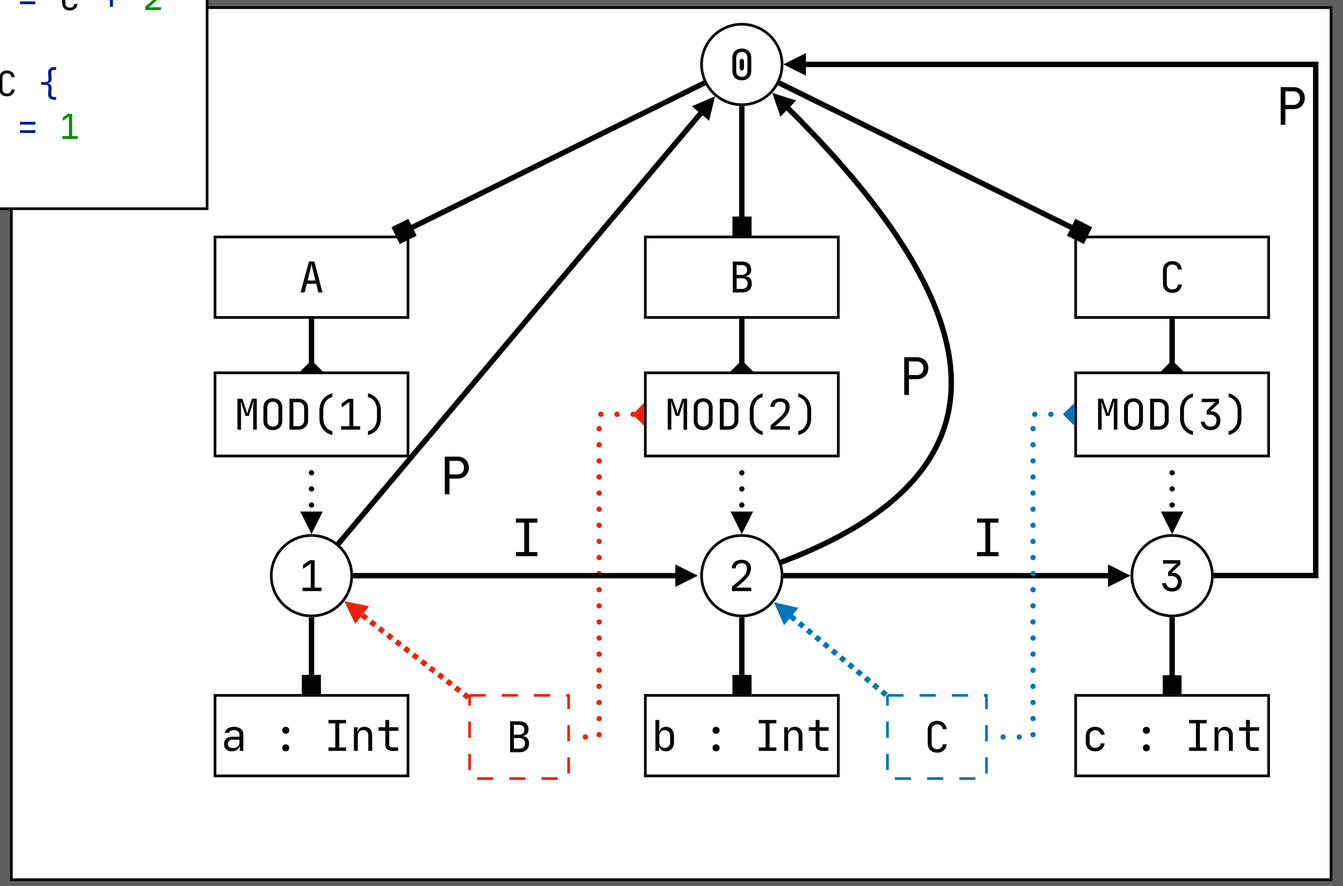
Transitive Import

```
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.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

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



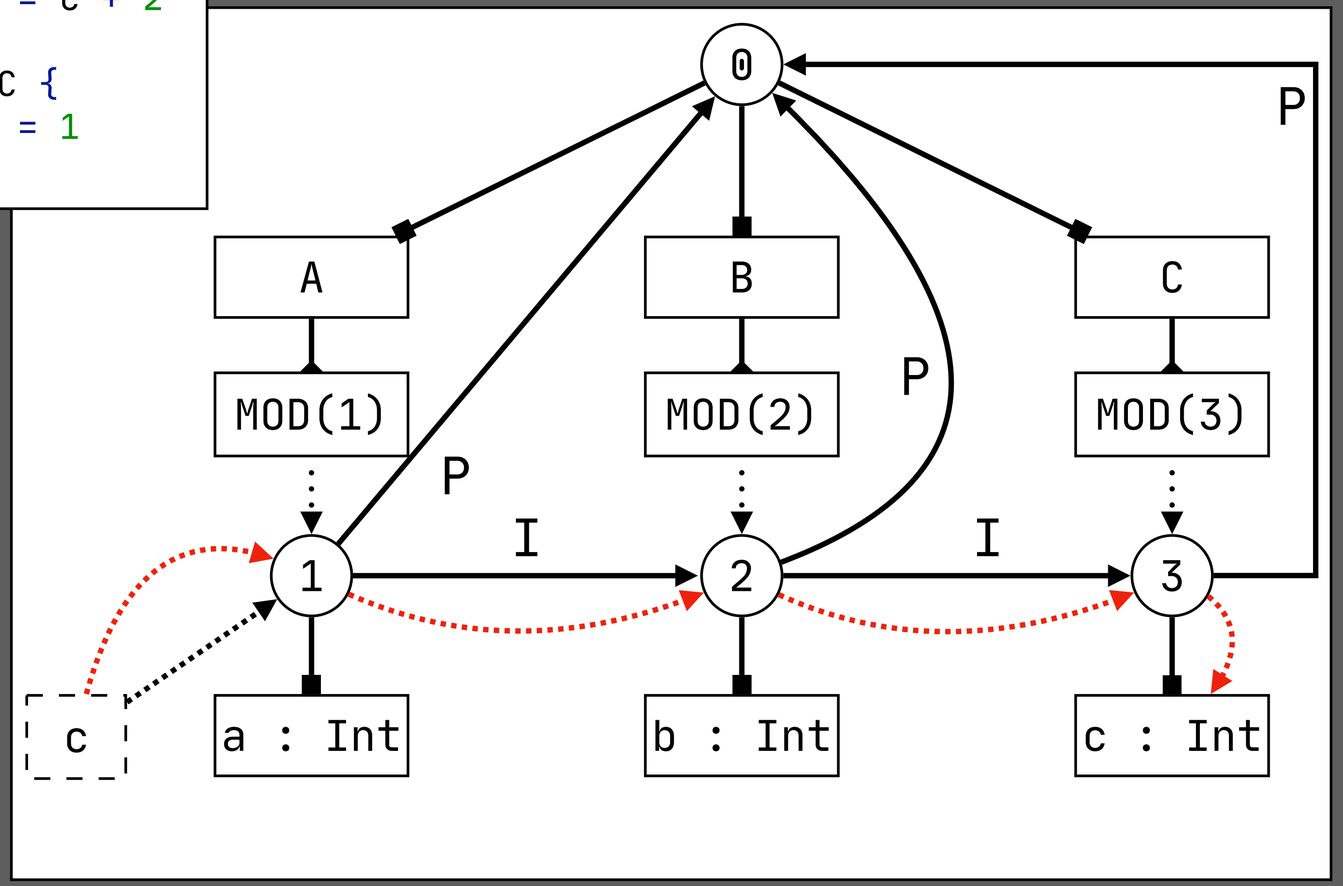
Transitive Import

