Type Checkers from Declarative Type System Specifications in Statix

Eelco Visser

TU Delft

INRIA | Paris | February 20, 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, …

Context: From Language Design to Language Implementation

Spoofax Language Workbench

- Language designer provides high-level language definition
- Declarative: abstracts from operational implementation details
- Automatically generate implementation from language definition

Meta-languages

- Syntax definition in SDF3
- **Static semantics in Statix**
- Transformation in Stratego
- Dynamic Semantics in DynSem/Dynamix

Type System Specification in Statix

Features

- Constraint-based language with declarative semantics
  - Understand type system without algorithmic reasoning
- Name binding using scope graphs *as part of constraint resolution*
- Implementation: interpret specification as type checker
  - Sound wrt declarative semantics
  - Scheduling of constraint resolution based on language independent principles

Publications

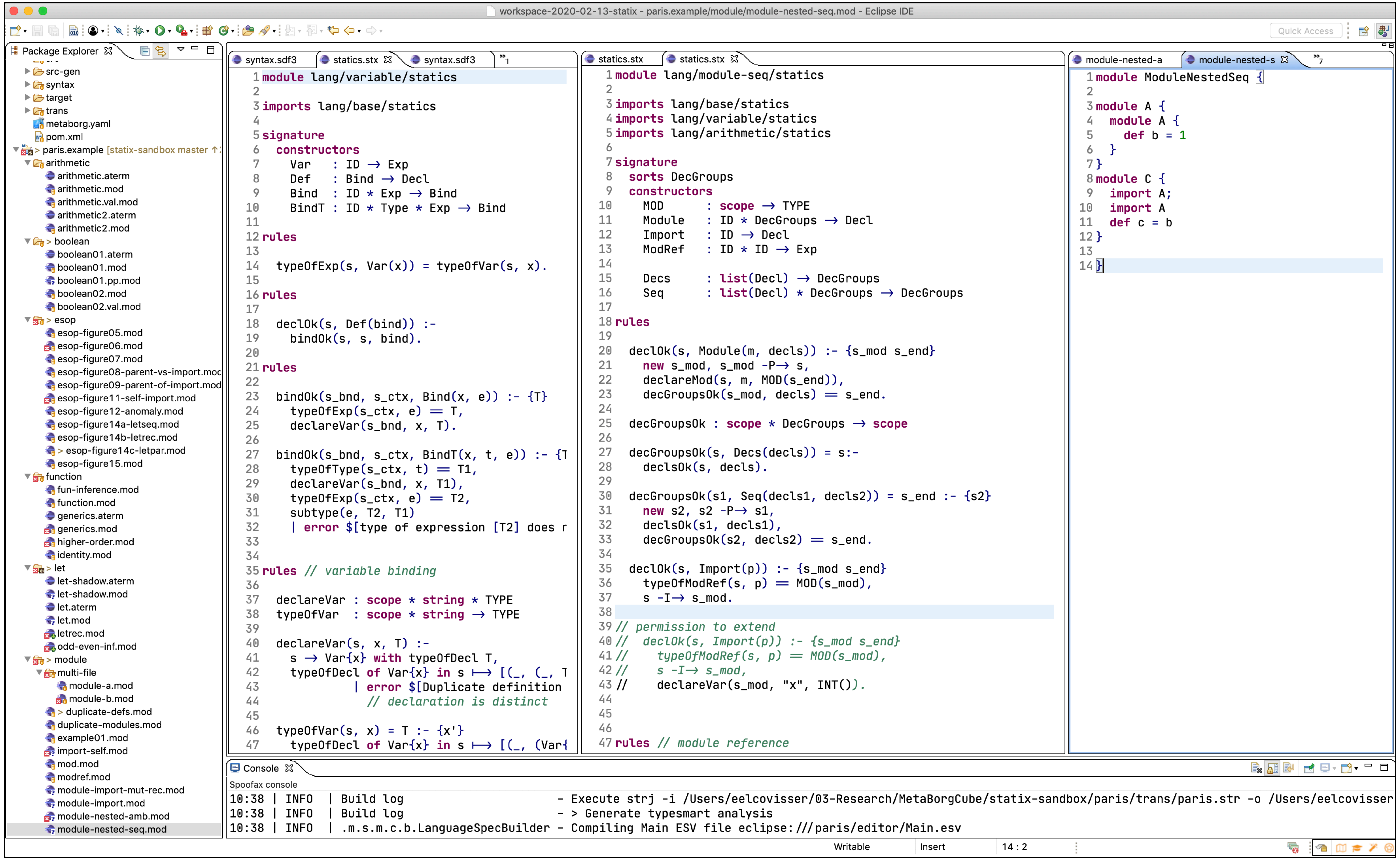
- **Scopes as Types**. Van Antwerpen, Bach Poulsen, Rouvoet, Visser. OOPSLA 2018
- **A constraint language for static semantic analysis based on scope graphs.** Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, Guido Wachsmuth. PEPM 2016
- **A Theory of Name Resolution**. Pierre Néron, Andrew P. Tolmach, Eelco Visser, Guido Wachsmuth. ESOP 2015

Outline

Statix by example

- Concrete and abstract syntax
- Type predicates
- Declaring and resolving names
- Lexical scope
- Scopes as types
- Modules and imports
- Incompleteness (by example)
- Scheduling queries and critical edges
- Permission to extend

Experiments on Demand



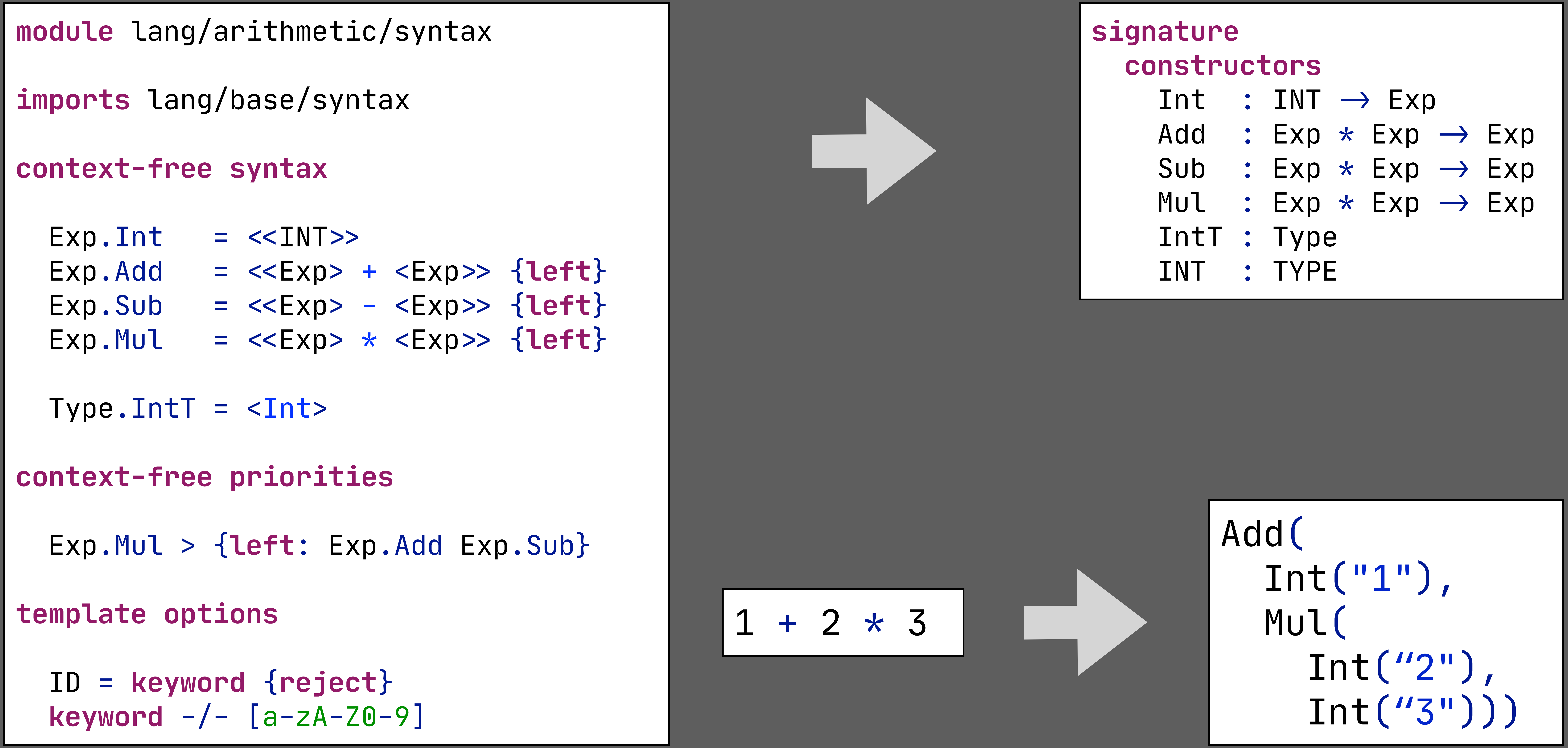
```
module lang/variable/statics

imports lang/base/statics

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

rules

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

rules

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

rules

  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)) :- {T
    typeOfType(s_ctx, t) == T1,
    declareVar(s_bnd, x, T1),
    typeOfExp(s_ctx, e) == T2,
    subtype(e, T2, T1)
    | error $[type of expression [T2] does r


rules // variable binding

  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
                // declaration is distinct

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

```
module lang/module-seq/statics

imports lang/base/statics
imports lang/variable/statics
imports lang/arithmetic/statics

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

rules

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

  decGroupsOk : scope * DecGroups → scope

  decGroupsOk(s, Decs(decls)) = s:-
    declsOk(s, decls).

  decGroupsOk(s1, Seq(decls1, decls2)) = s_end :- {s2}
    new s2, s2 -P→ s1,
    declsOk(s1, decls1),
    decGroupsOk(s2, decls2) == s_end.

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

// permission to extend
//  declOk(s, Import(p)) :- {s_mod s_end}
//    typeOfModRef(s, p) == MOD(s_mod),
//    s -I→ s_mod,
//    declareVar(s_mod, "x", INT()).



rules // module reference
```

```
module ModuleNestedSeq {

module A {
  module A {
    def b = 1
  }
}
module C {
  import A;
  import A
  def c = b
}

}
```

```
10:38 | INFO  | Build log                         - Execute strj -i /Users/eelcovisser/03-Research/MetaBorgCube/statix-sandbox/paris/trans/paris.str -o /Users/eelcovisser
10:38 | INFO  | Build log                         - > Generate typesmart analysis
10:38 | INFO  | .m.s.m.c.b.LanguageSpecBuilder - Compiling Main ESV file eclipse:///paris/editor/Main.esv
```

Concrete and Abstract Syntax

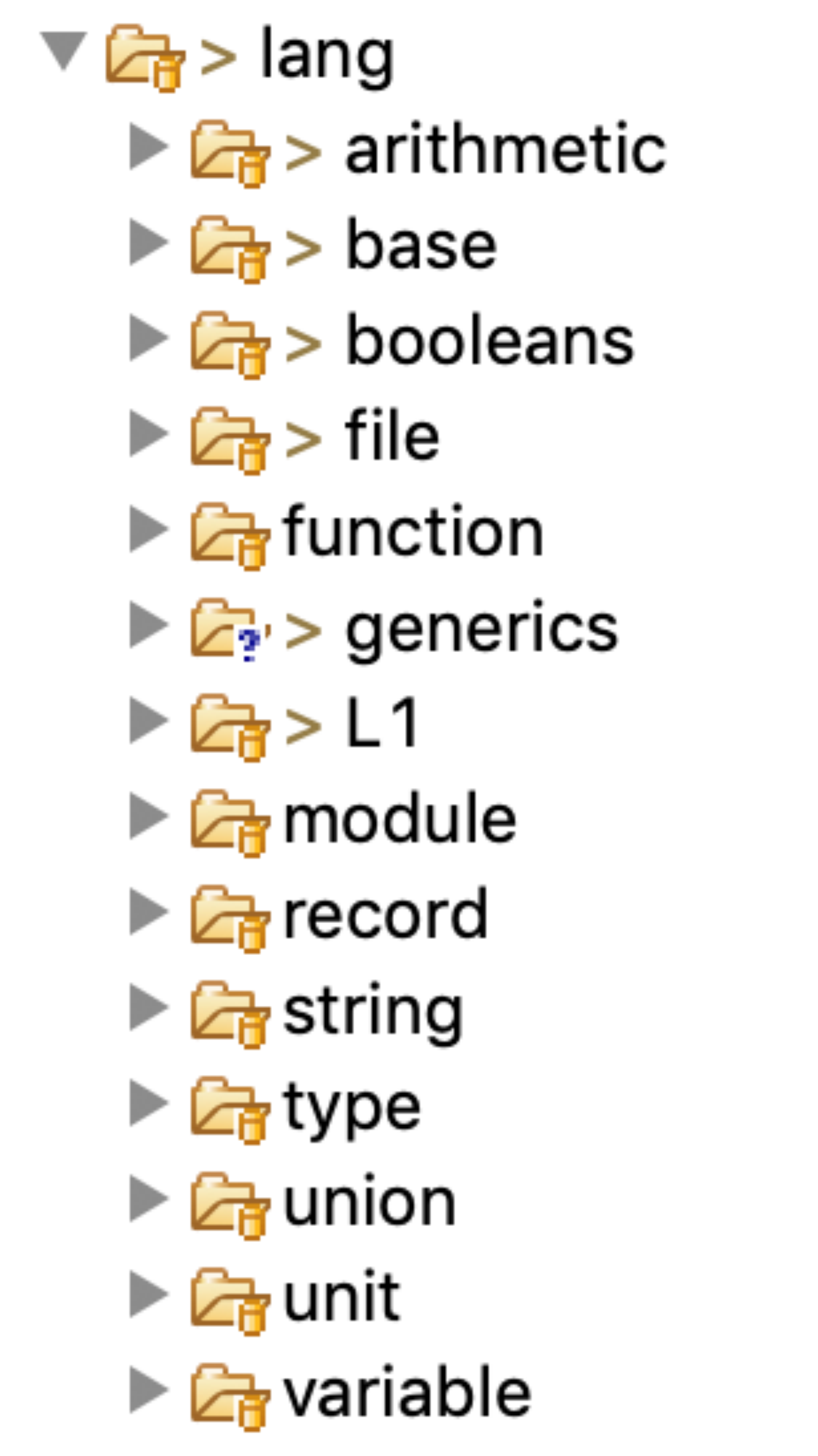


```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax

  Exp.Int    = <<INT>>
  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}

template options

  ID = keyword {reject}
  keyword -/- [a-zA-Z0-9]
```

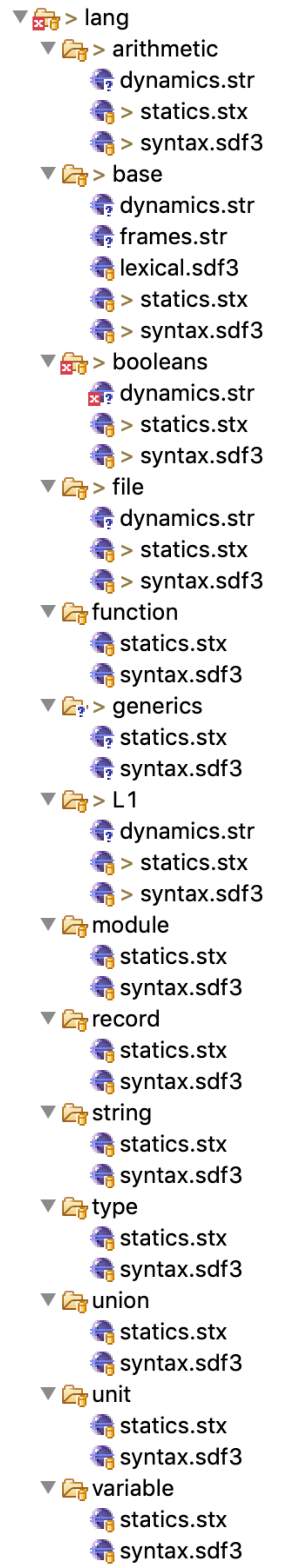
```
signature
  constructors
    Int  : INT → Exp
    Add  : Exp * Exp → Exp
    Sub  : Exp * Exp → Exp
    Mul  : Exp * Exp → Exp
    IntT : Type
    INT  : TYPE
```

```
1 + 2 * 3
```

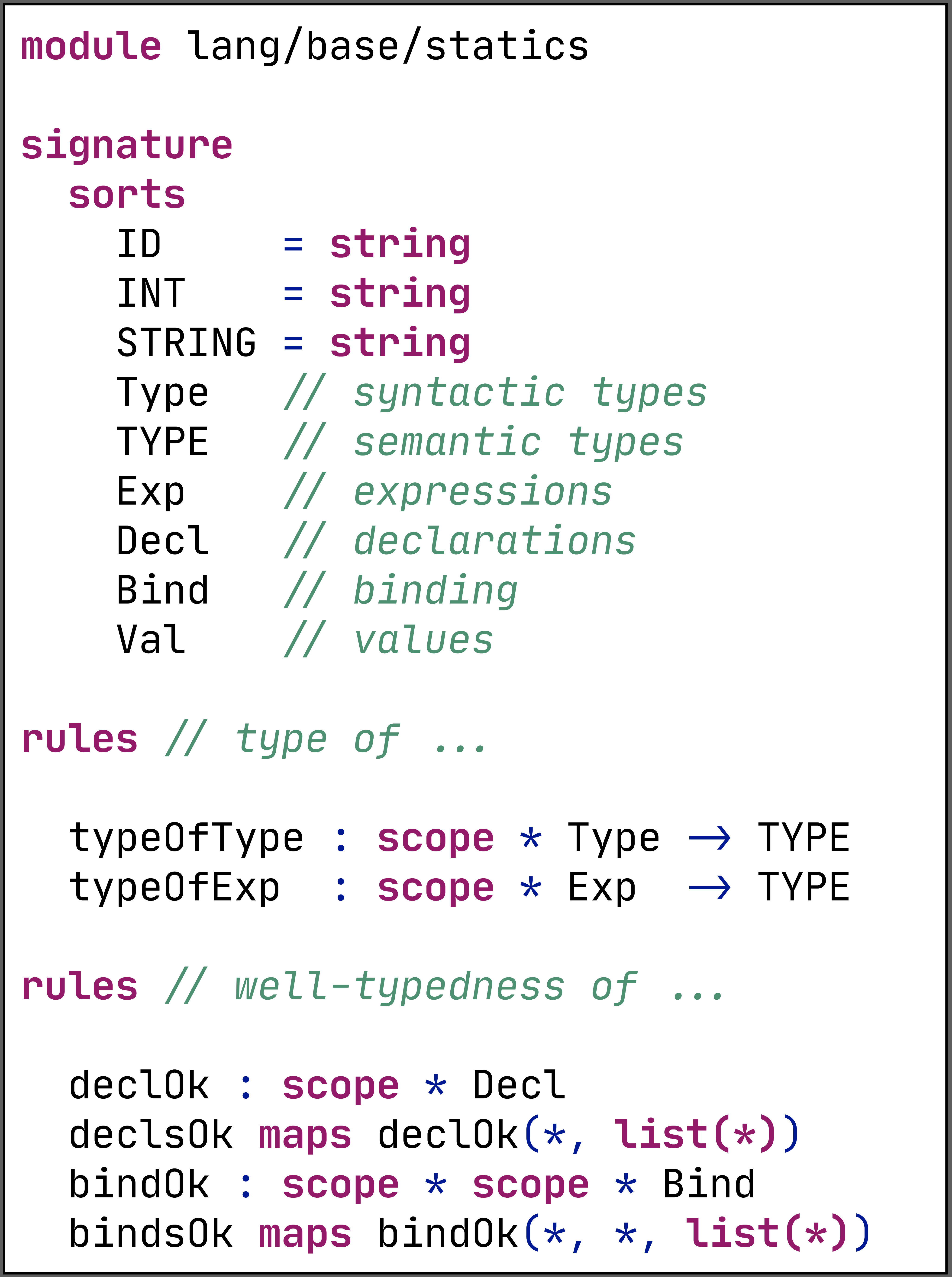
```
Add(
  Int("1"),
  Mul(
    Int(“2"),
    Int(“3")))
```



```
▼ > lang
    ▶ > arithmetic
    ▶ > base
    ▶ > booleans
    ▶ > file
    ▶ function
    ▶ > generics
    ▶ > L1
    ▶ module
    ▶ record
    ▶ string
    ▶ type
    ▶ union
    ▶ unit
    ▶ variable
```



```
▼ > lang
    ▼ > arithmetic
        dynamics.str
        > statics.stx
        > syntax.sdf3
    ▼ > base
        dynamics.str
        frames.str
        lexical.sdf3
        > statics.stx
        > syntax.sdf3
    ▼ > booleans
        dynamics.str
        > statics.stx
        > syntax.sdf3
    ▼ > file
        dynamics.str
        > statics.stx
        > syntax.sdf3
    ▼ function
        statics.stx
        syntax.sdf3
    ▼ > generics
        statics.stx
        syntax.sdf3
    ▼ > L1
        dynamics.str
        > statics.stx
        > syntax.sdf3
    ▼ module
        statics.stx
        syntax.sdf3
    ▼ record
        statics.stx
        syntax.sdf3
    ▼ string
        statics.stx
        syntax.sdf3
    ▼ type
        statics.stx
        syntax.sdf3
    ▼ union
        statics.stx
        syntax.sdf3
    ▼ unit
        statics.stx
        syntax.sdf3
    ▼ variable
        statics.stx
        syntax.sdf3
```



```
module lang/base/statics

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

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(*))
```

Type Predicates

```
signature
  constructors
    IntT : Type
    INT  : TYPE
    Int  : INT -> Exp
    Add  : Exp * Exp -> Exp
    Sub  : Exp * Exp -> Exp
    Mul  : Exp * Exp -> Exp
```

```
rules

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

rules

  typeOfExp(s, Int(i)) = 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().
```

```
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).
```

Declaring and Resolving Names

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

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

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

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

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

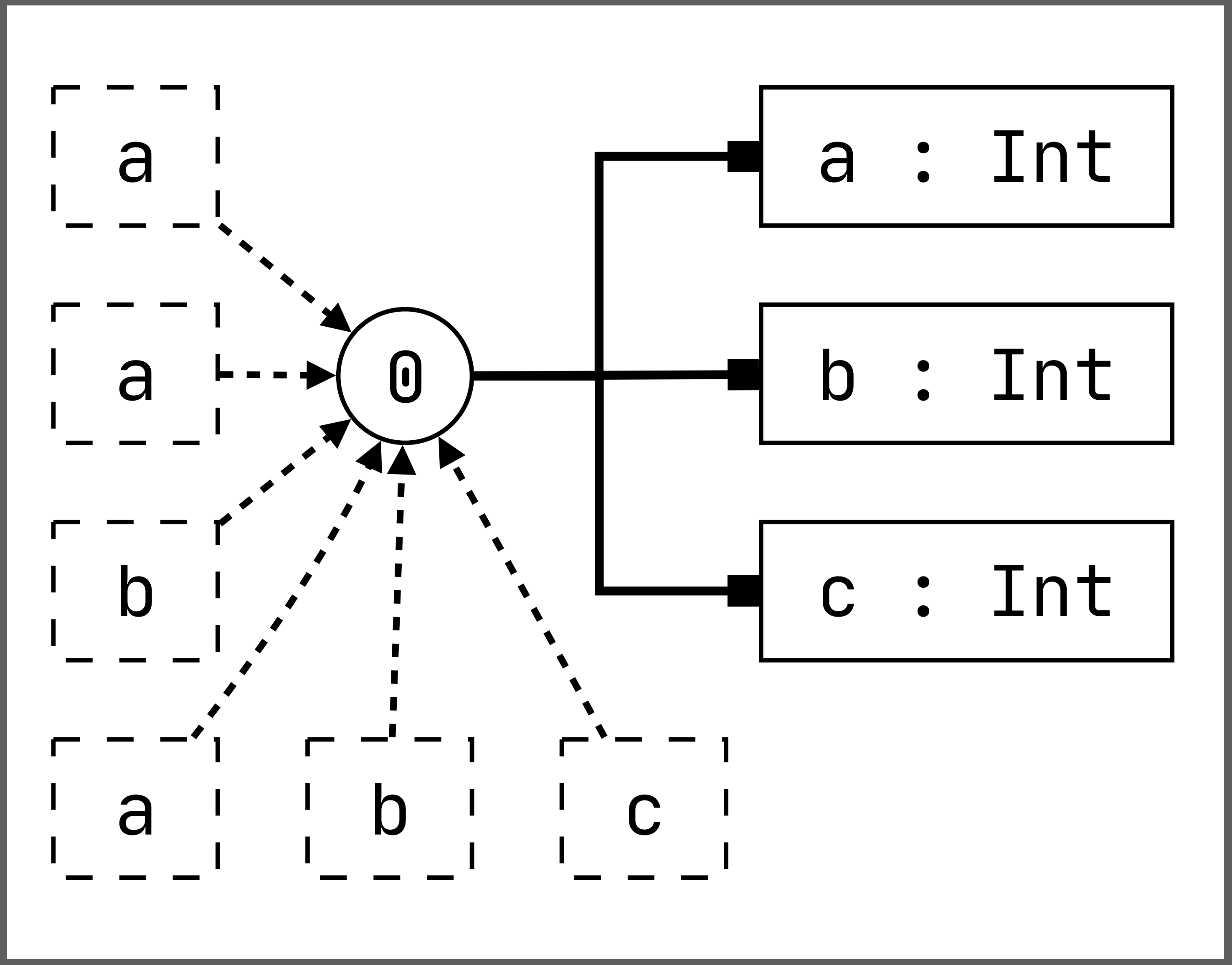
```
def a = 0
def b = a + 1
def c = a + d
> a + e + c
```

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

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

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

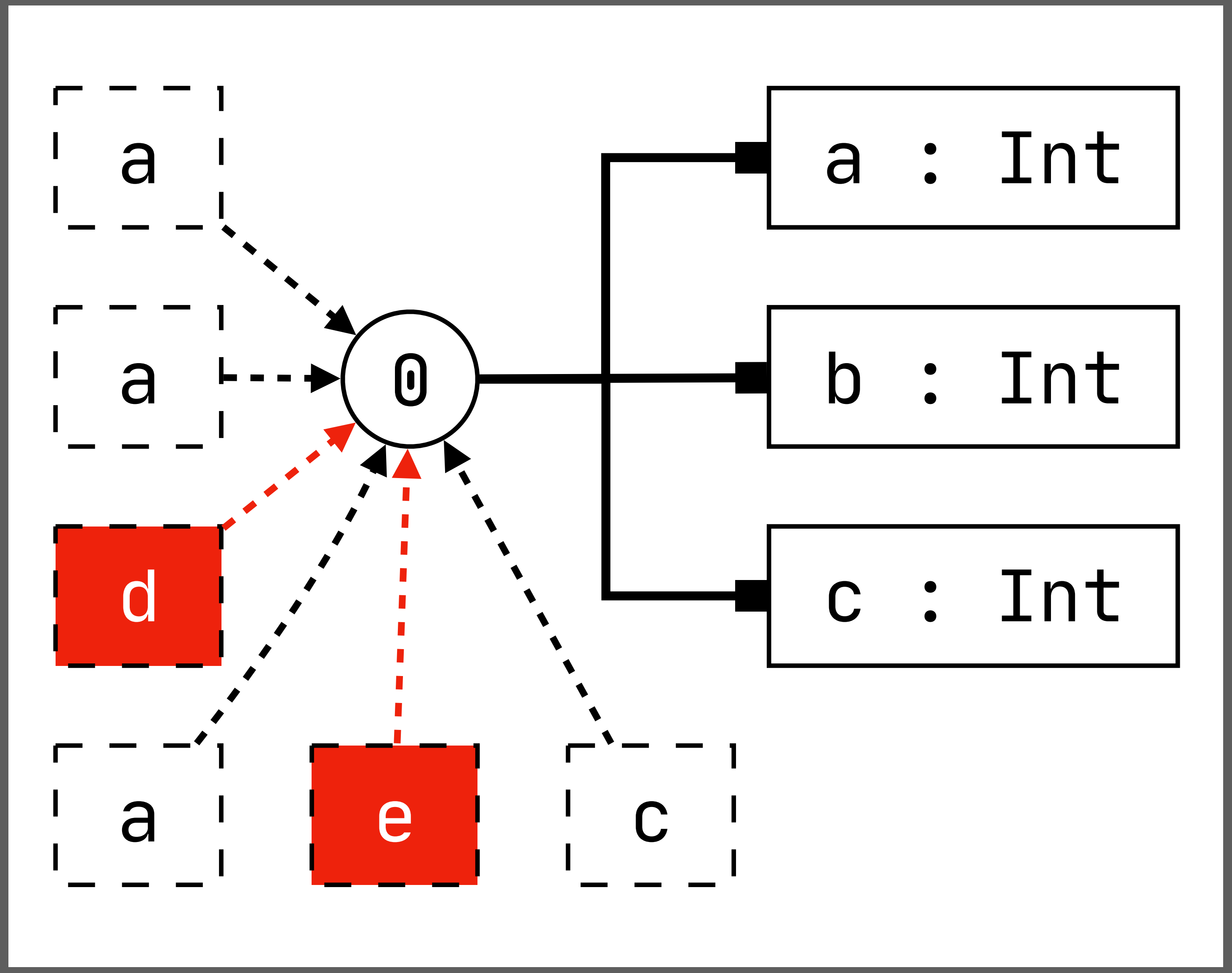
```
rules

  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))].
```



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

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

```
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))].
```

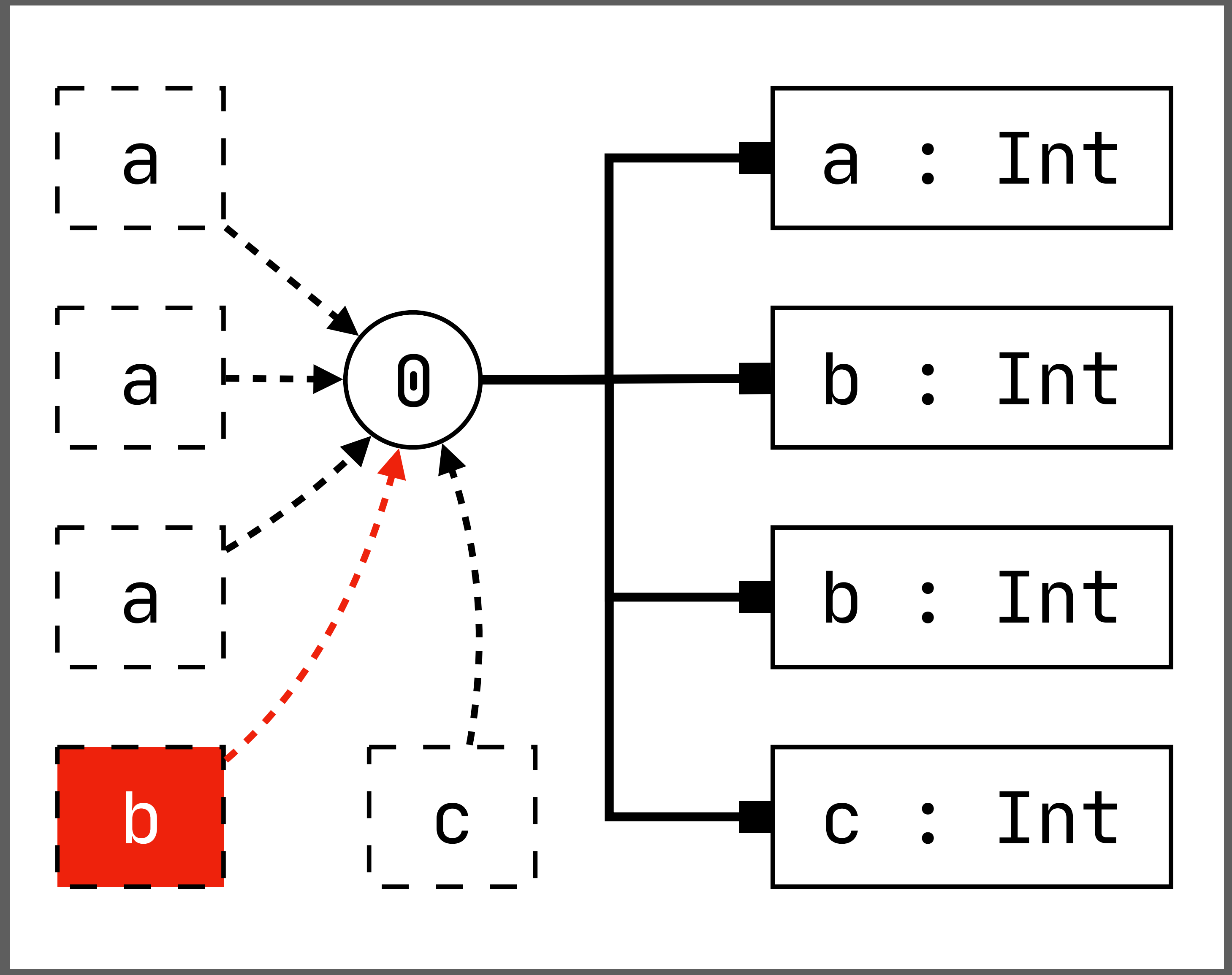


```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
```

```
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
```

```
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))].
```

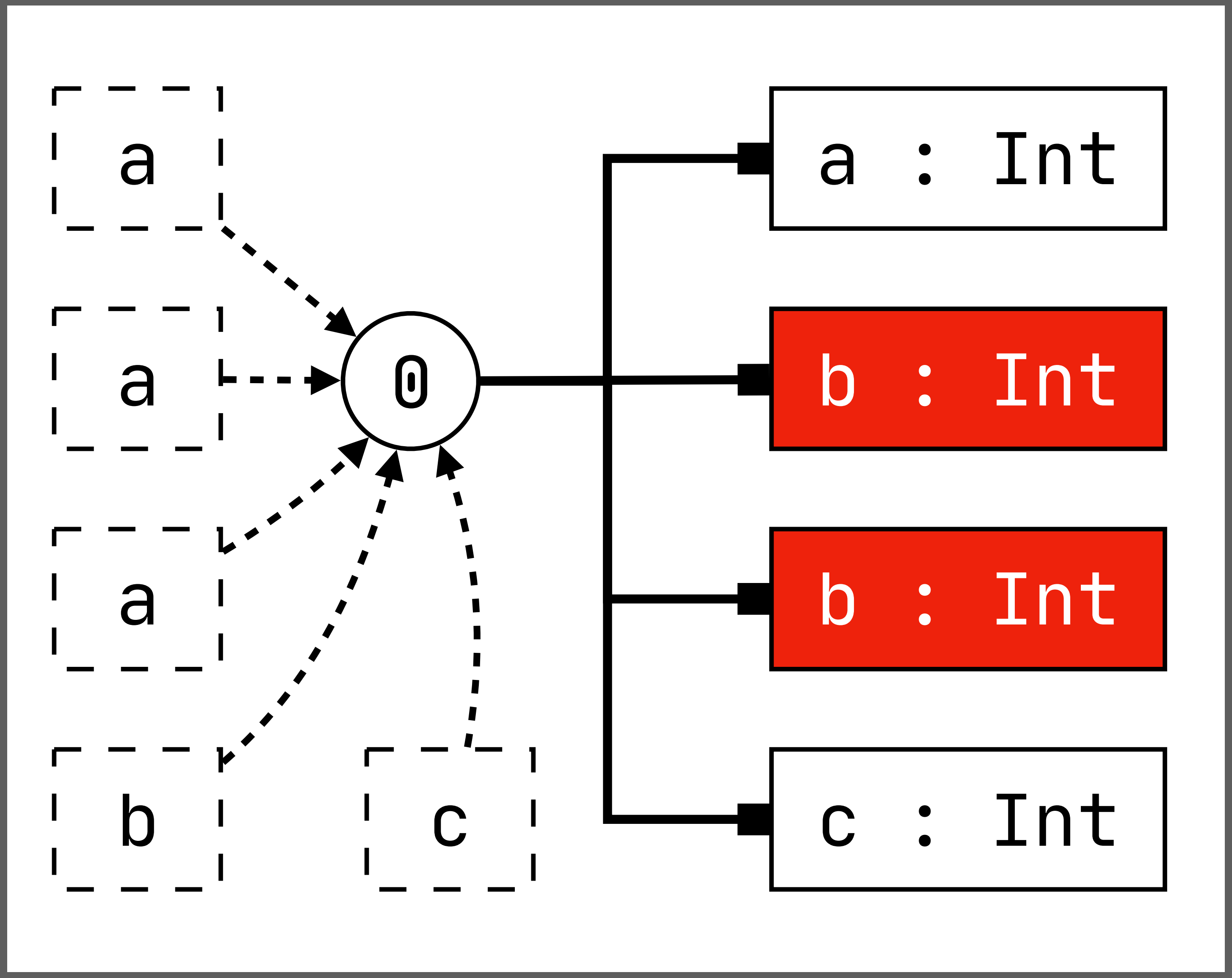