```
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.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

```
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)

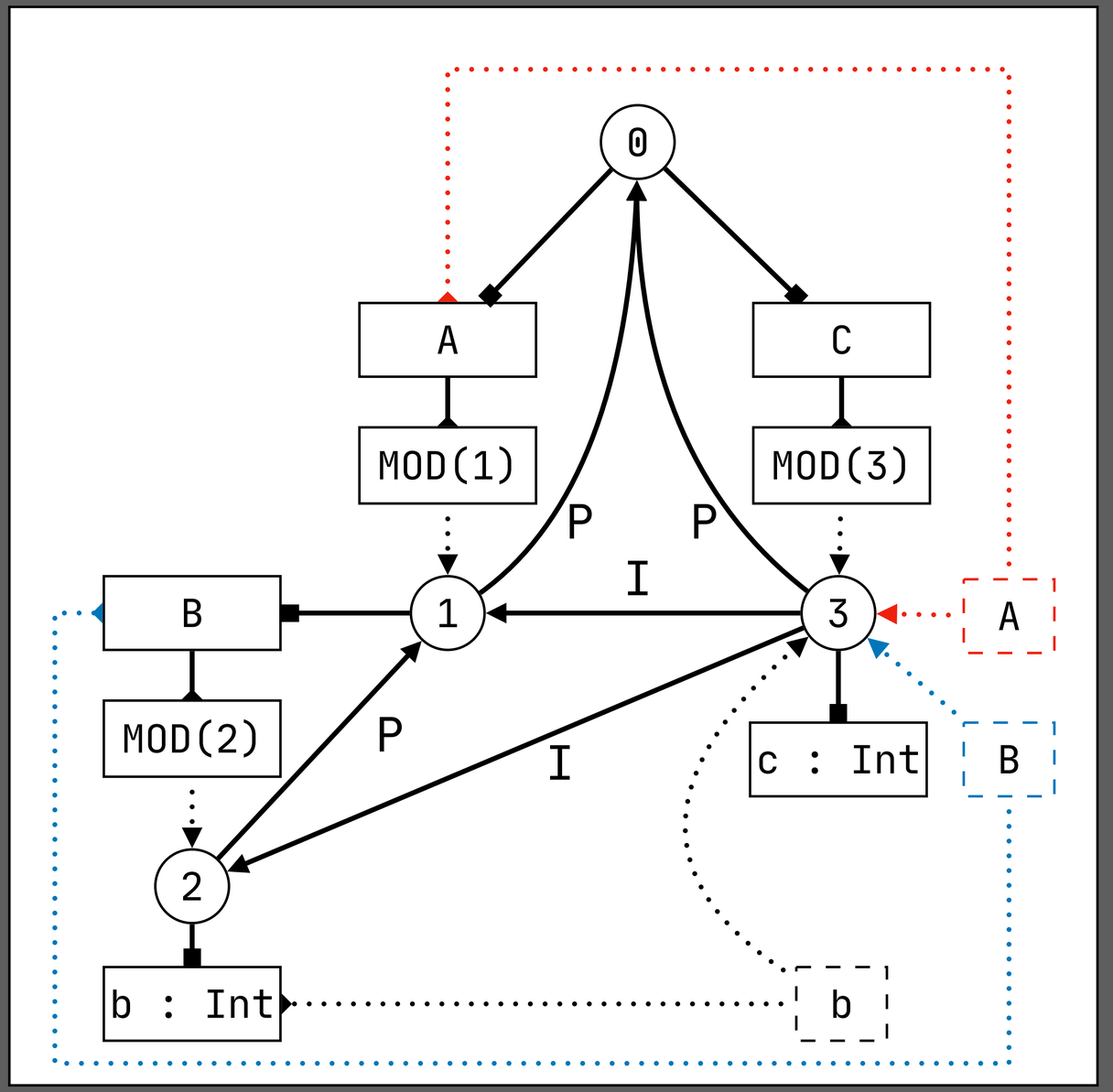
Nested Modules

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

```
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.
```

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P* I*
      min $ < I, $ < P, I < P
```

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

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



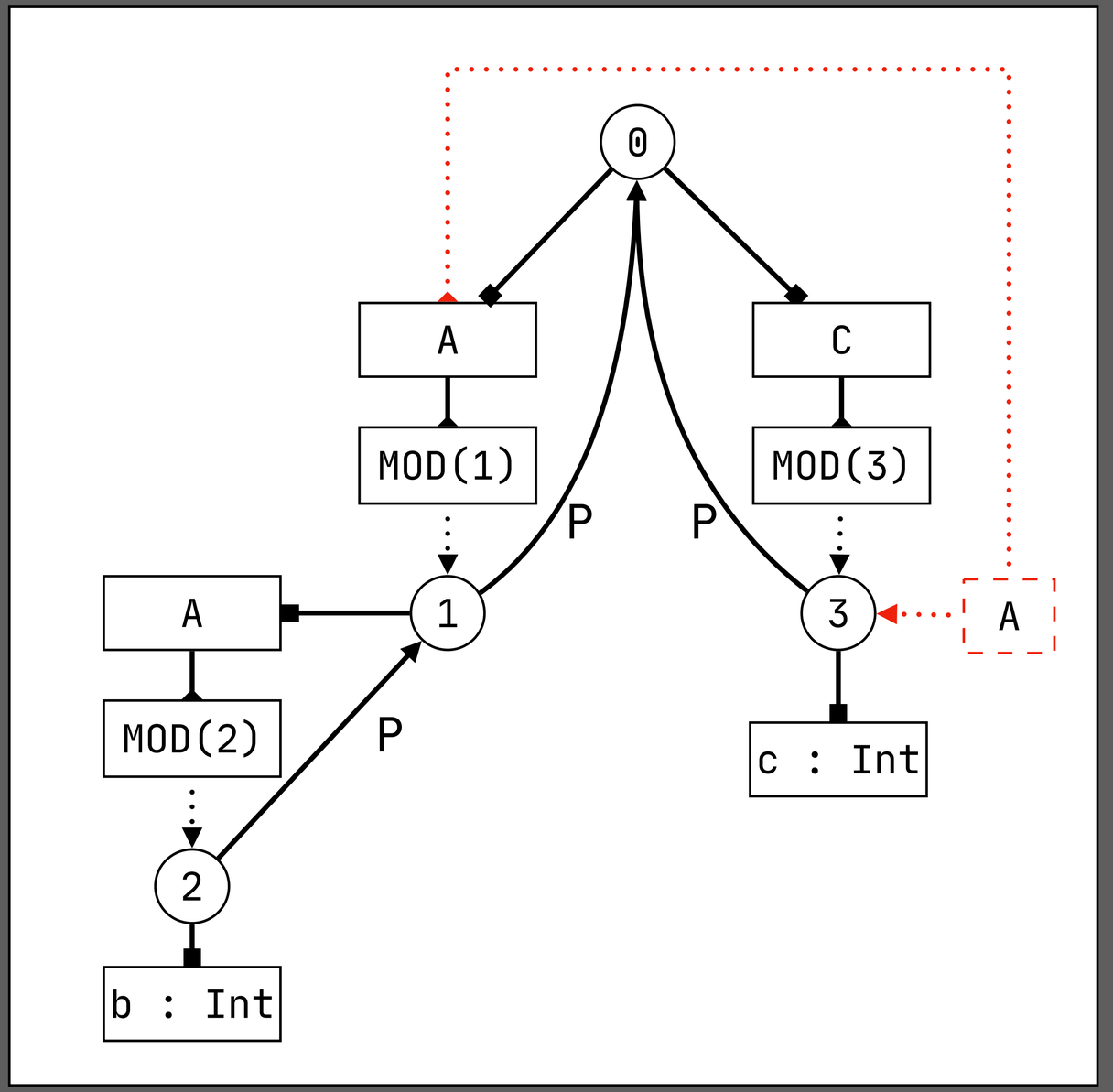
Changing Result of Query

```
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.
```