```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
```

```
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
```

```
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'.
```

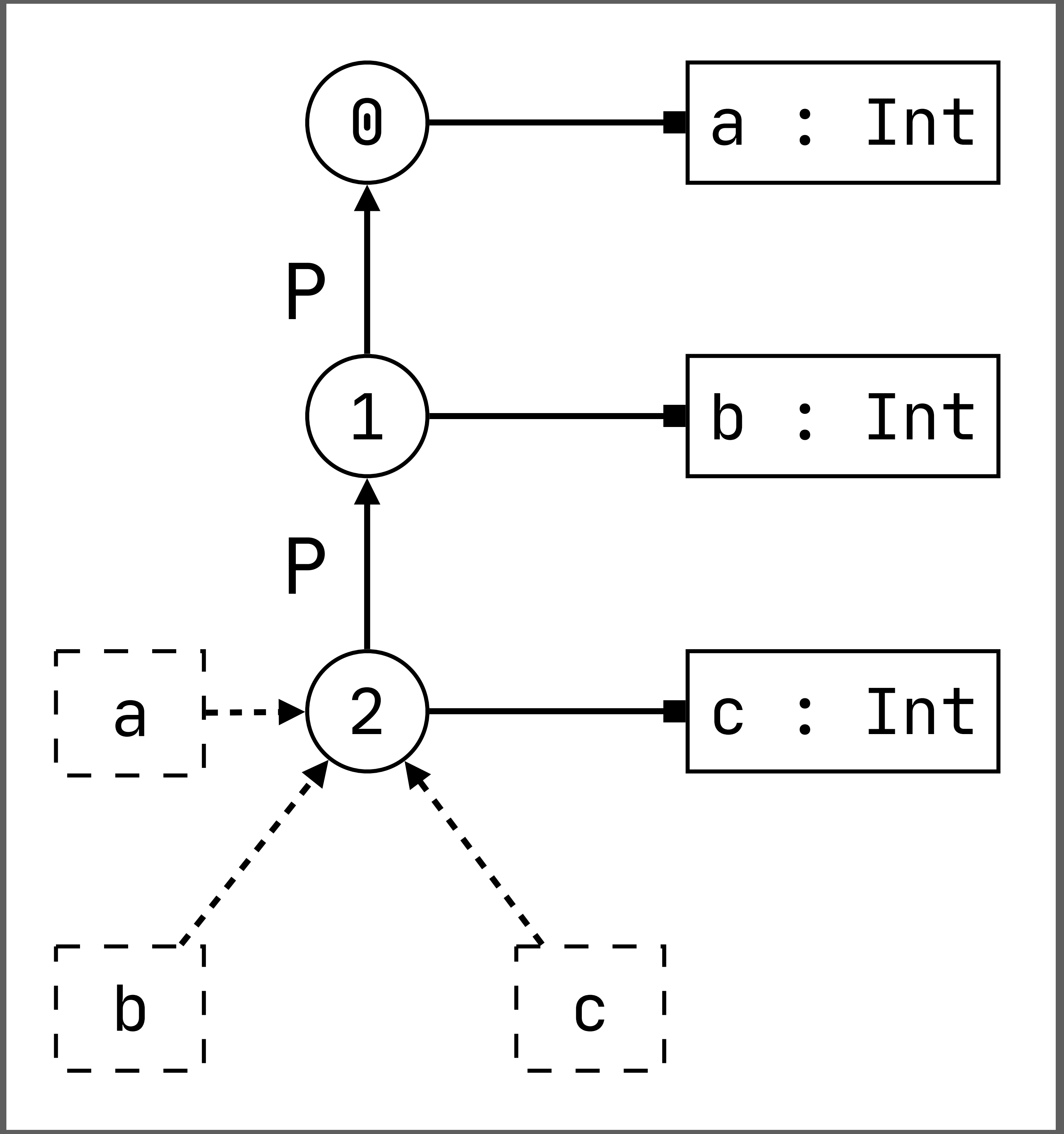


Lexical Scope

```
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.
```

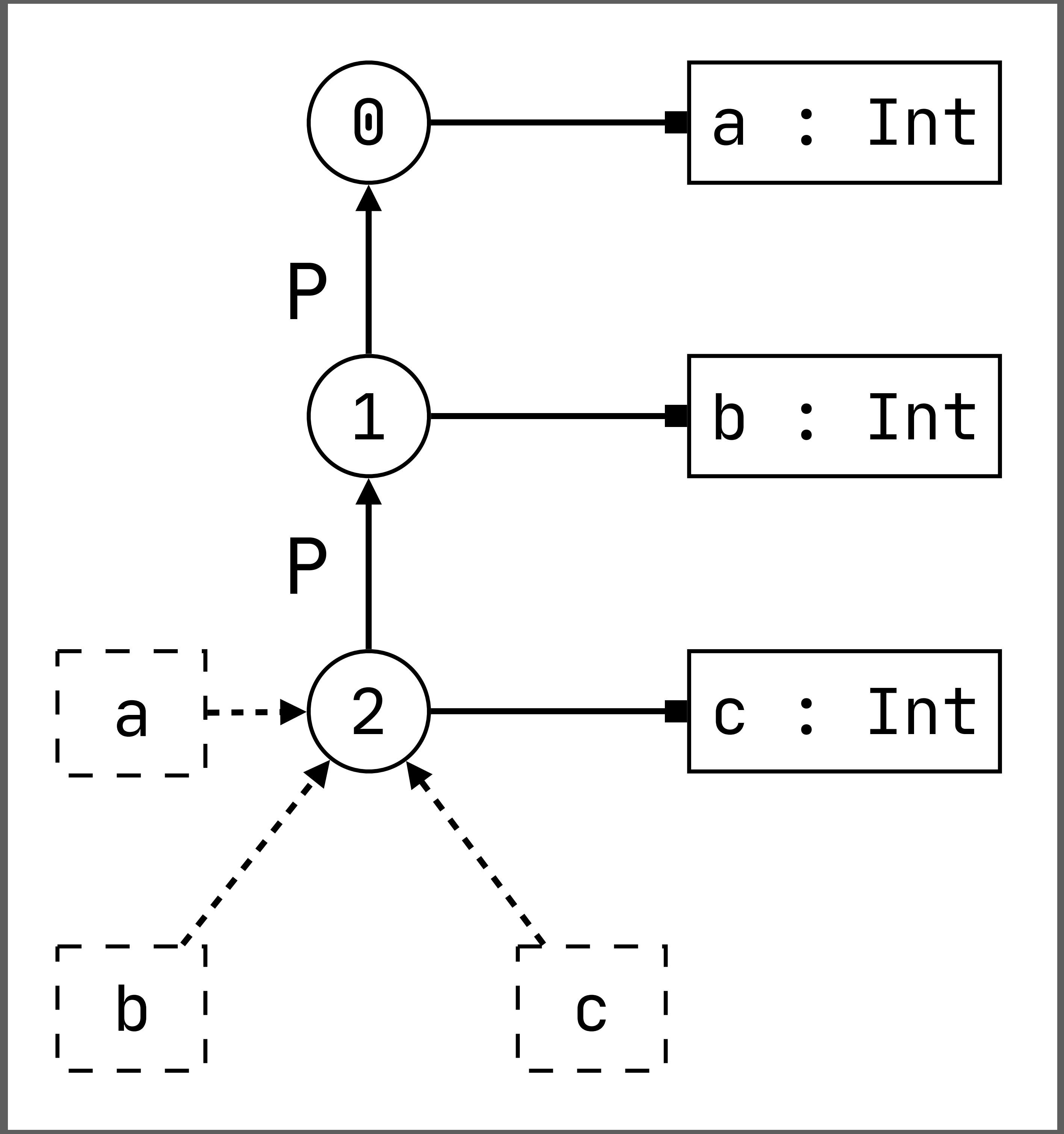


```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
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
```

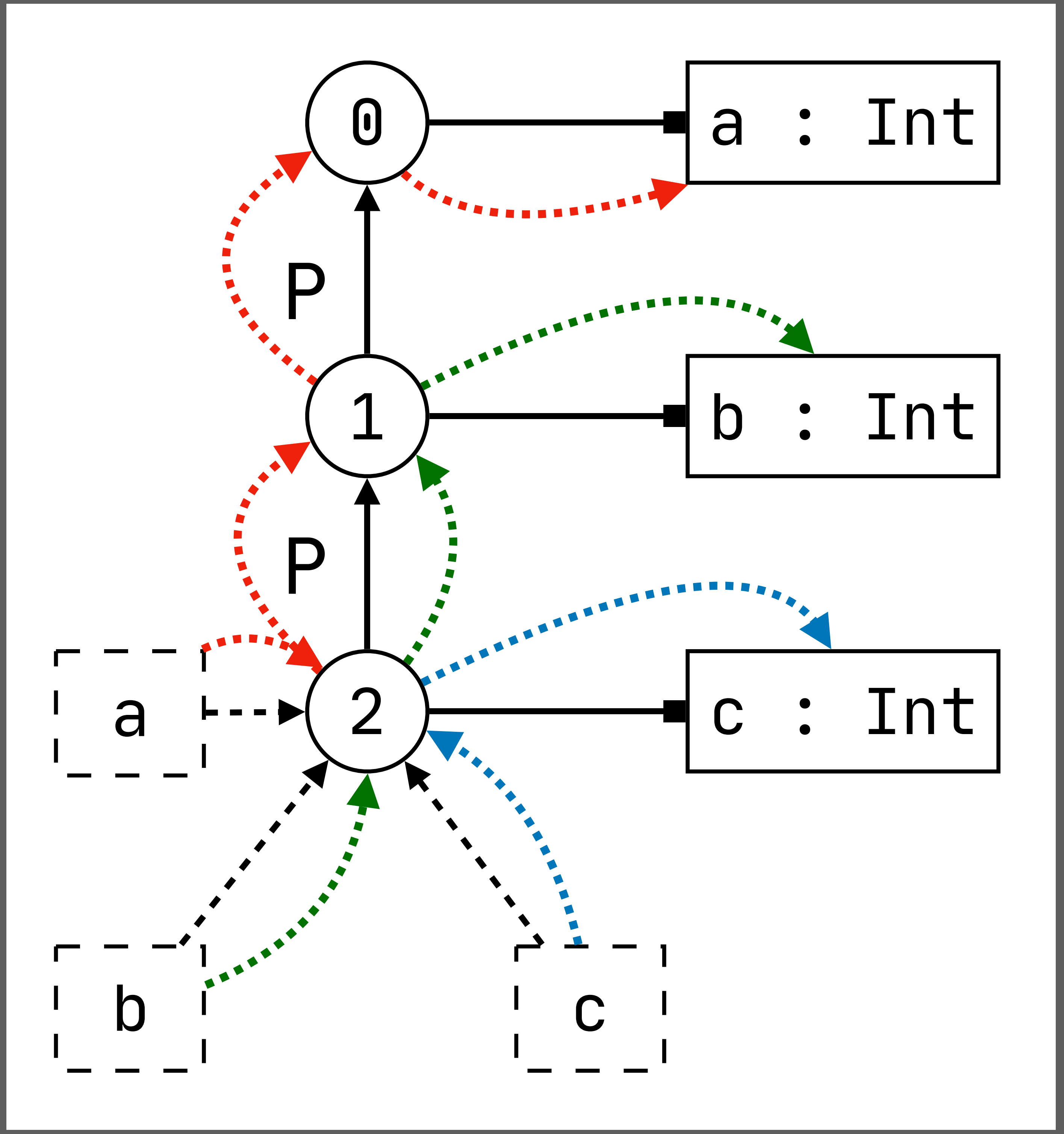
```
rules

  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.
```



```
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.
```

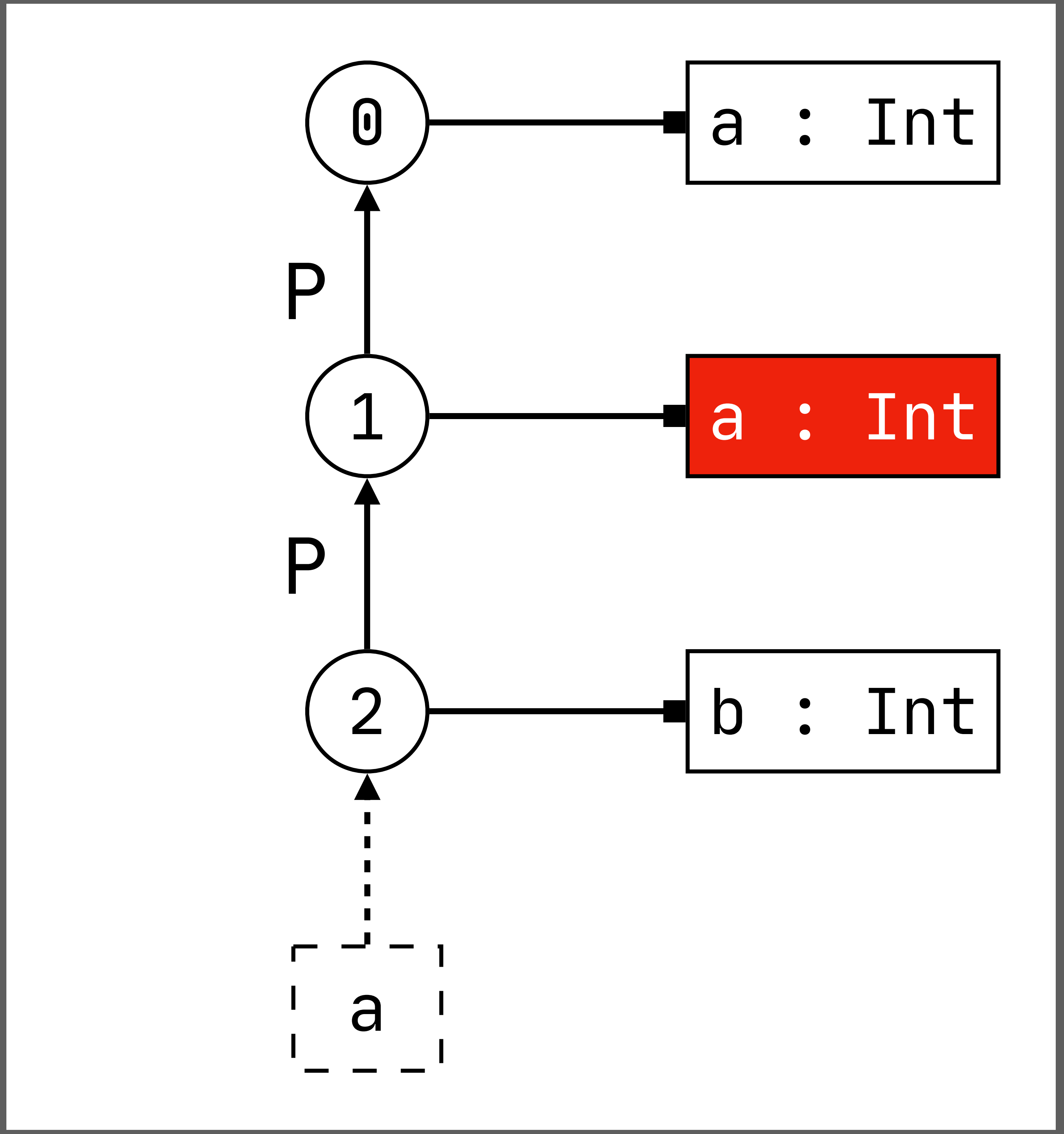


```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

```
signature
  constructors
    Let : ID * Exp * Exp -> Exp
```

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

```
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.
```

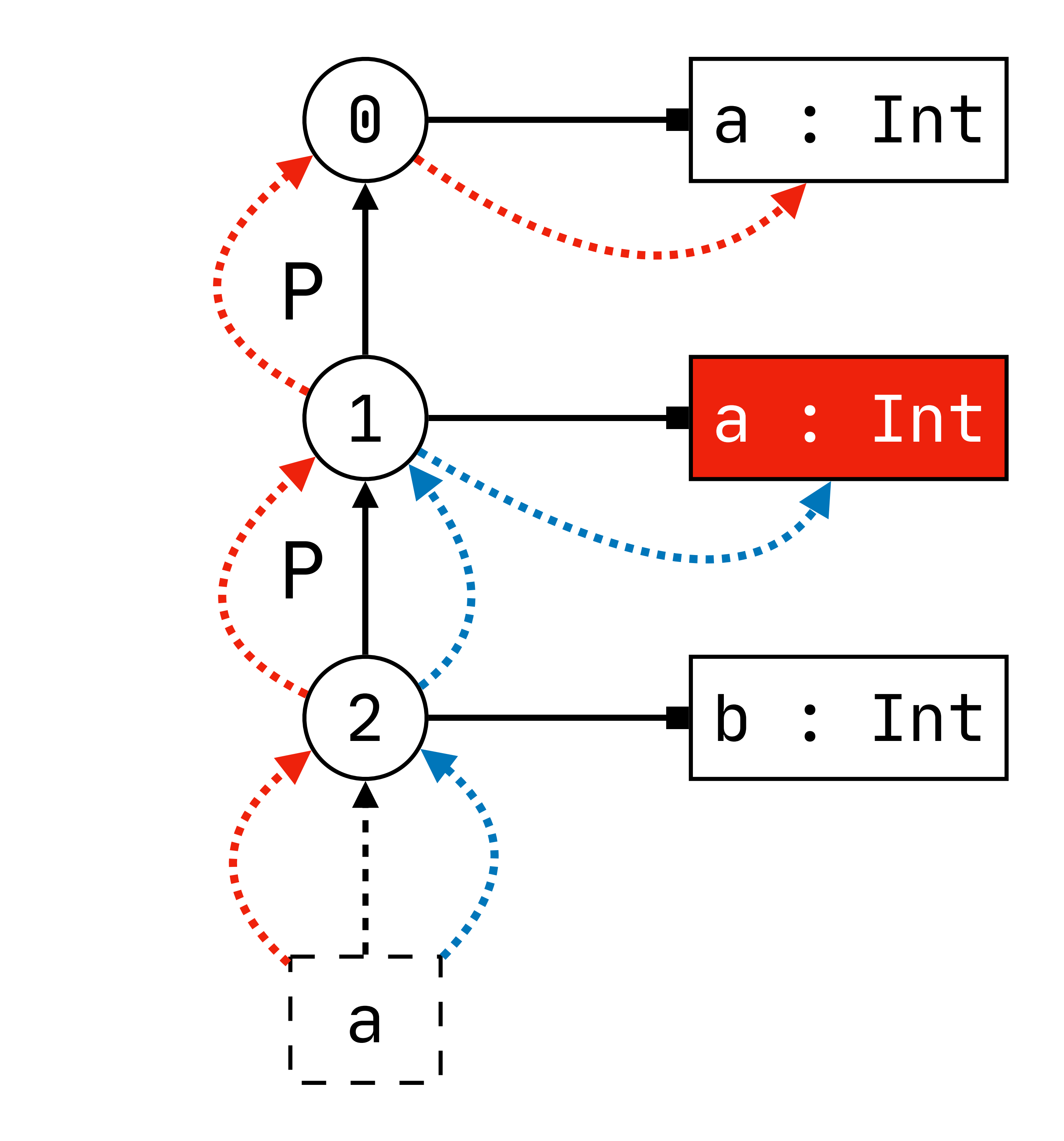


```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

```
signature
  constructors
    Let : ID * Exp * Exp -> Exp
```

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

```
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.
```

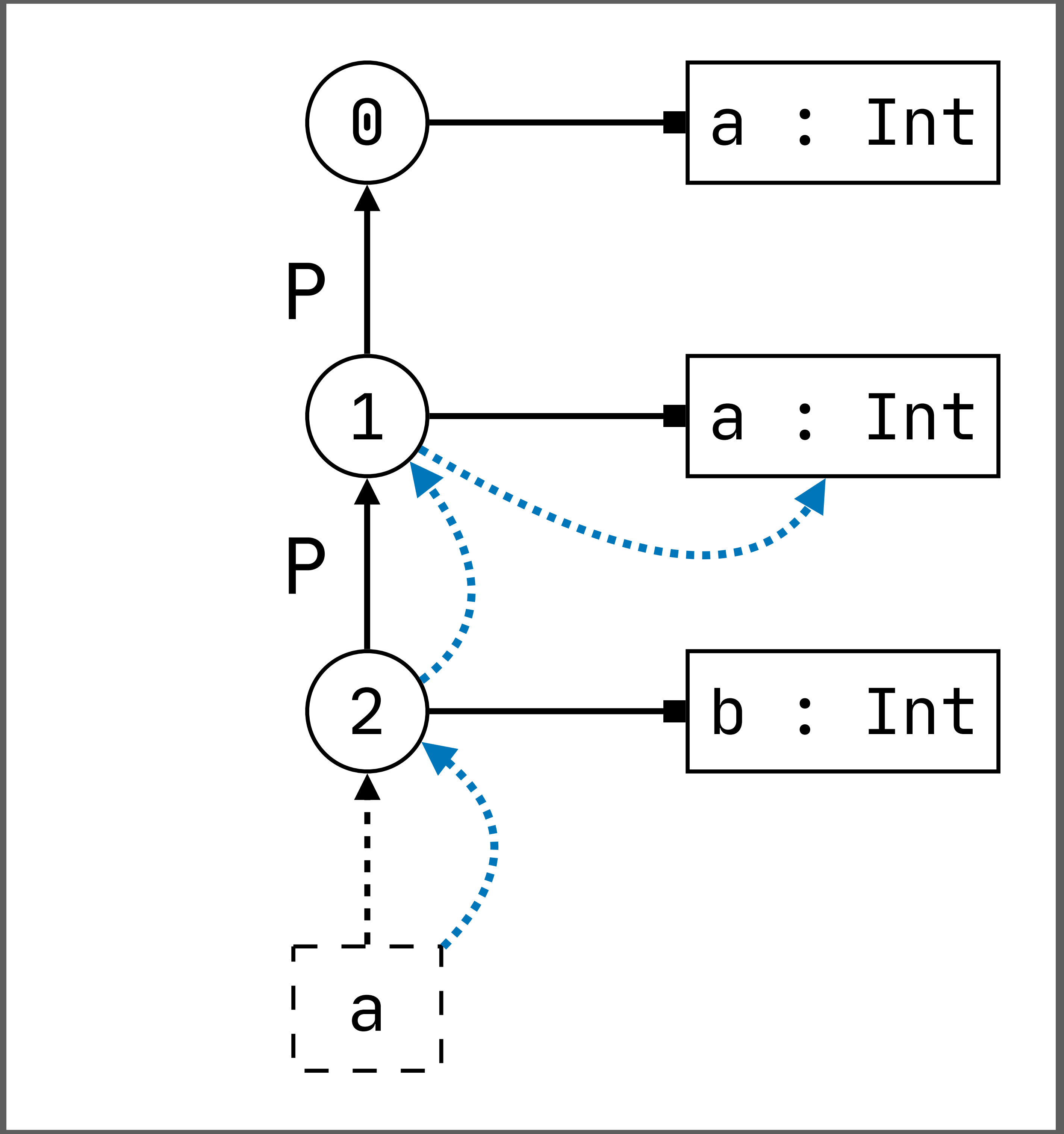


```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

```
signature
  constructors
    Let : ID * Exp * Exp -> Exp
```

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

```
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.
```



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

Scopes as Types

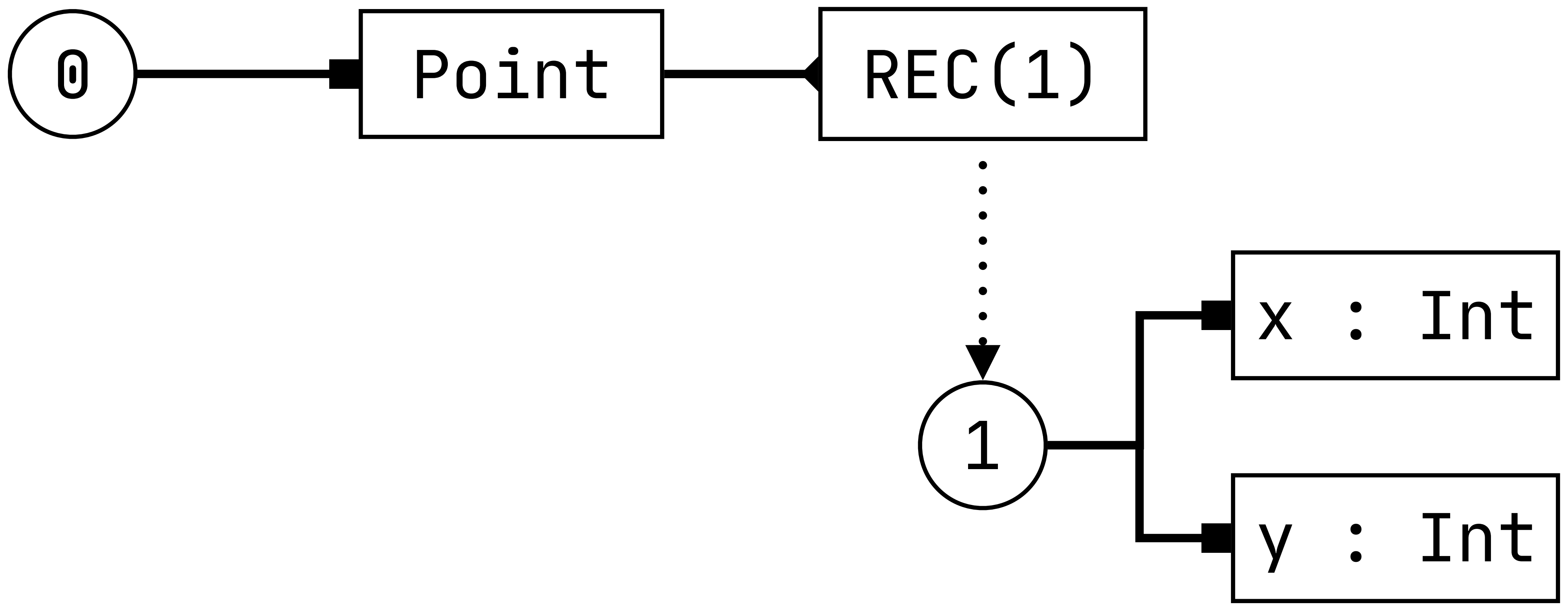
```
signature
  constructors
    REC    : scope ⟶ TYPE
    Record : ID * list(FDecl) ⟶ Decl
    FDecl  : ID * Type ⟶ FDecl
    New    : ID * list(FBind) ⟶ Exp
    FBind  : ID * Exp ⟶ FBind
    Proj   : Exp * ID ⟶ Exp
```

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

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

> p.y
```

```
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 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).
```

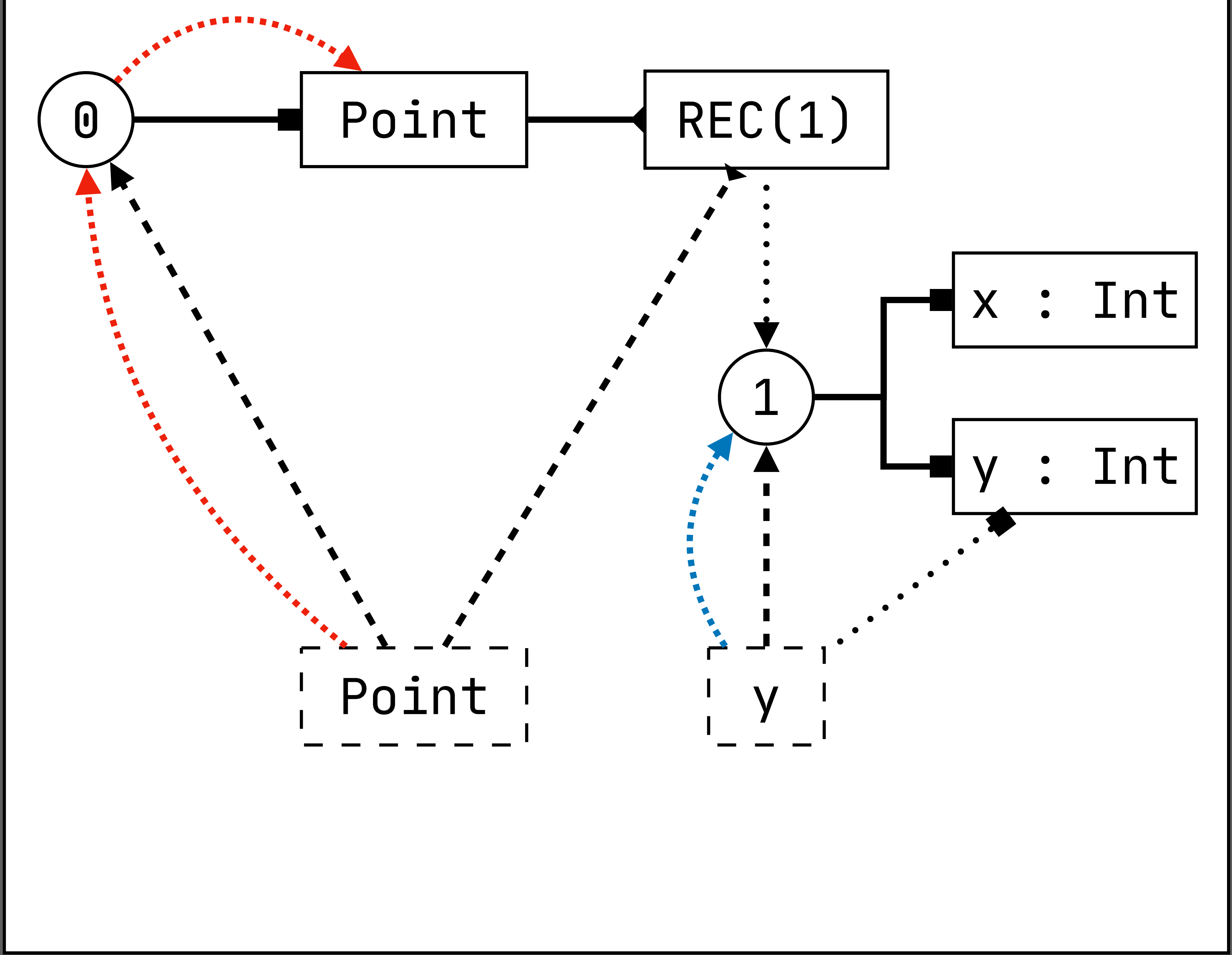
```
record Point { x : Int, y : Int }

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

> p.y
```



```
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).
```

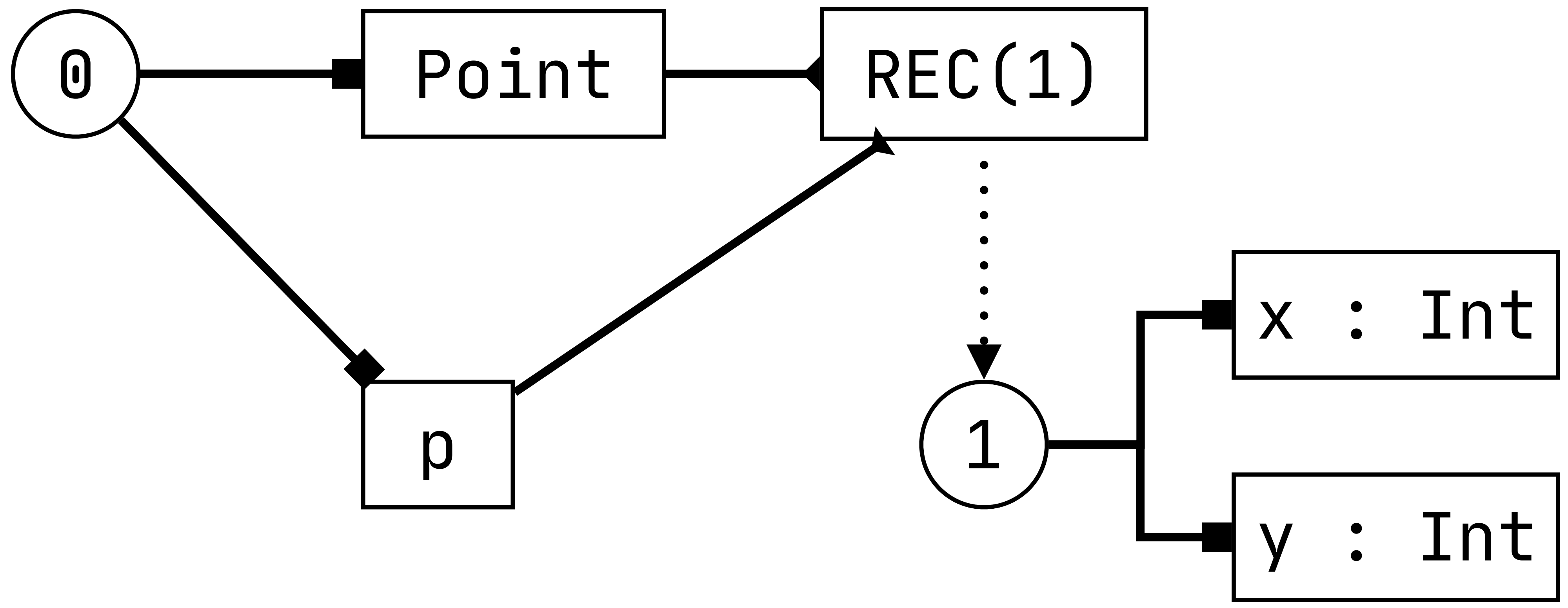
```
record Point { x : Int, y : Int }

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

> p.y
```



```
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).
```

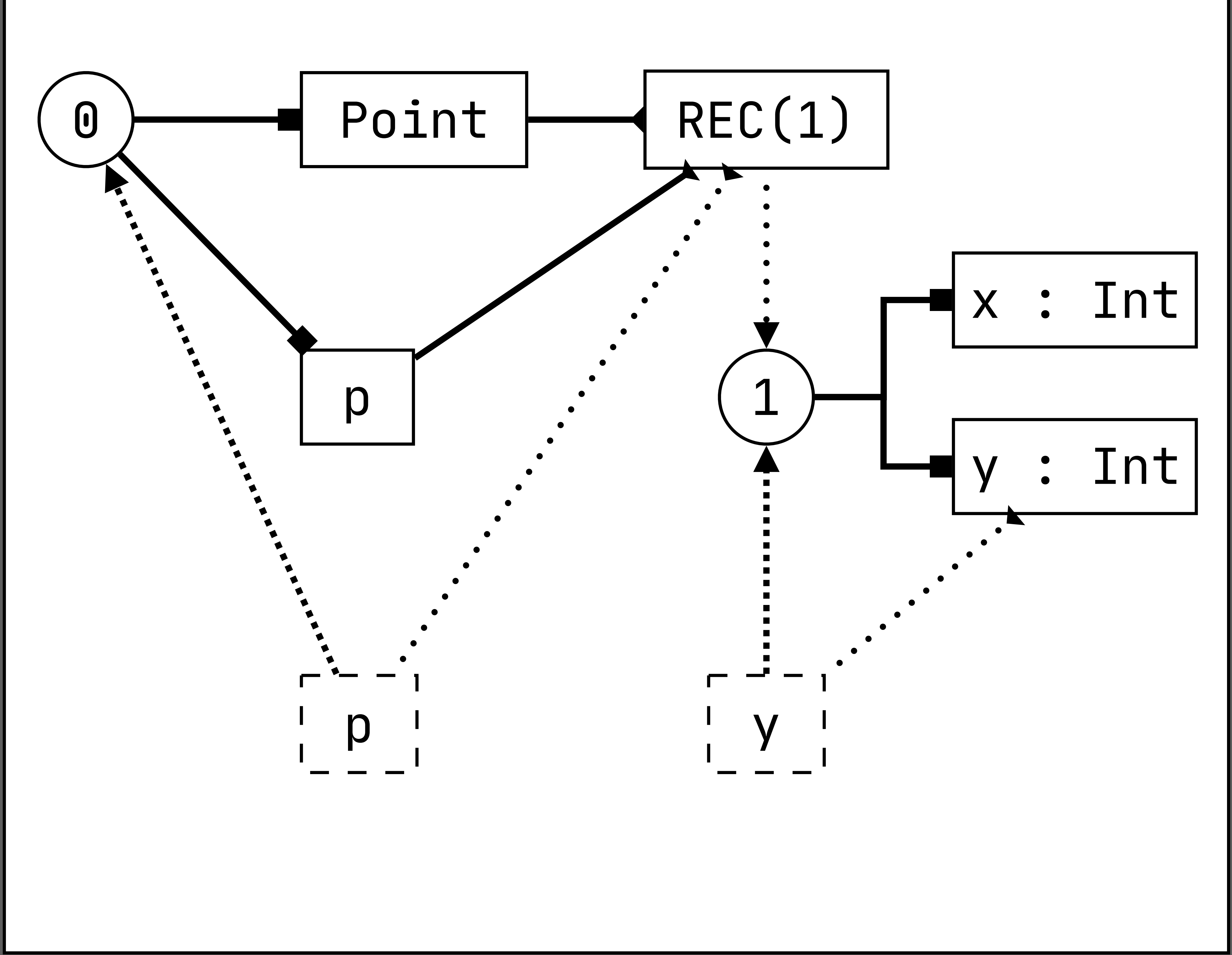
```
record Point { x : Int, y : Int }

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

> p.y
```



```
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.
```

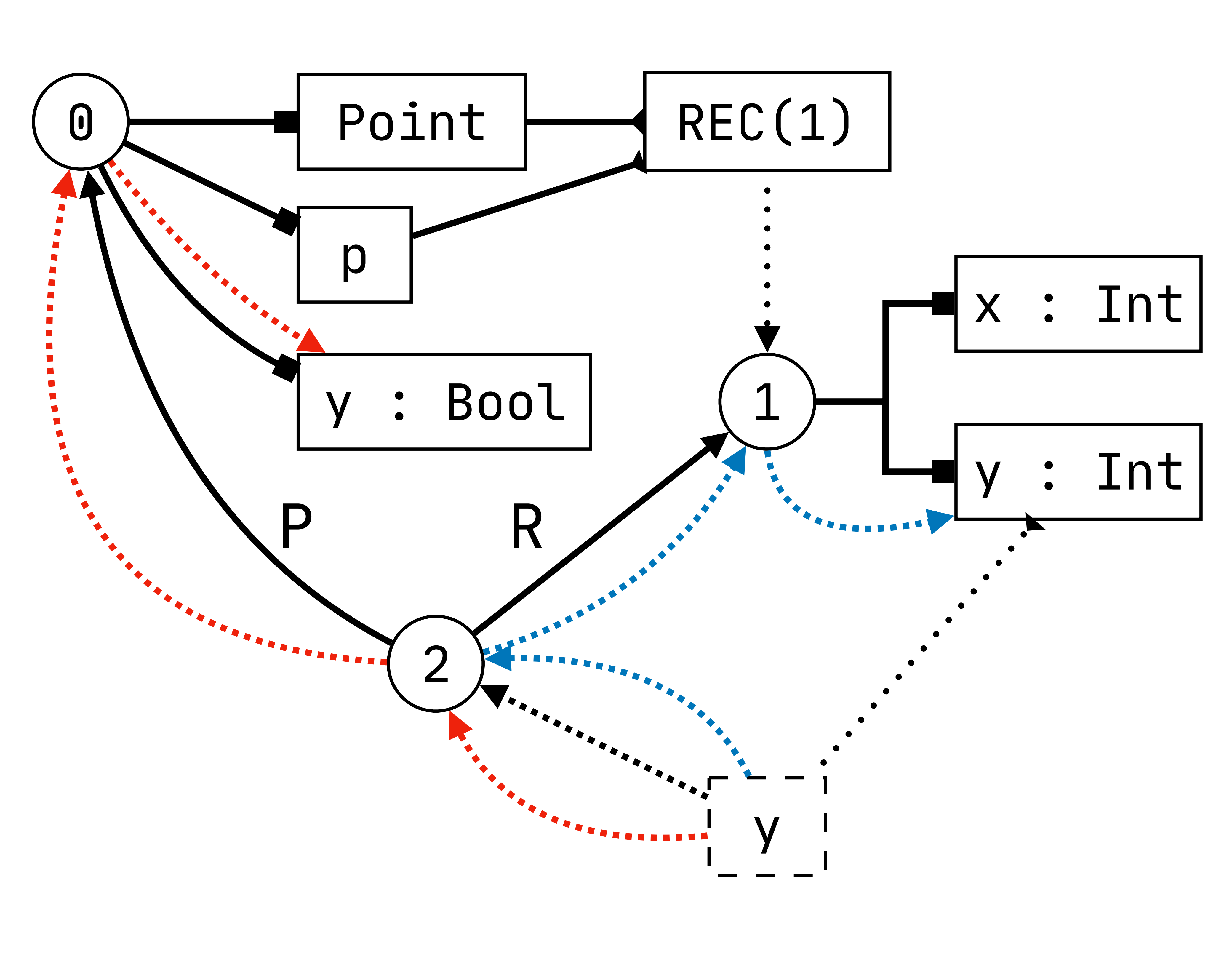
```
record Point { x : Int, y : Int }