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P* I*
      min $ < I, $ < P, I < P
```

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

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



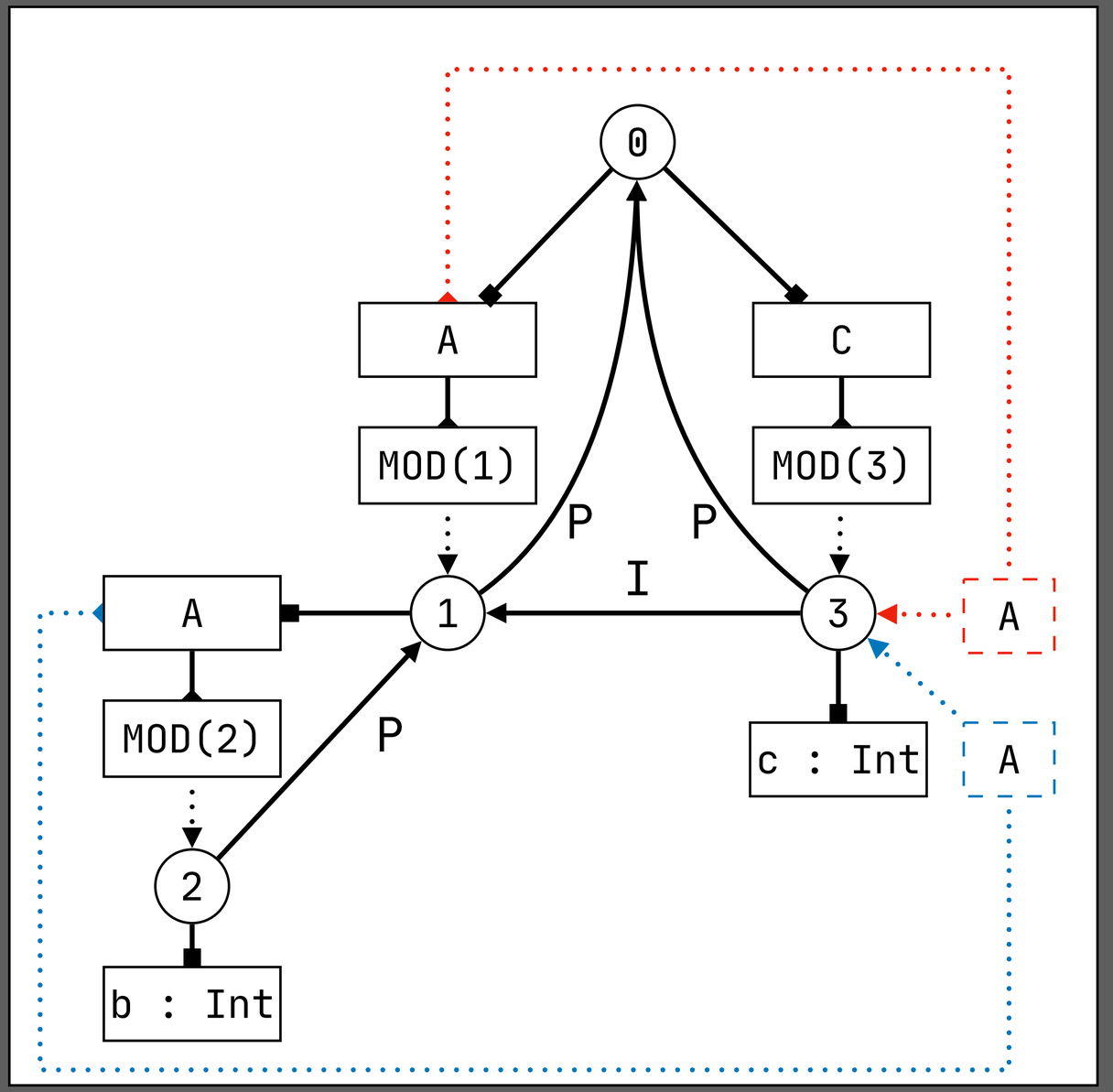
Changing Result of Query

```
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.
```

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P* I*
      min $ < I, $ < P, I < P
```

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

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



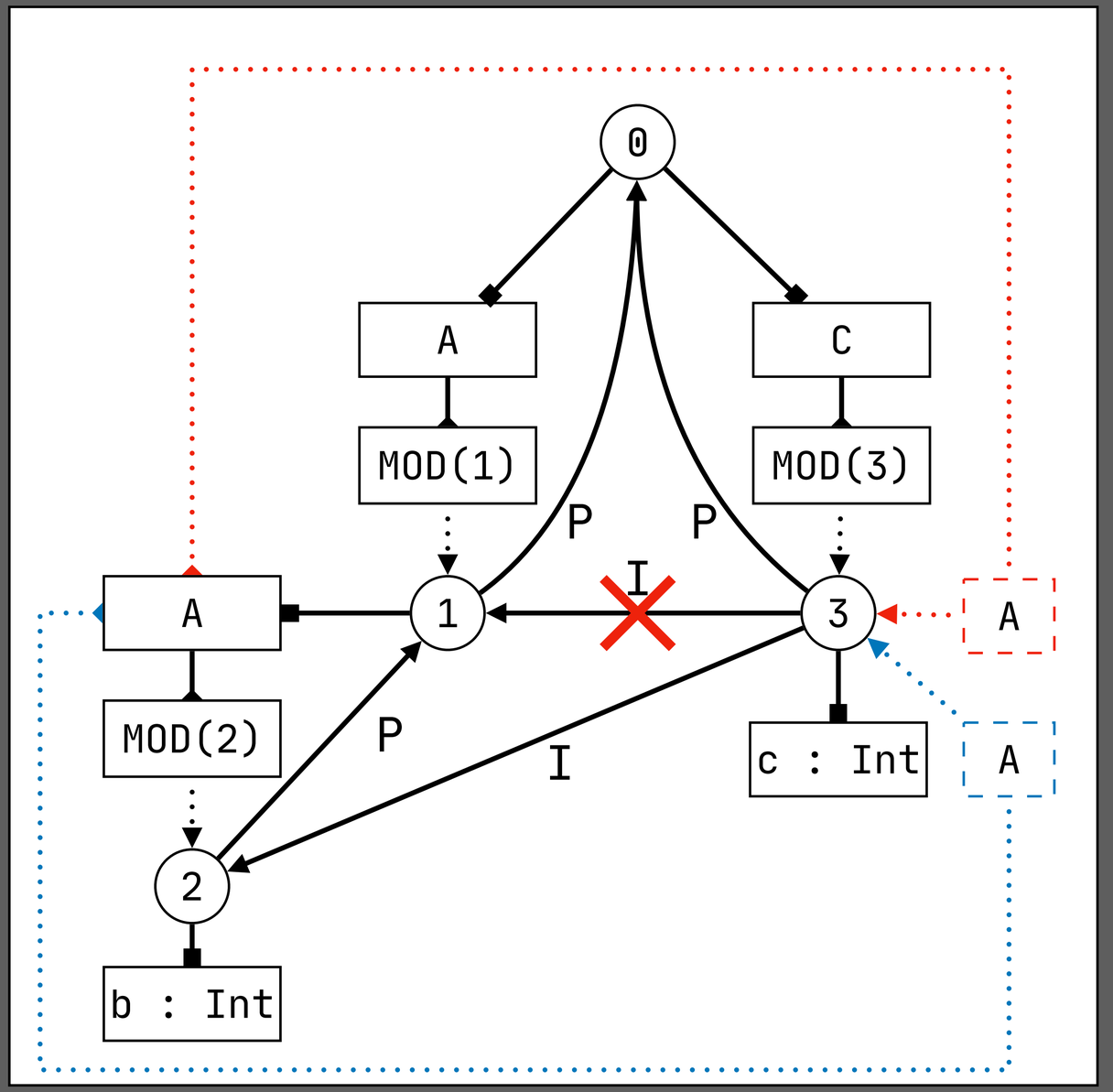
Changing Result of Query

```
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.
```

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P* I*
      min $ < I, $ < P, I < P
```

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

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

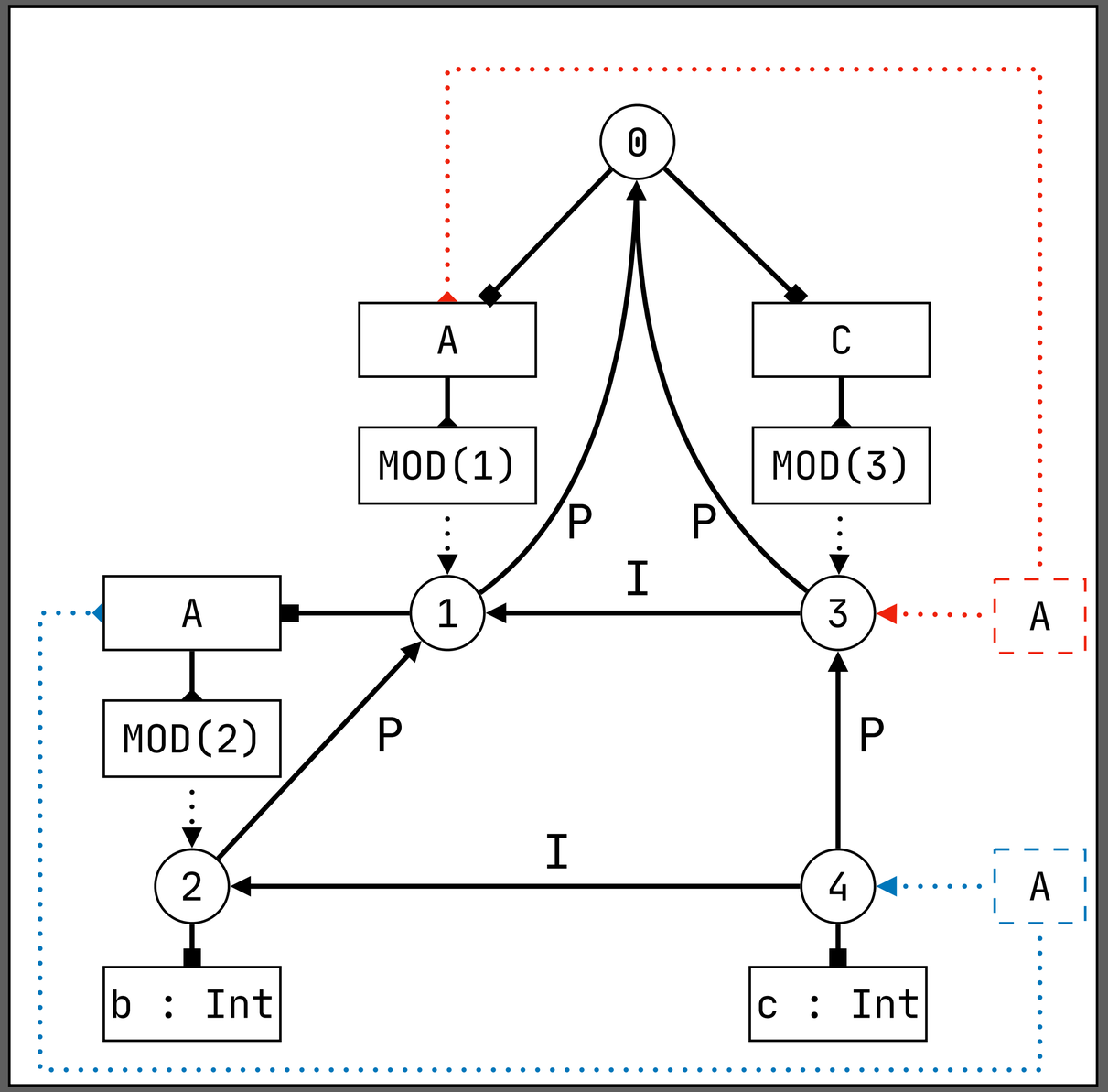


Alternative Encoding: Scoped Imports