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

> p.y
```



```
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}

def y = true

> with p do y
```

```
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
```

Modules

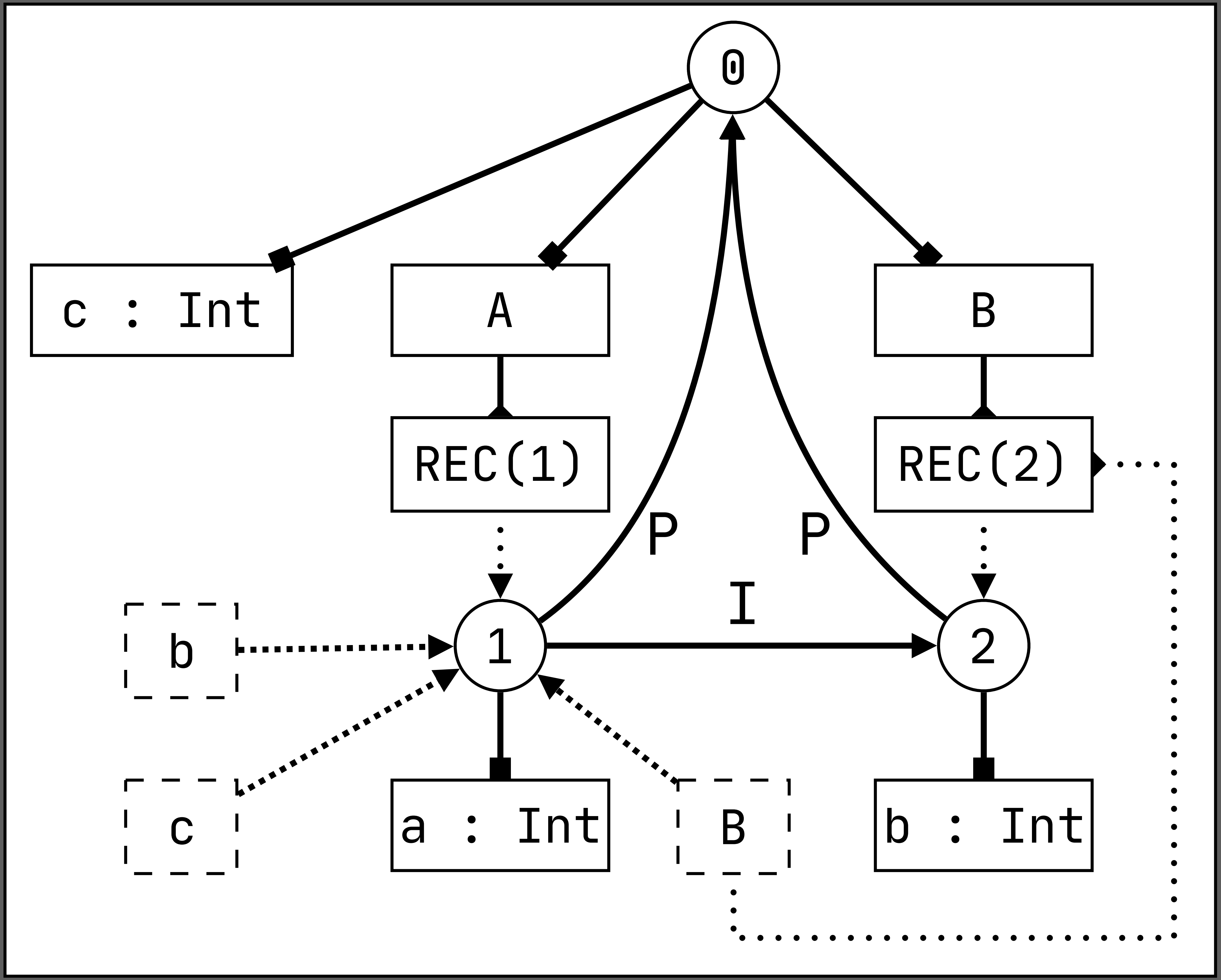
Import

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

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



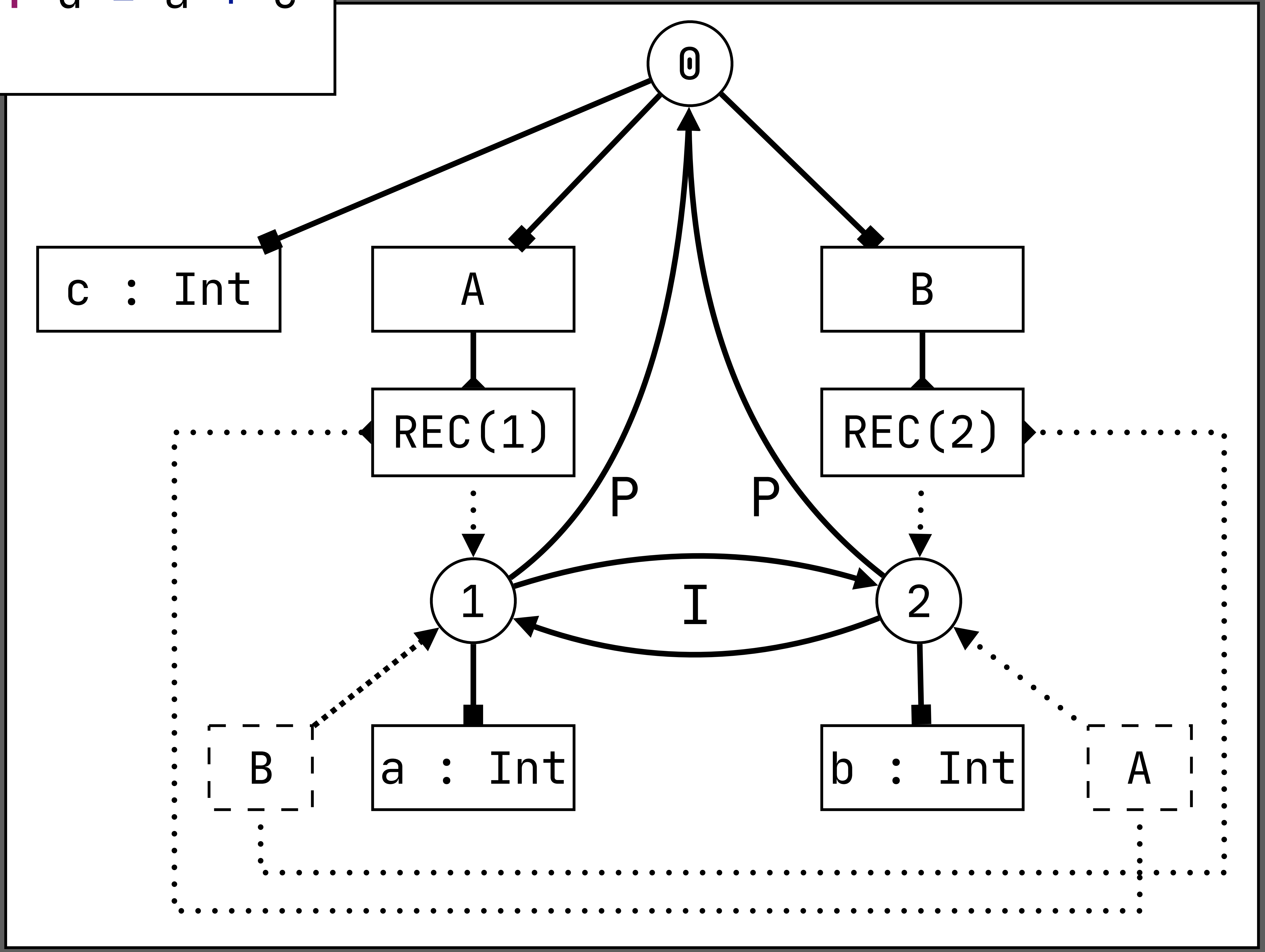
Mutual Imports

```
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.
```

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

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



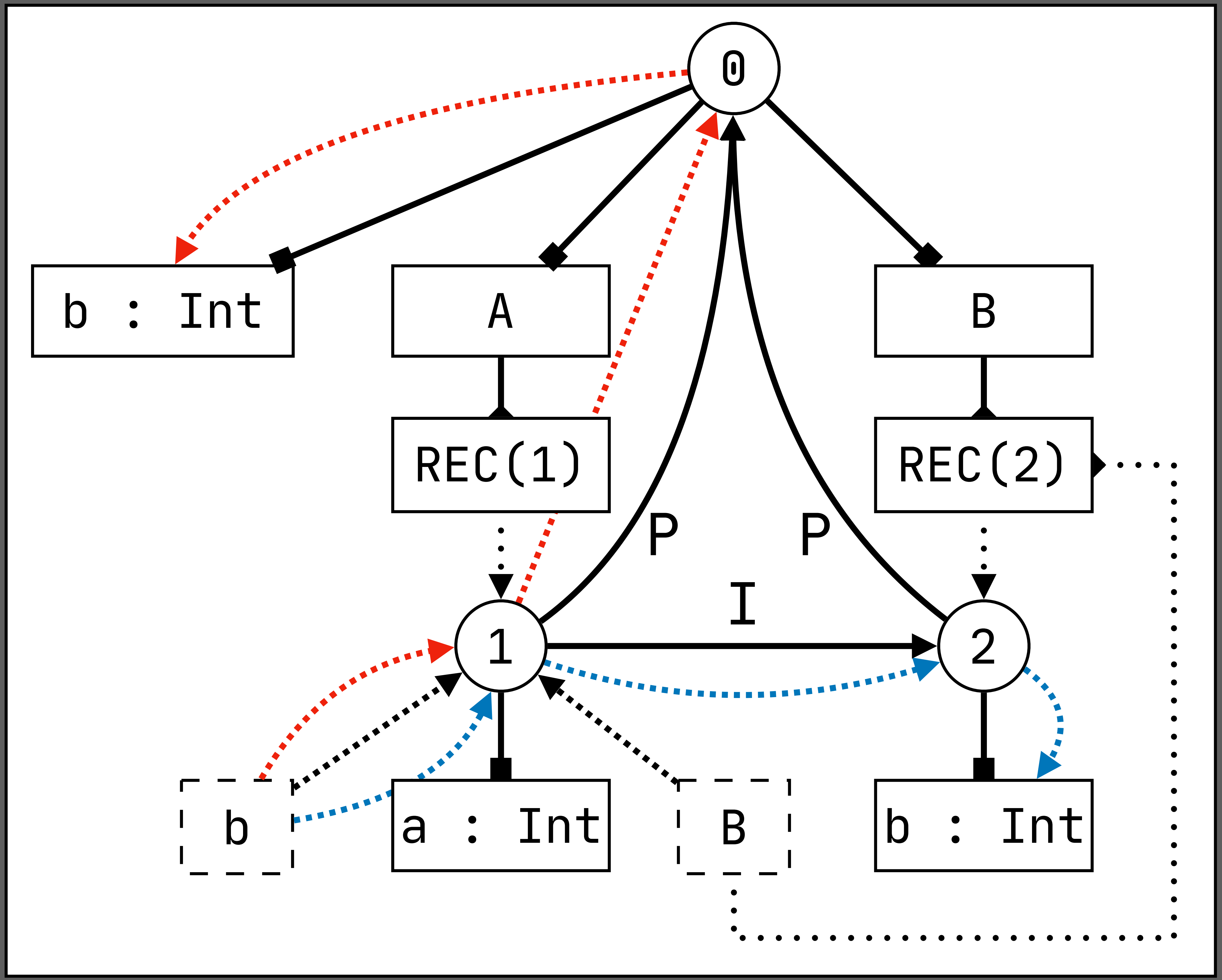
Import vs Parent

```
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* (R* | I*)
      min $ < I, $ < P, I < P, R < P
```

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



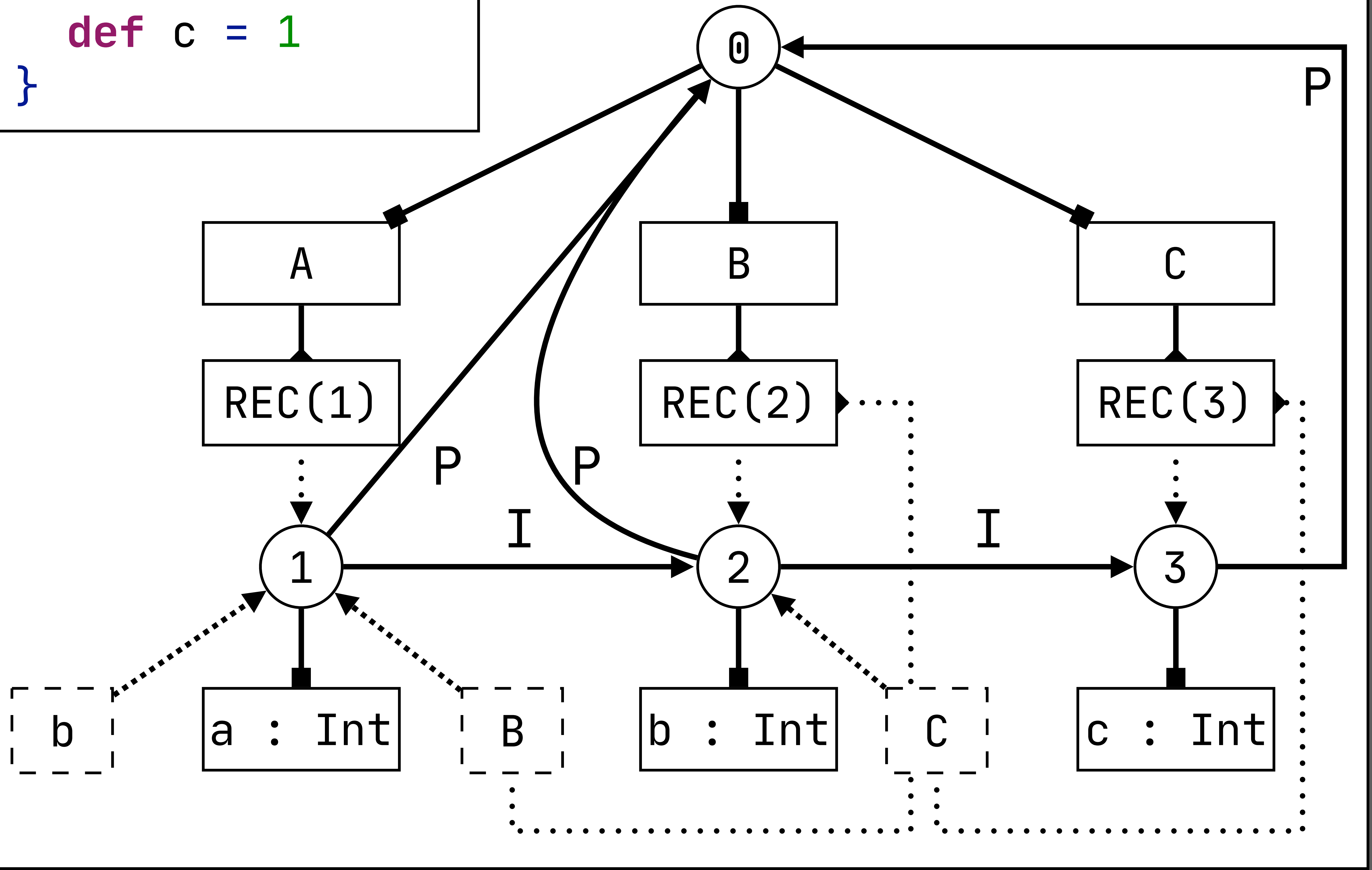
Transitive Import

```
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* (R* | I*)
      min $ < I, $ < P, I < P, R < 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)

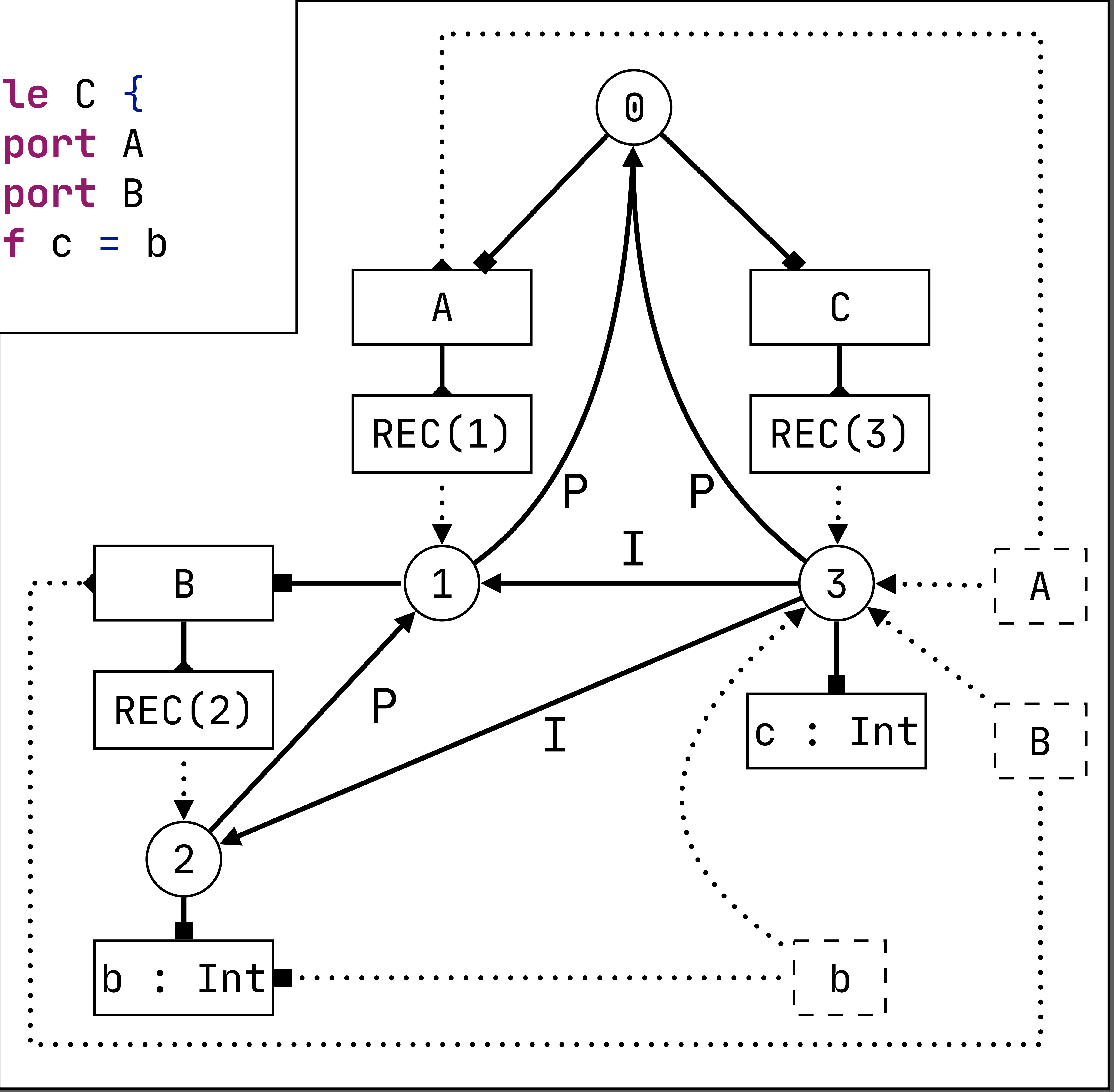
Nested Modules

```
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, R < P
```

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



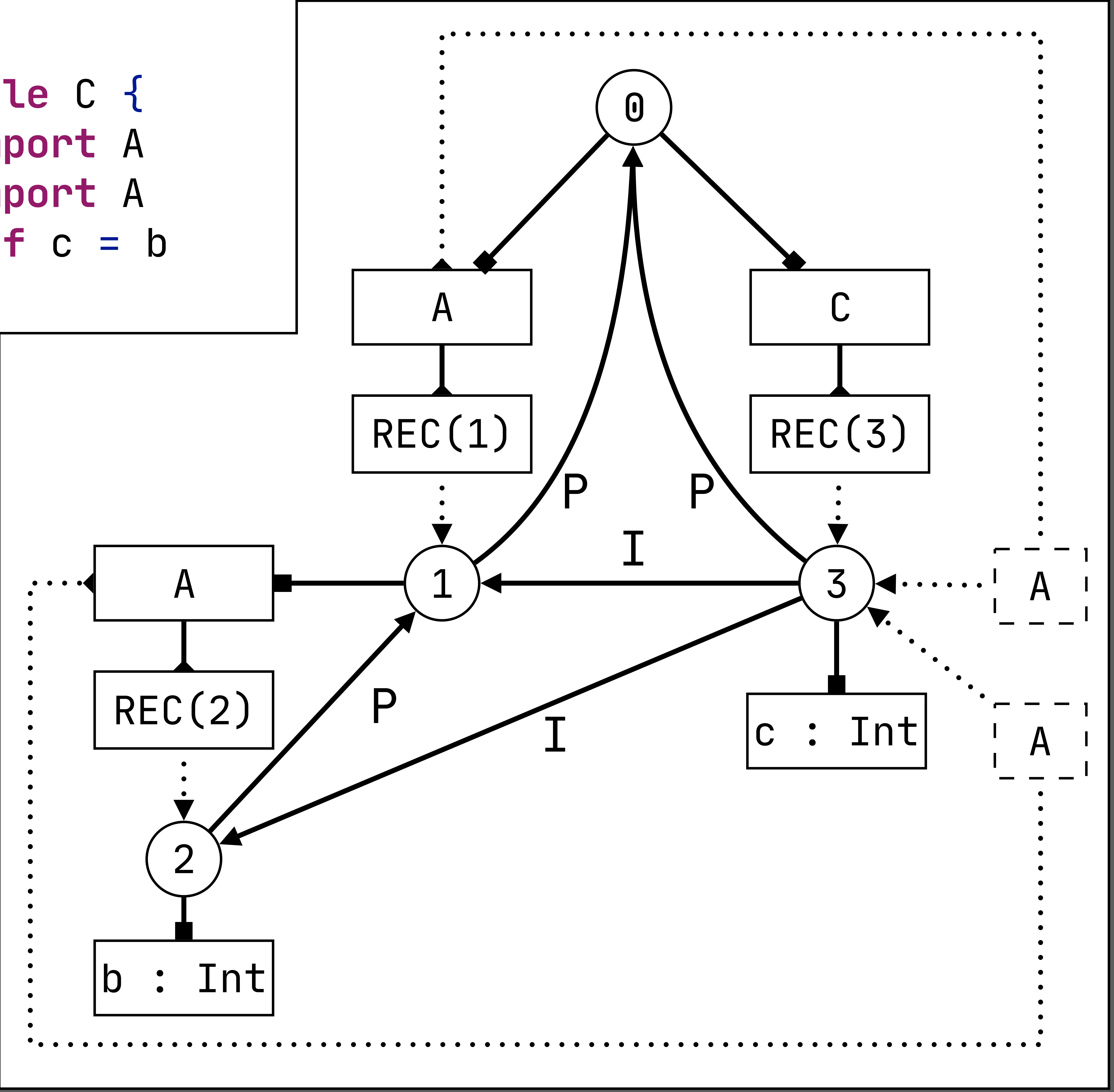
Changing Result of Query

```
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, R < P
```

```
module A {
  module A {
    def b = 1
  }
}
module C {
  import A
  import A
  def c = b
}
```

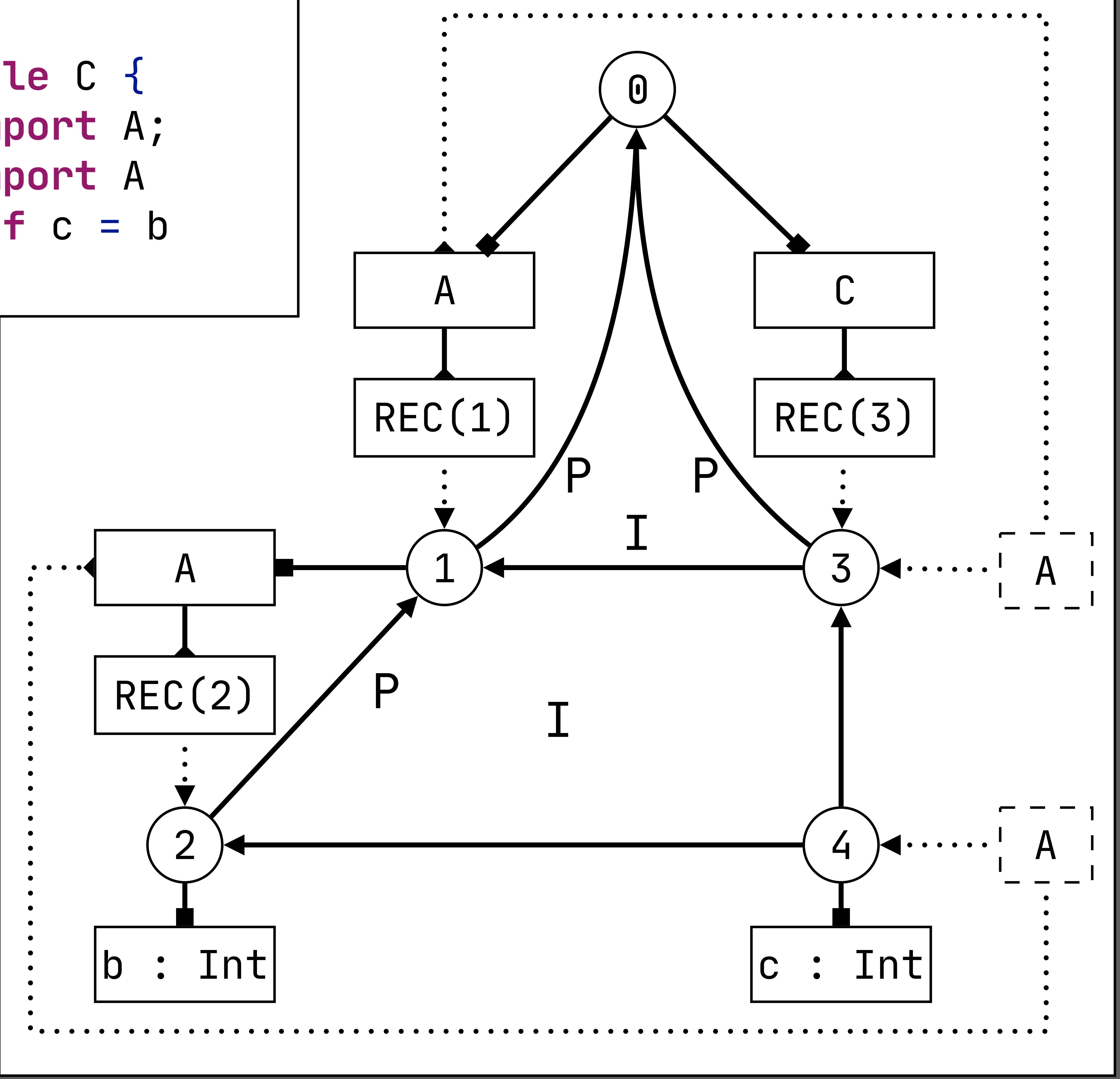


Scoping Imports

```
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
```

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

```
module A {
  module A {
    def b = 1
  }
}
module C {
  import A;
  import A
  def c = b
}
```



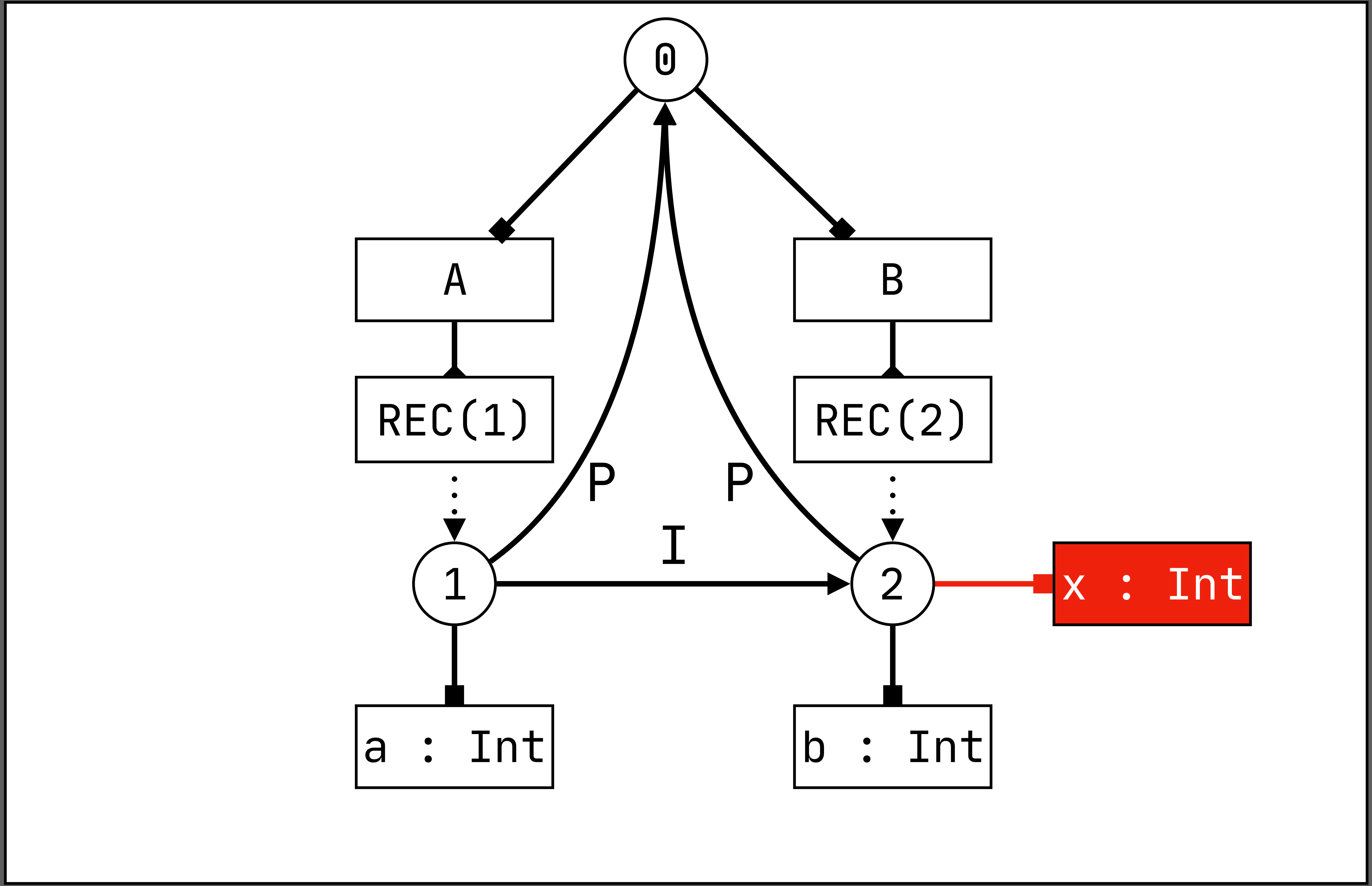
Permission to Extend

Permission to Extend

```
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,
    declareVar(s_mod, "x", INT()).
```

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



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

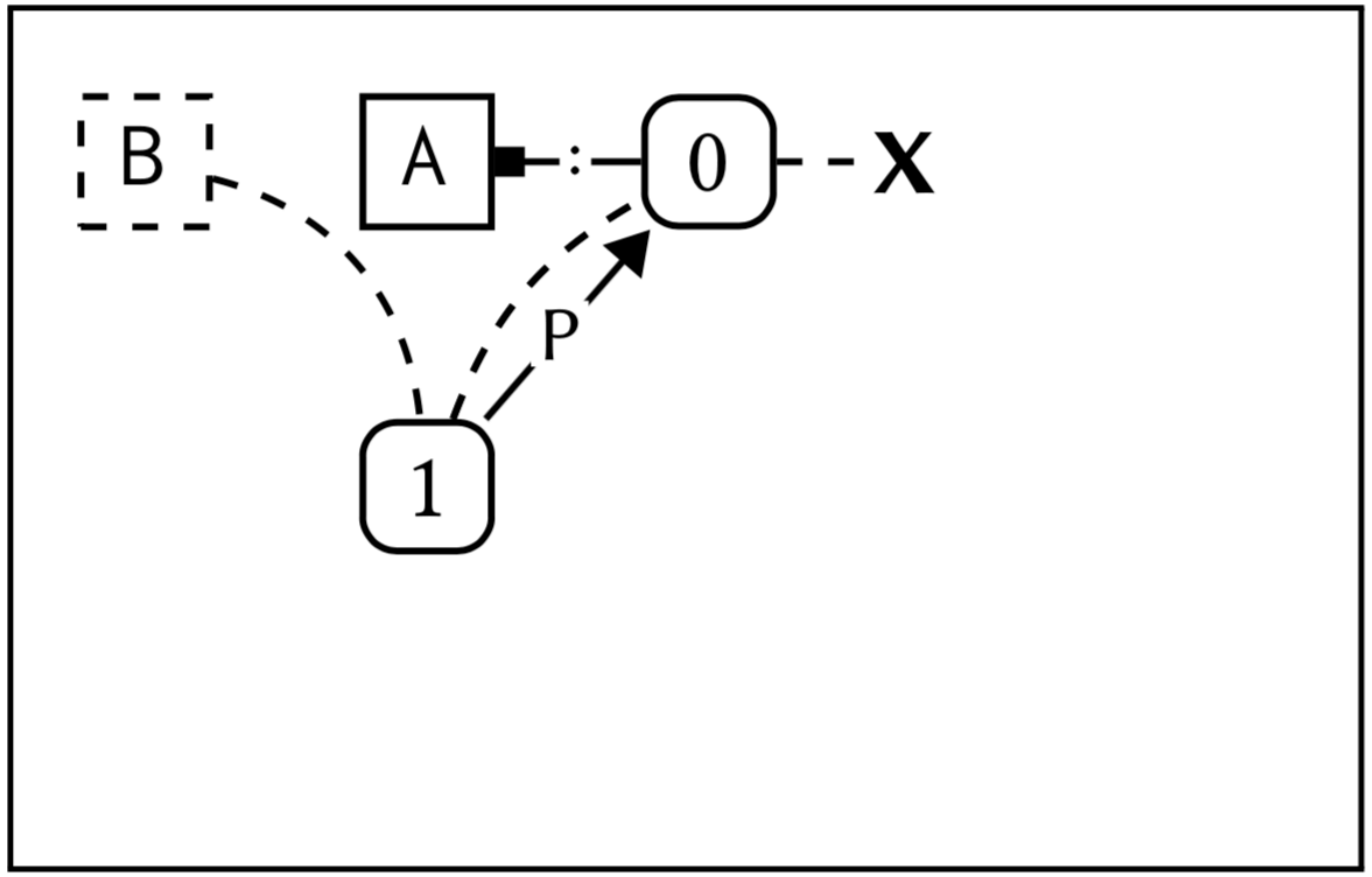
Scope graph construction depends on queries

- Imports require name resolution of module name

When is it safe to query the scope graph?