```
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
}
```

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P P* I*
      min $ < I, $ < P, I < P
```

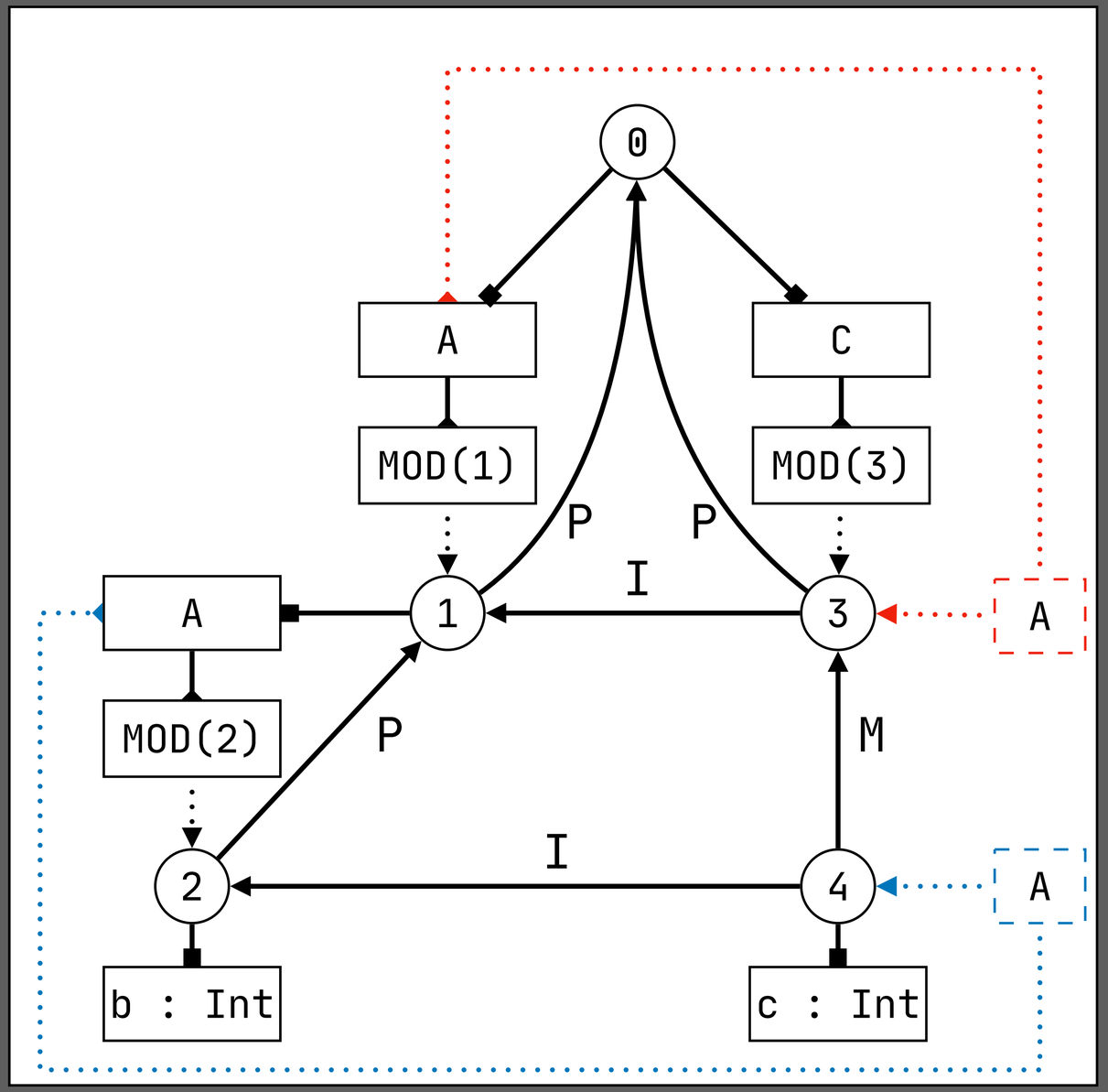


Alternative Encoding: Scoped Imports — M Edge Label

```
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
}
```

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter (P | M) P* (I | M)*
      min $ < I, $ < P, I < P
```



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
}
```

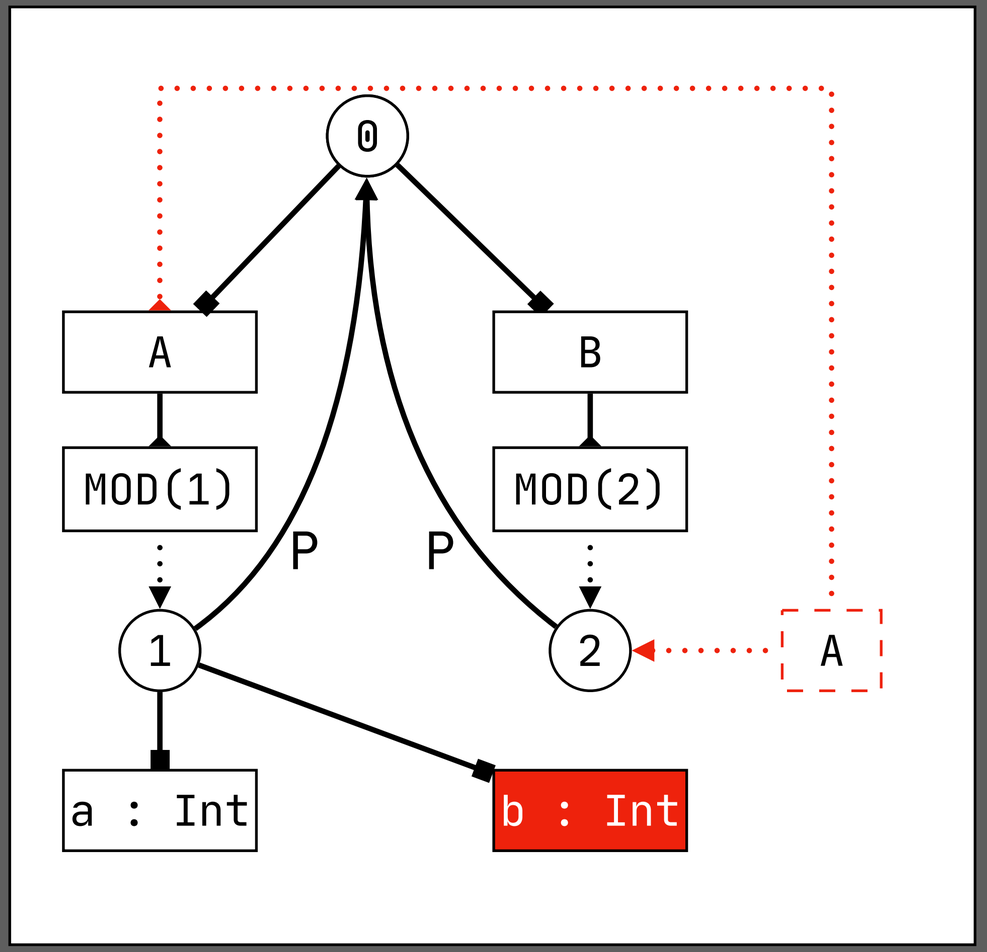
extend remote: `def M.x := e`
extend module `M` with declaration of `x`

```
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
```