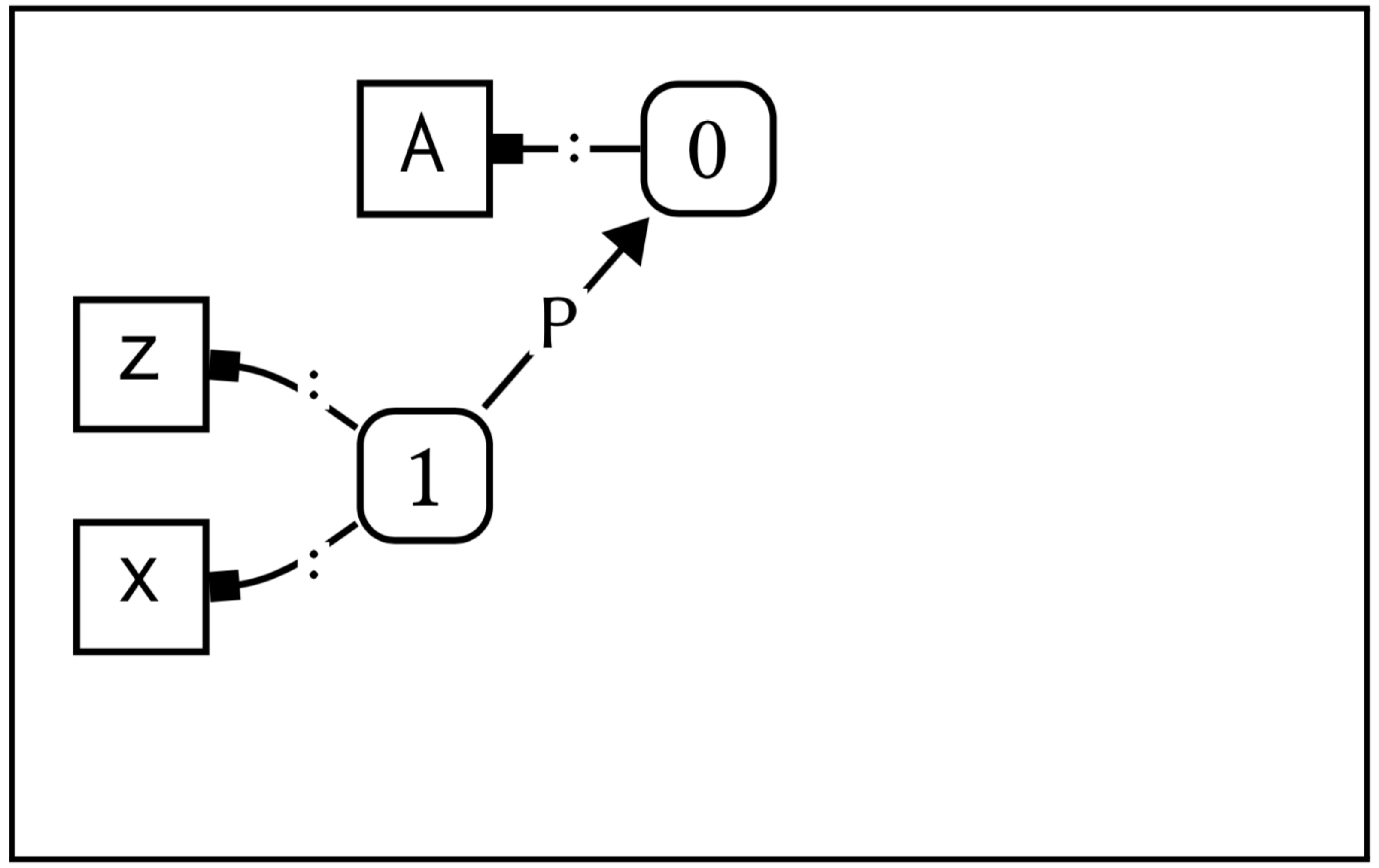
- In what order should type checker perform construction, querying?

A Single Stage Type Checker (Fails)

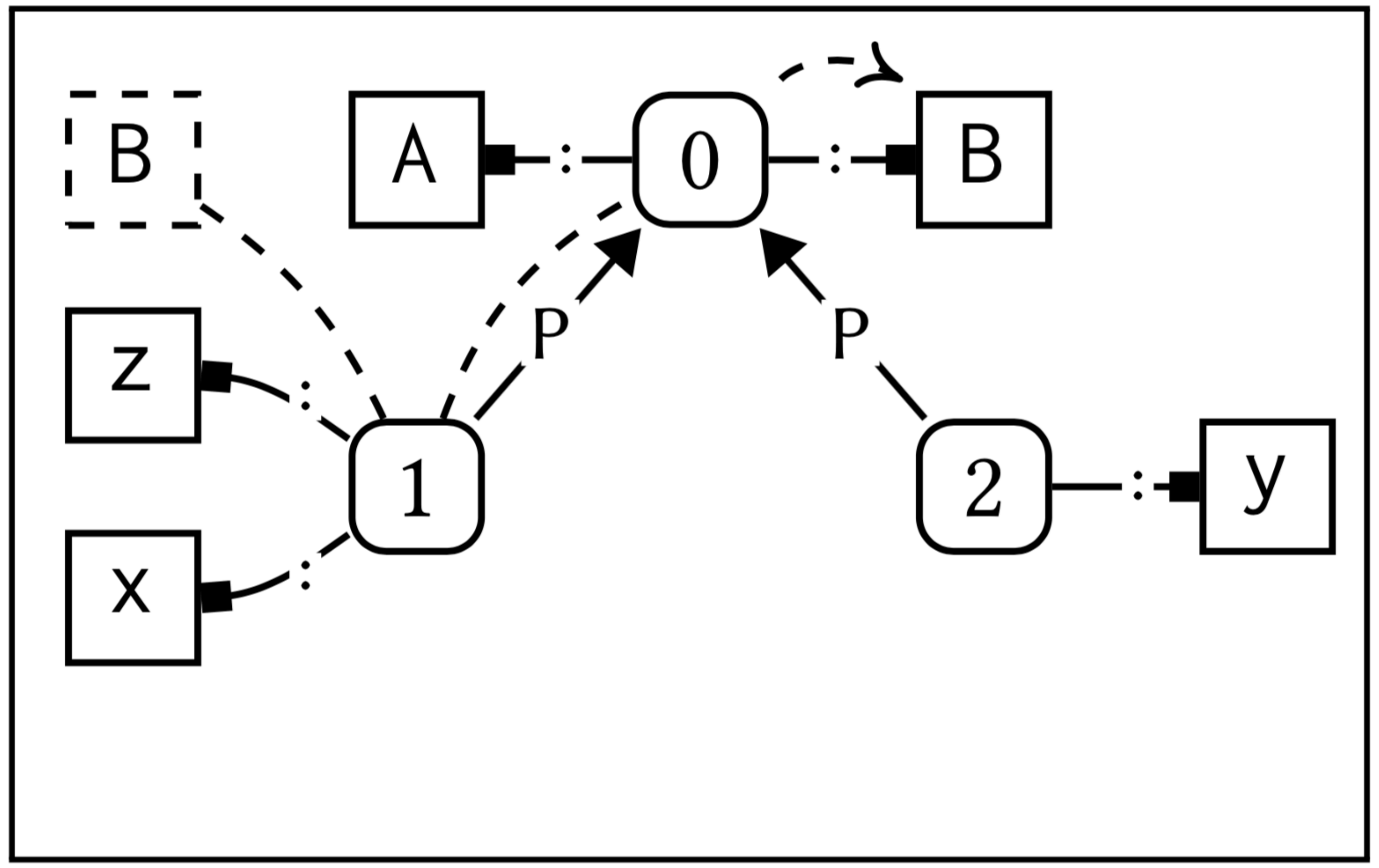


```
module A {
  import B
  def z:int = 3
  def x:int = y + z
}
module B {
  import A
  def y:int = z * 2
}
```

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



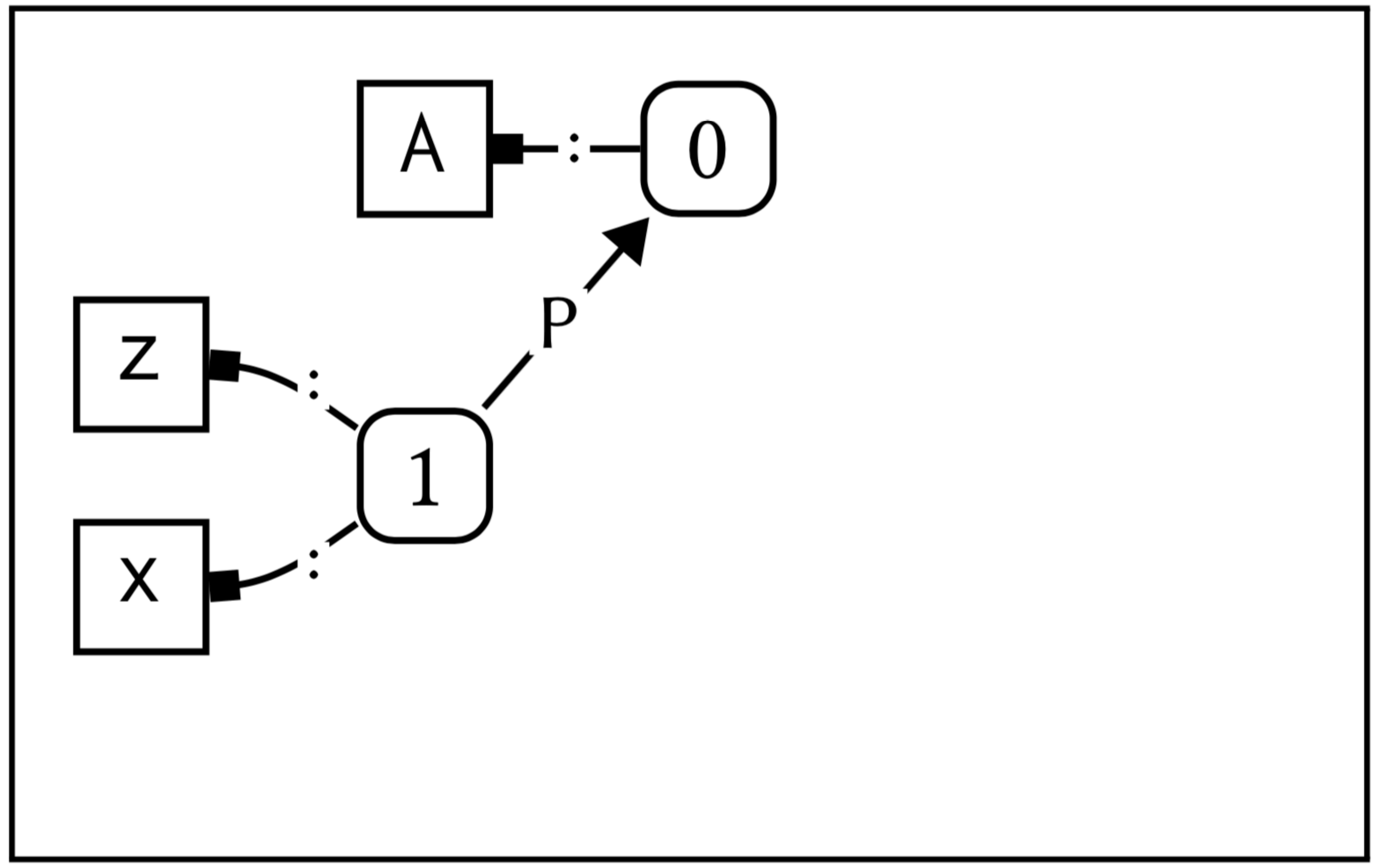
(1)



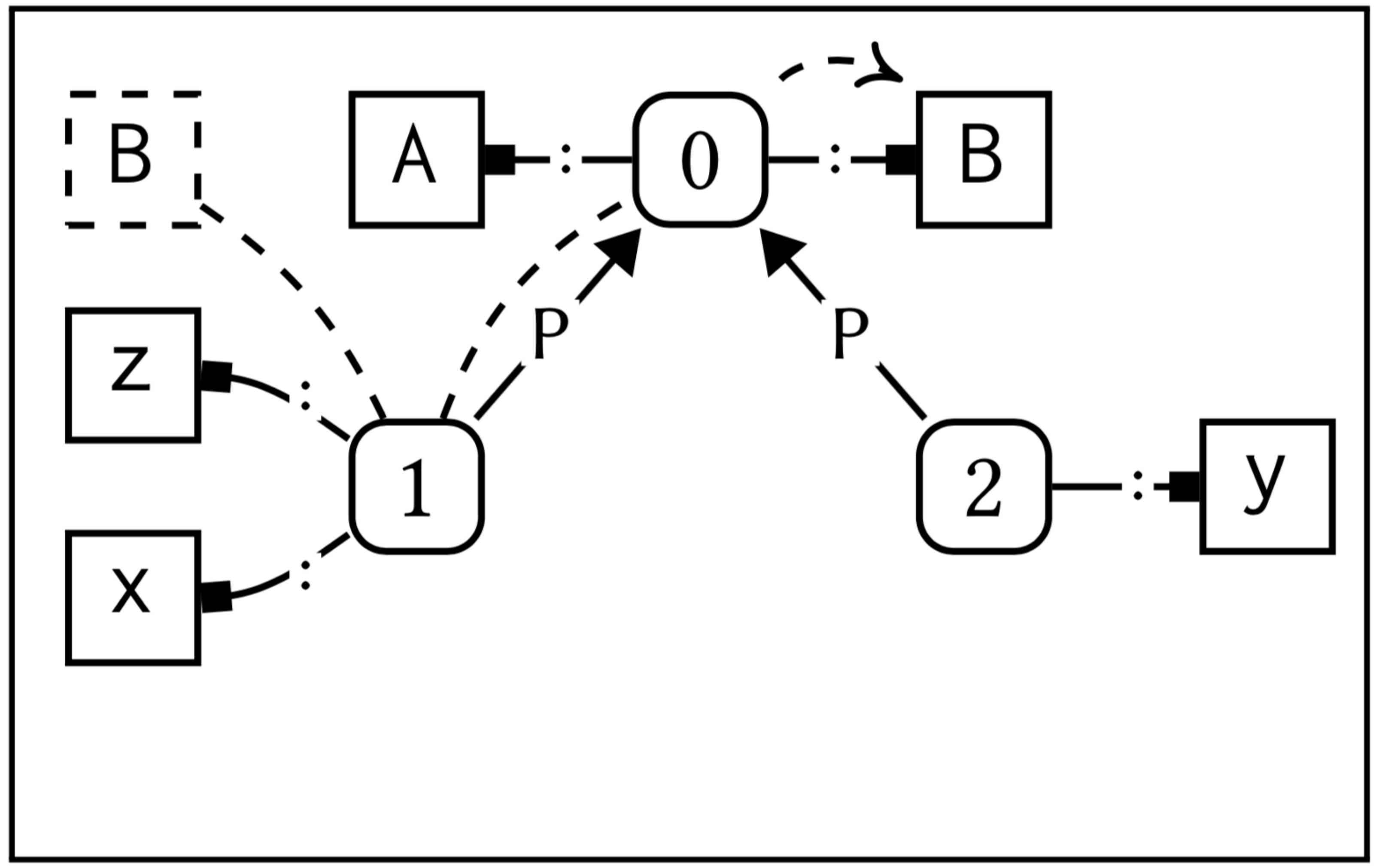
(2)

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

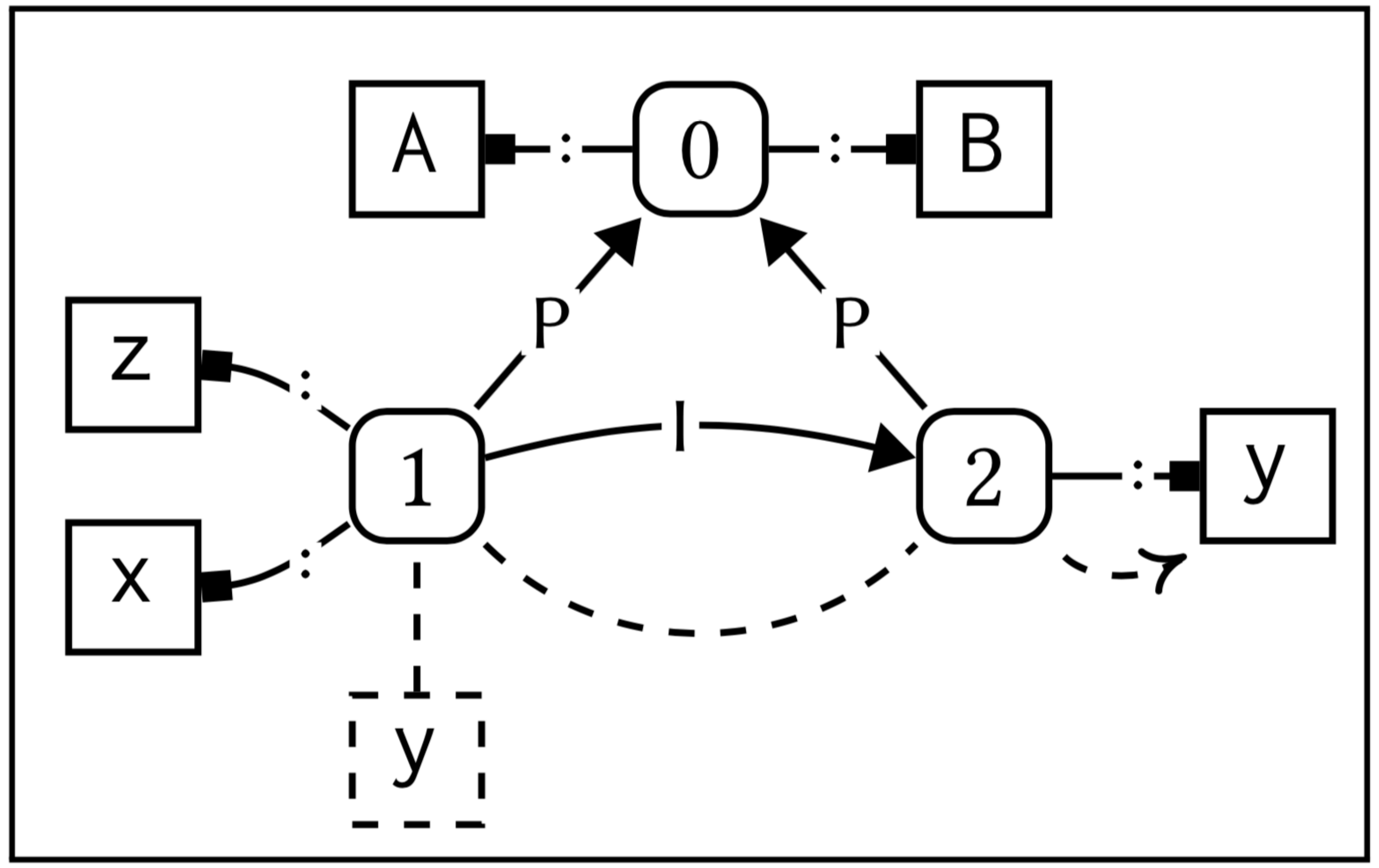
A Two Stage Type Checker: Stage 2 (Check Modules)