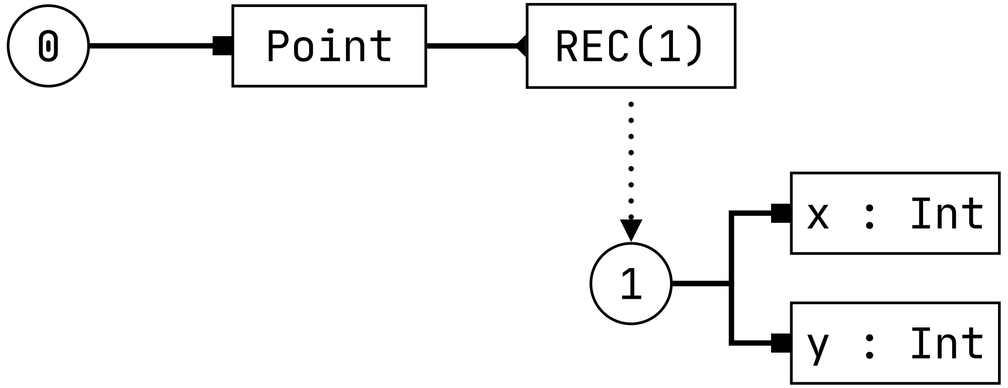
Record Type: Scope as Type

```
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
```

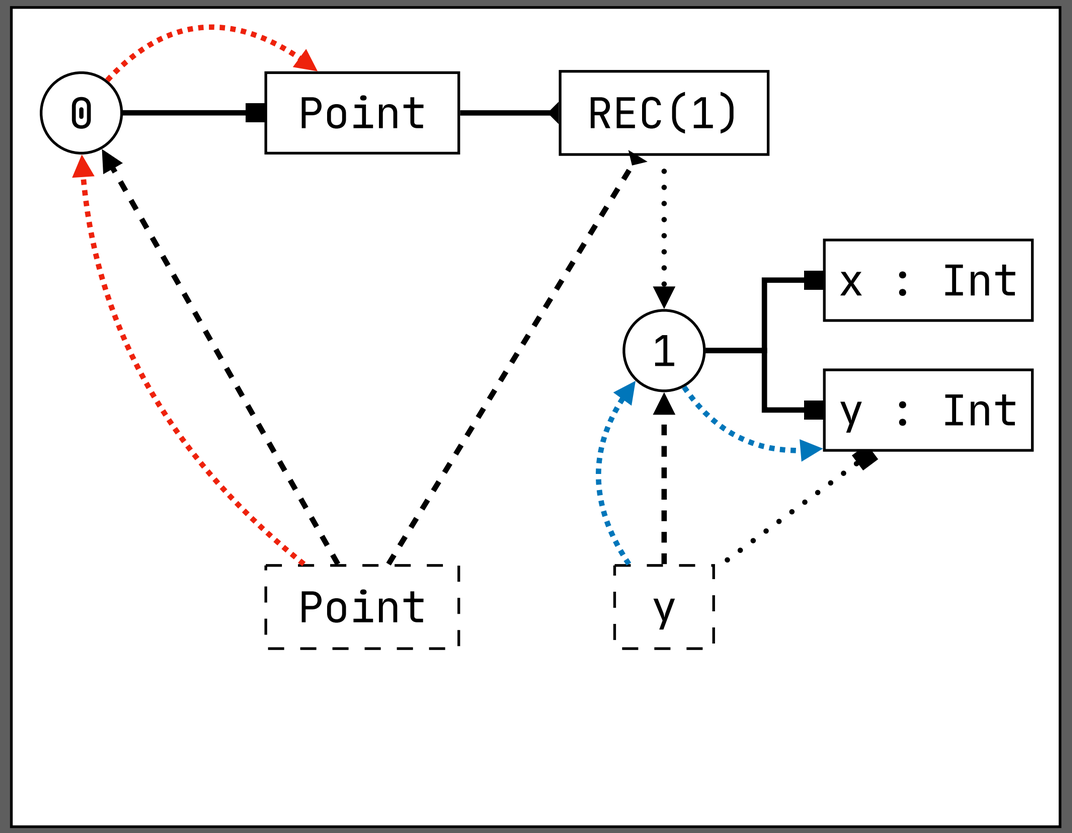


Record Construction & Initialization

```
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
```

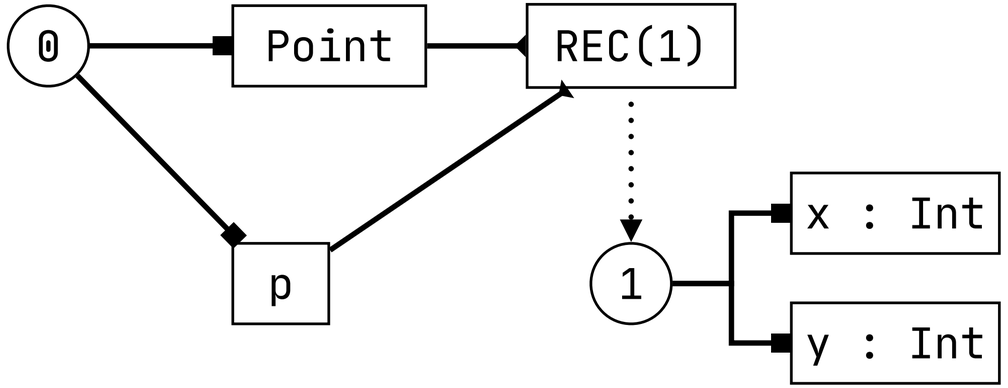


Type-Dependent Name Resolution

```
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
```

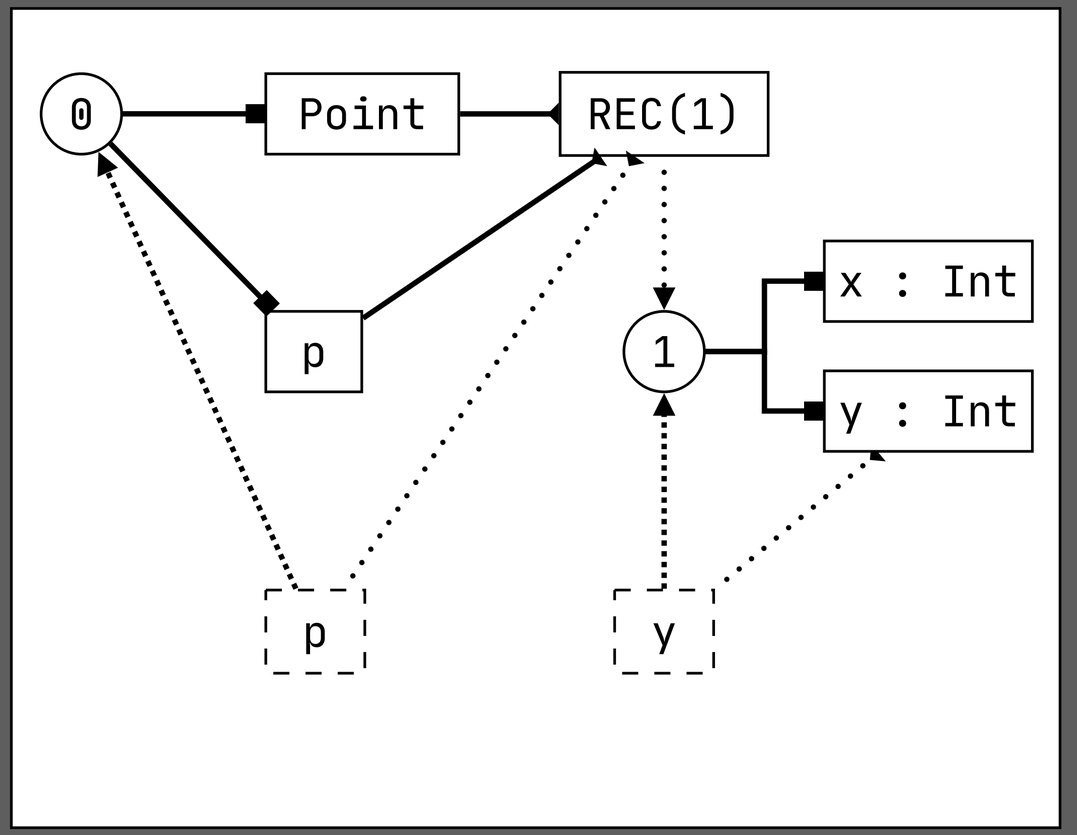


Type-Dependent Name Resolution

```
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
```



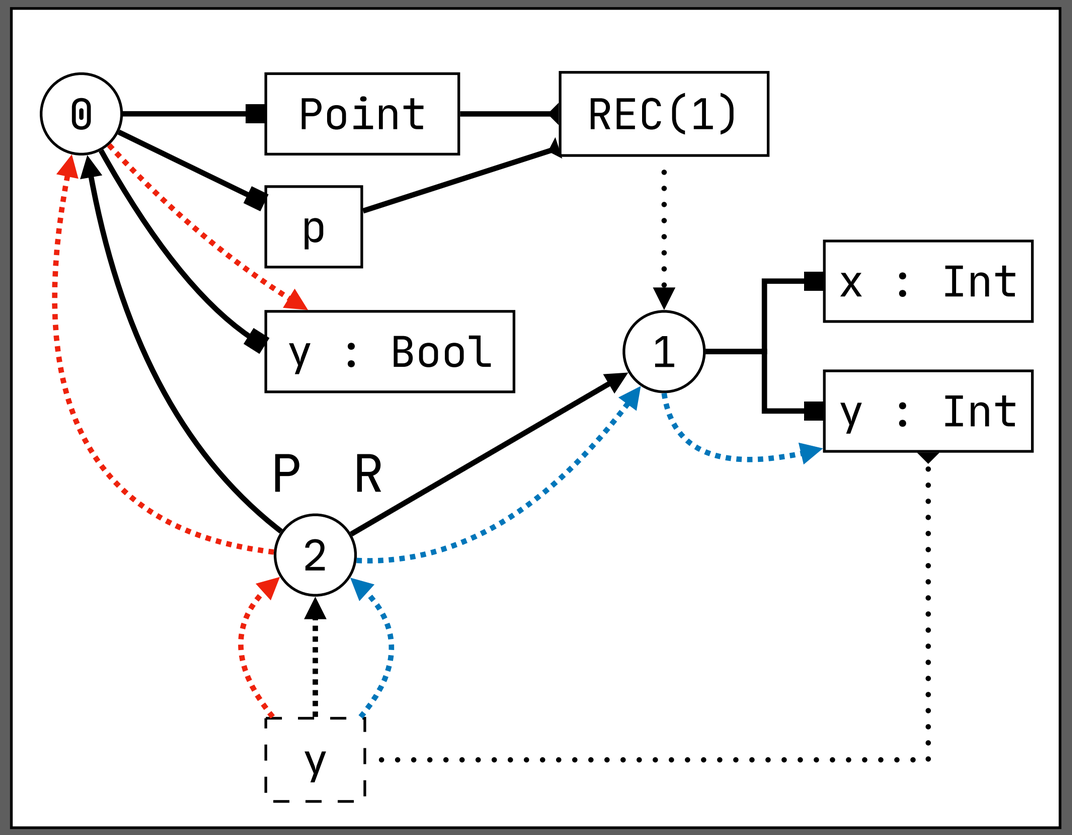
With

```
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.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P* R* min $ < P, R < P
```

```
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

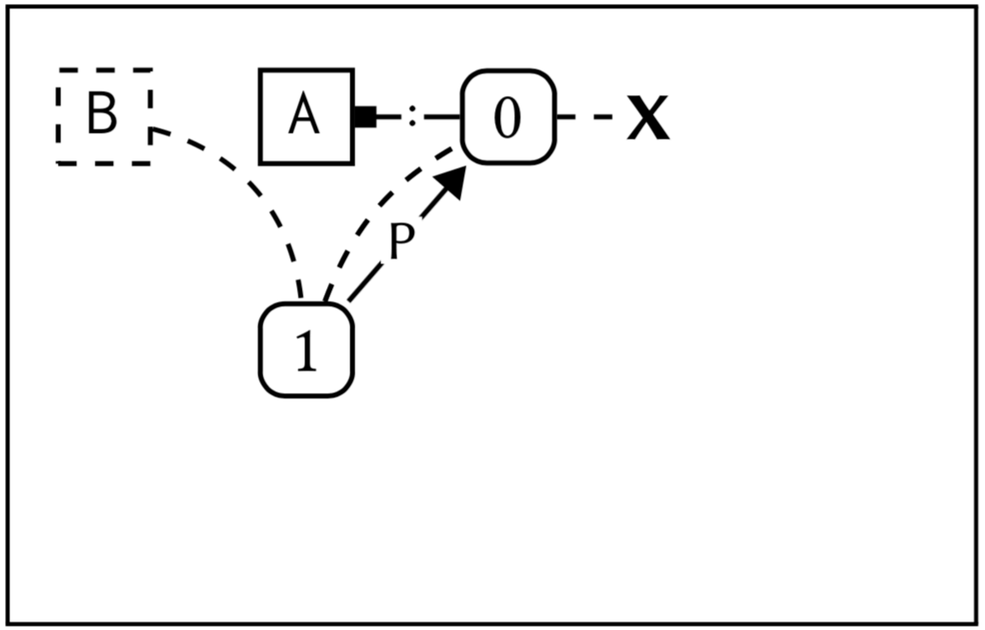
Scope graph construction depends on queries

- Imports require name resolution of module name

When is it safe to query the scope graph?

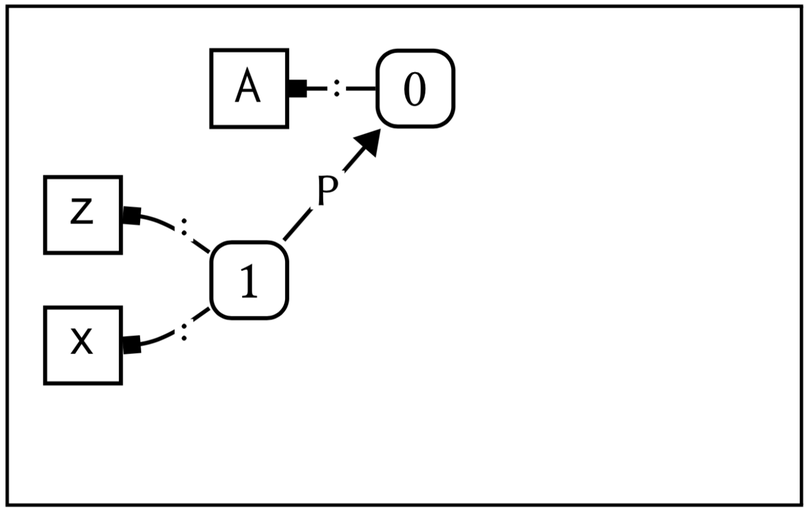
- In what order should type checker perform construction, querying?

A Single Stage Type Checker (Fails)

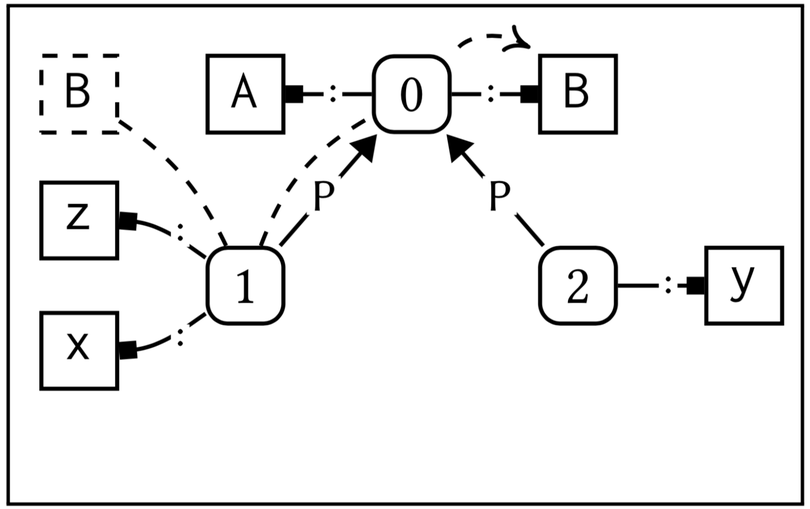



```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

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



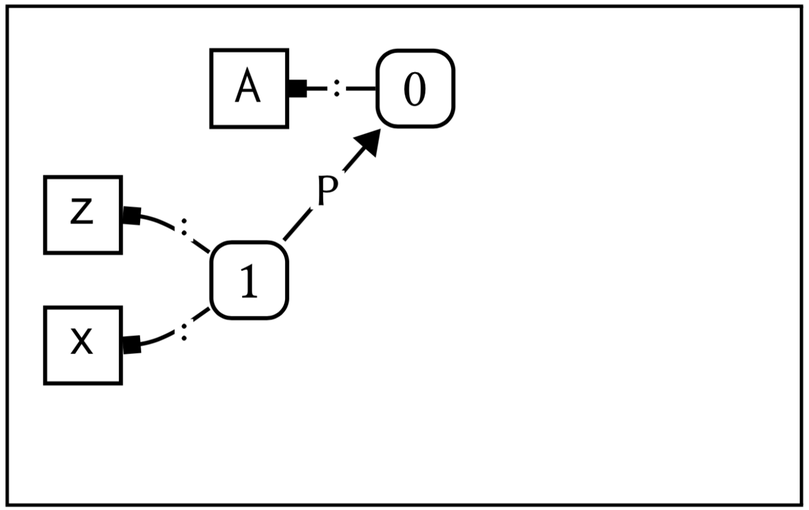
(1)



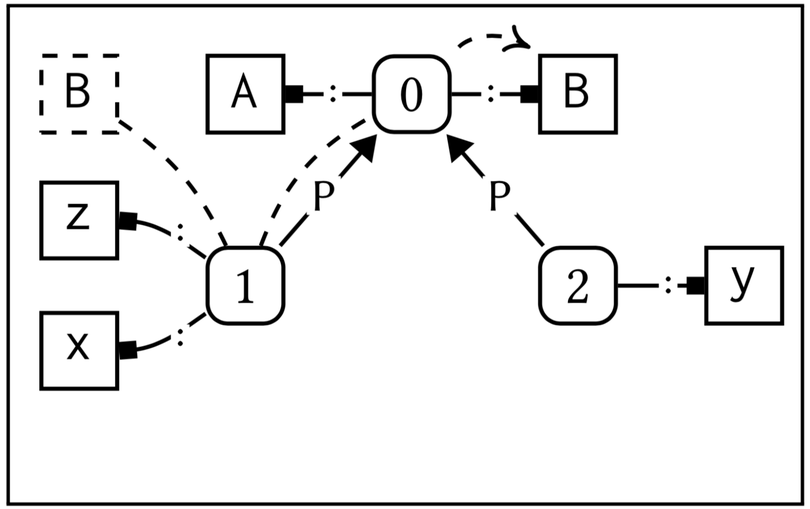
(2)

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

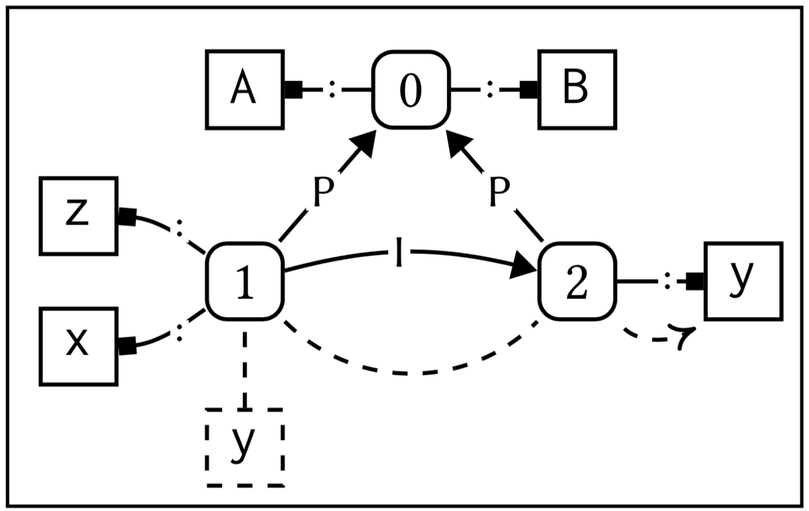
A Two Stage Type Checker: Stage 2 (Check Modules)



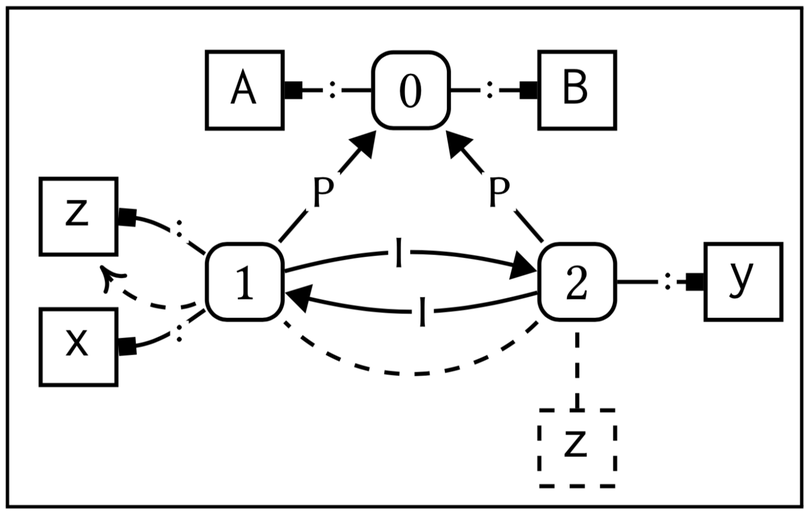
(1)



(2)



(3)

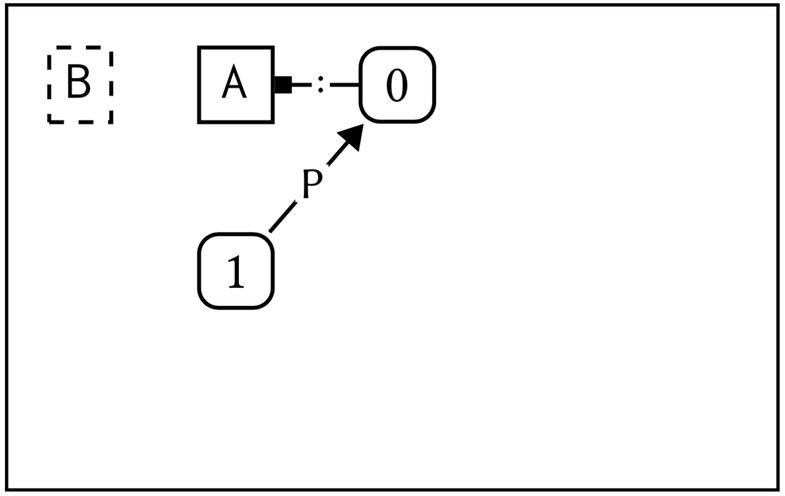


(4)

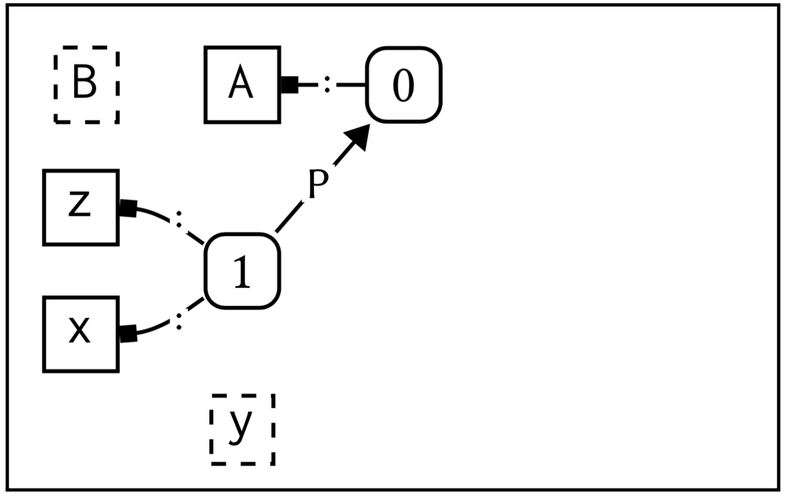
Requires that imports are resolved before variable references

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

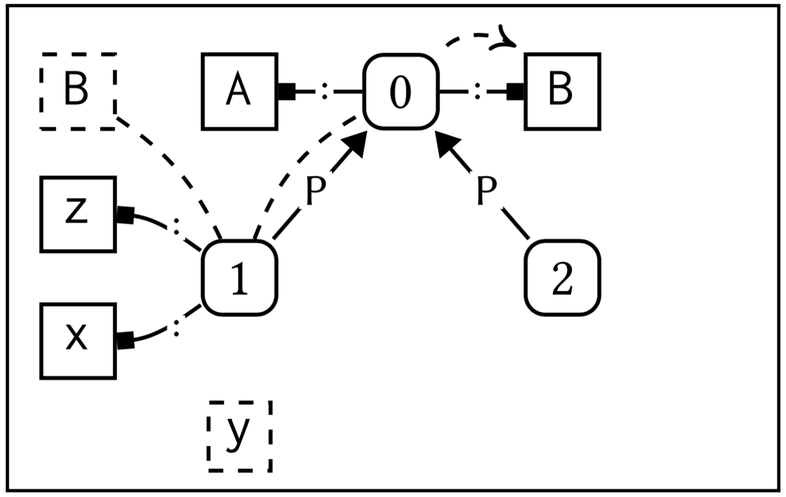
Dynamic



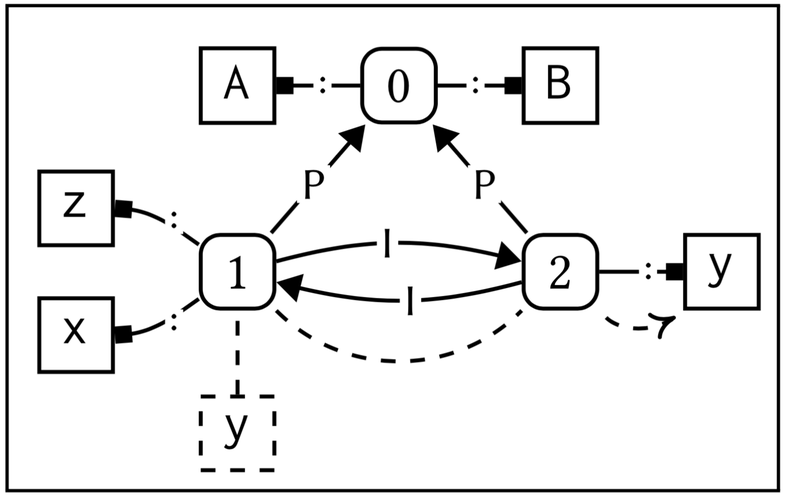
(1)



(2)



(3)

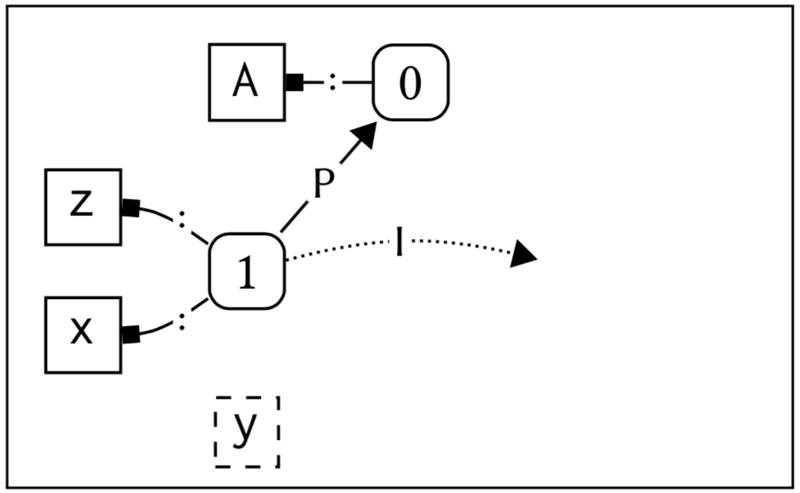


(4)

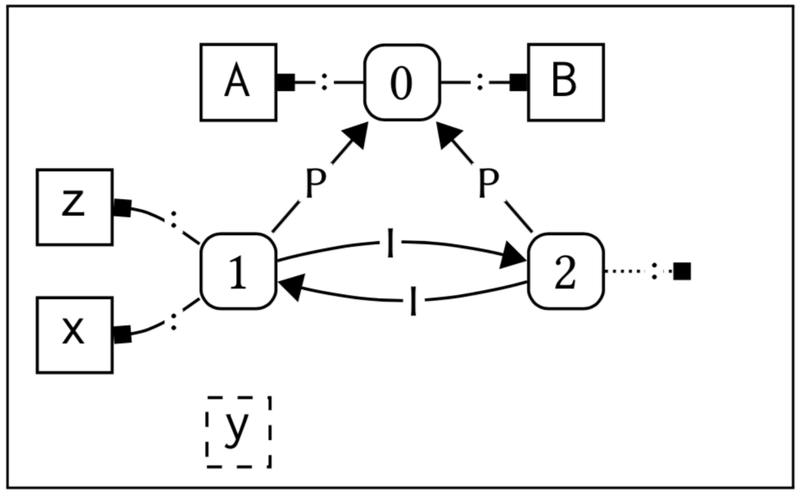
When do we have sufficient information to answer a query?

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

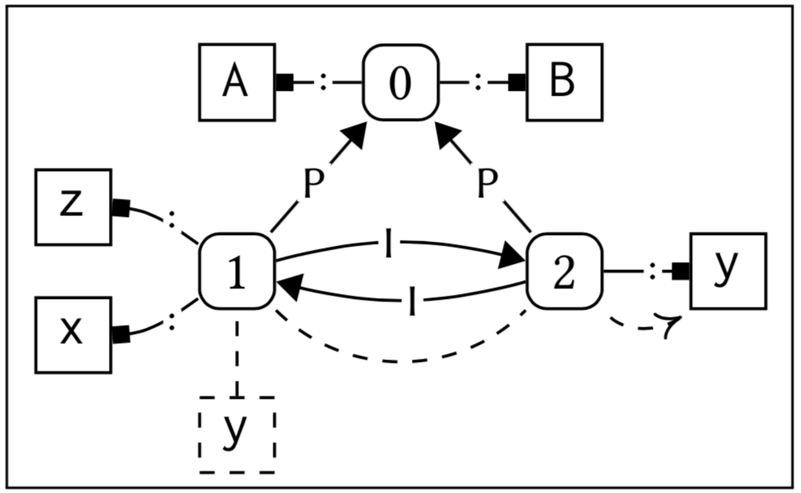
Critical Edges



(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

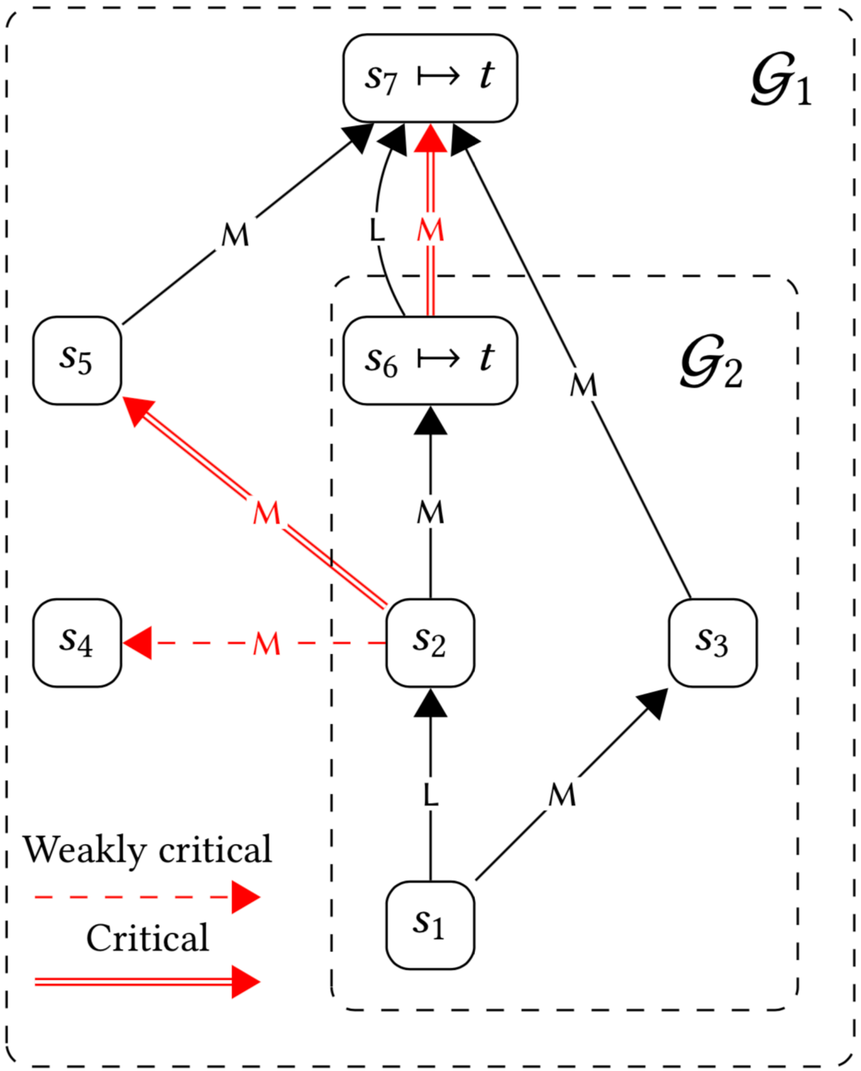


Fig. 11. (Weakly) critical edges for the query $s_1 \xrightarrow{LM*}\!\!\!\!\!\rightarrow_{\mathsf{R}} D$, assuming $t \in D$

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

- 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)

This talk: ESOP’15 + PEPM’16 in Statix

Generics

Scopes as Types

- Van Antwerpen, Bach Poulsen, Rouvoet, Visser. OOPSLA 2018

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