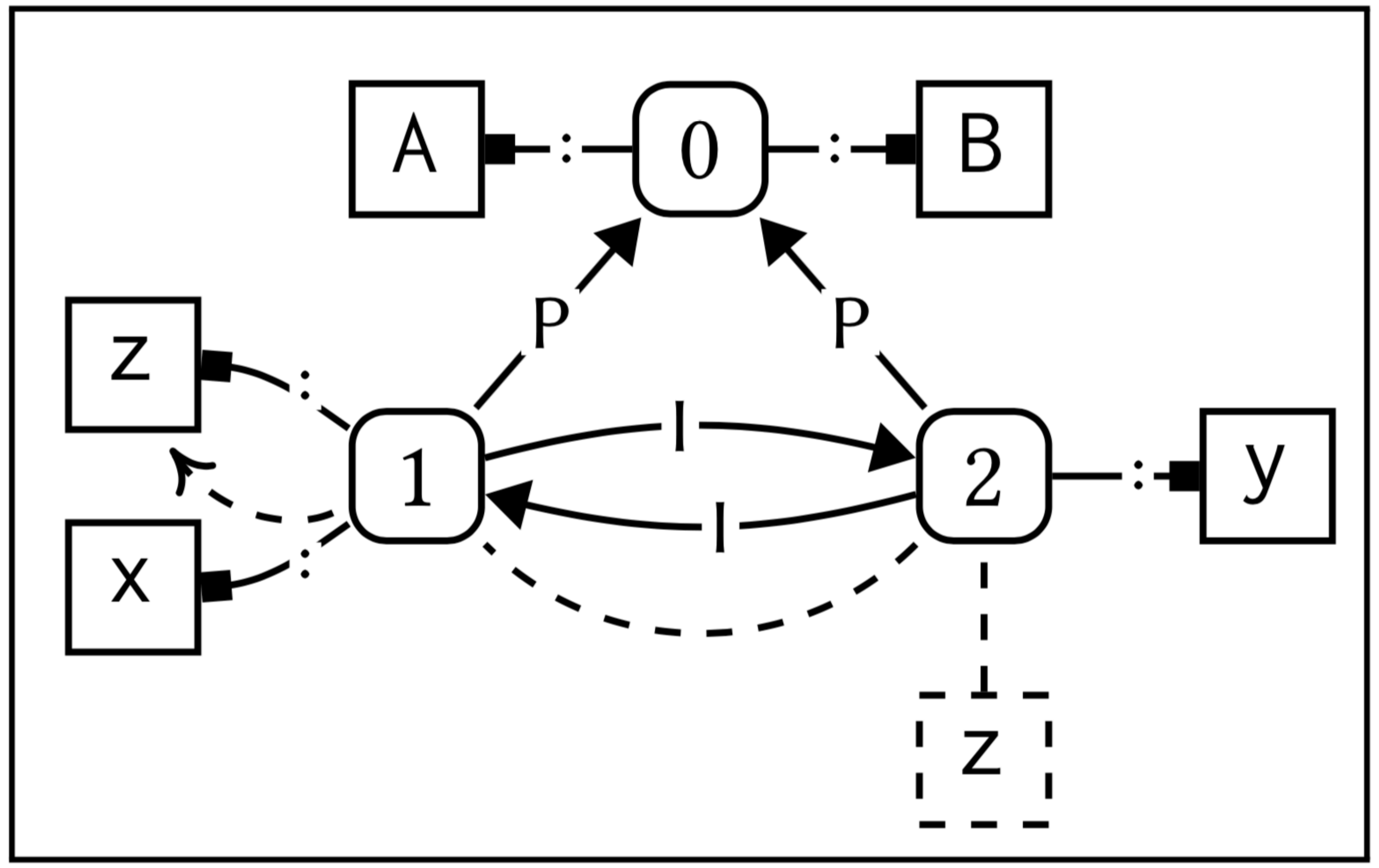
(1)



(2)



(3)

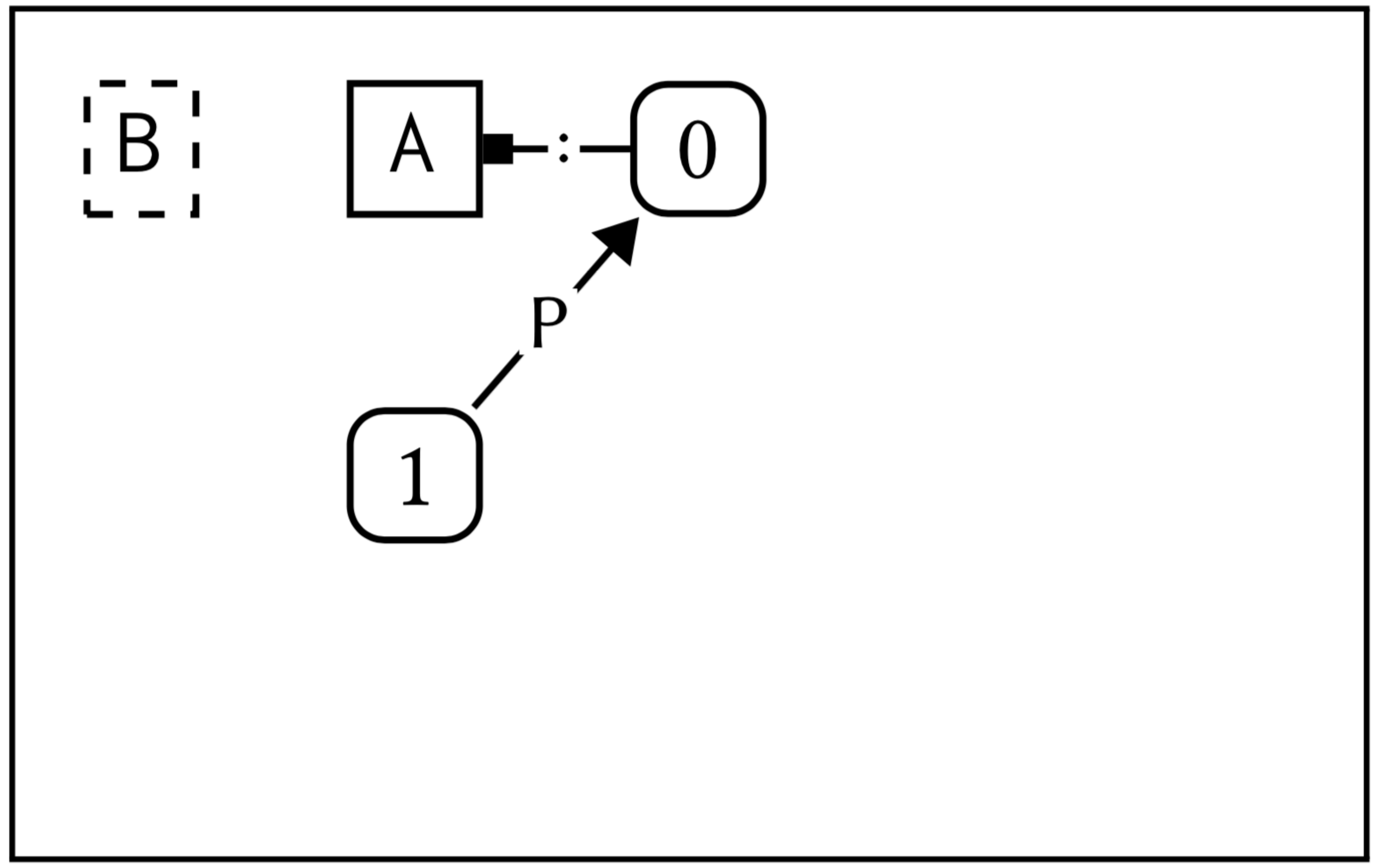


(4)

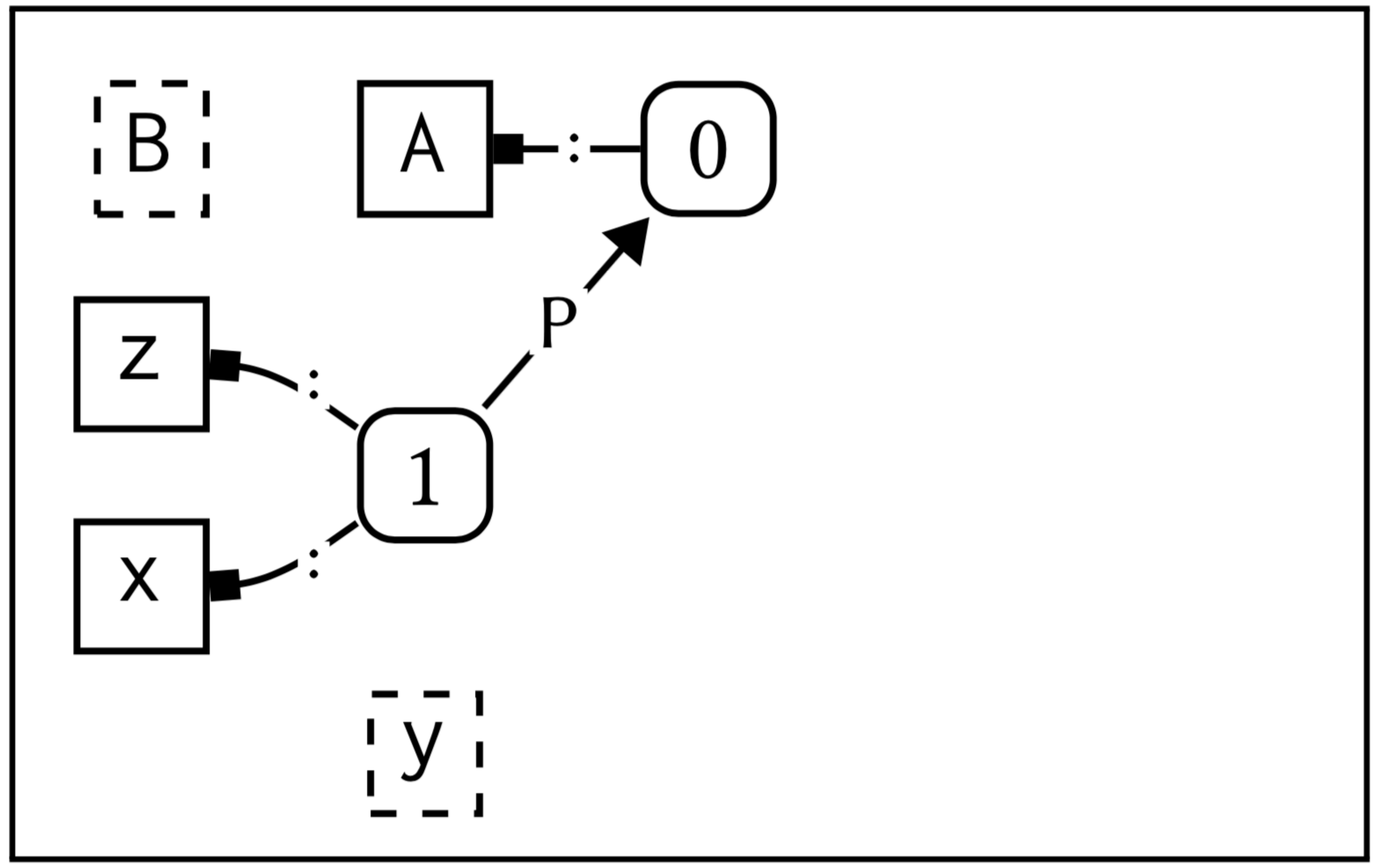
Requires that imports are resolved before variable references

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

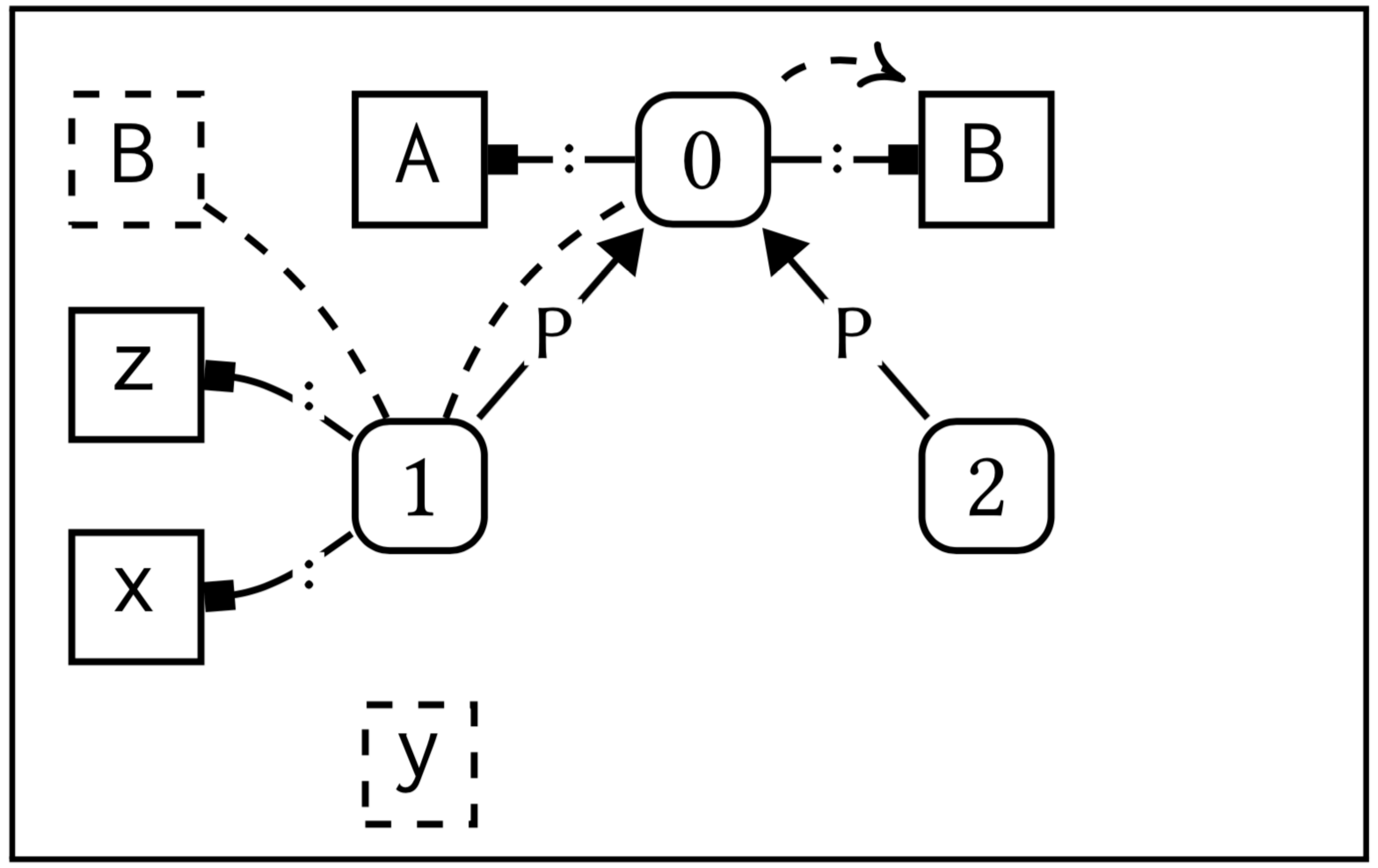
Dynamic



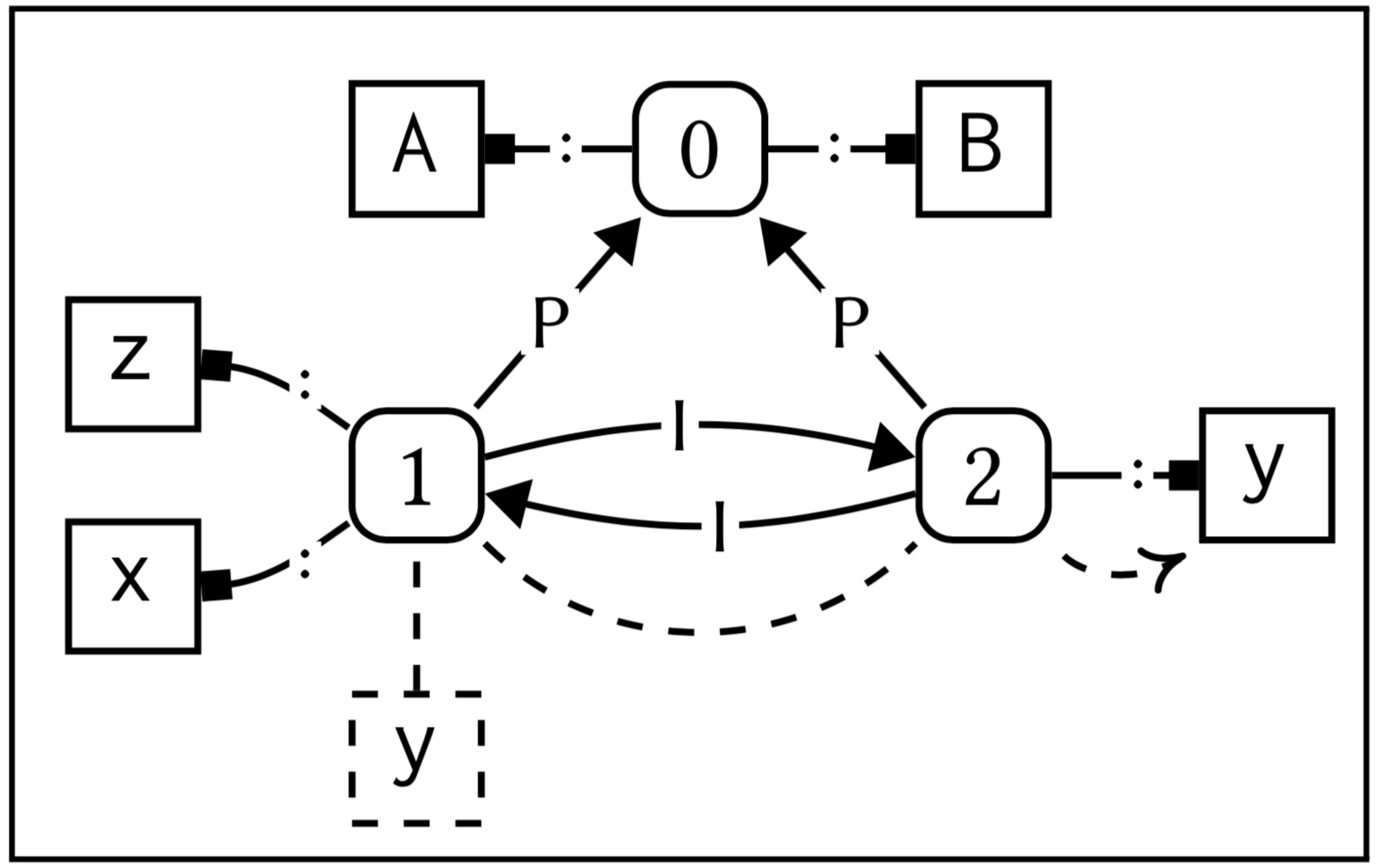
(1)



(2)



(3)

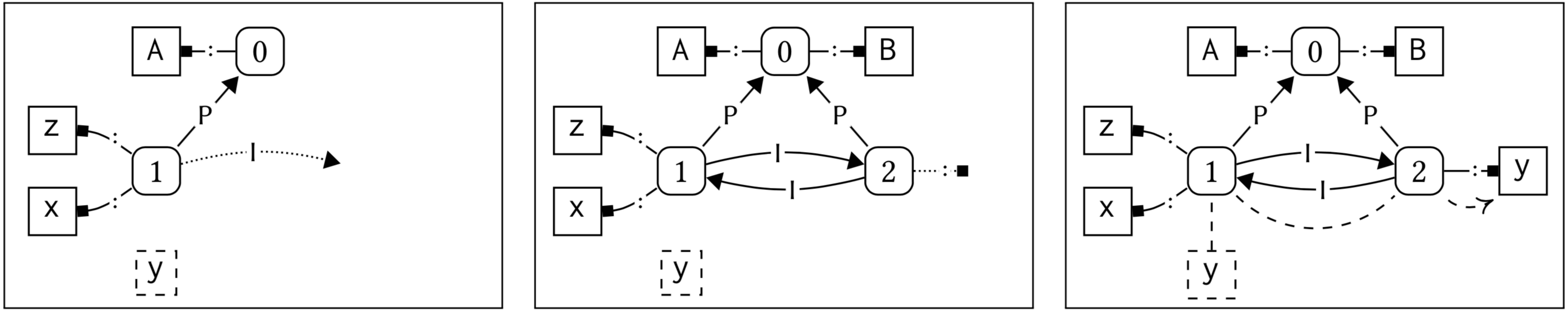


(4)

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

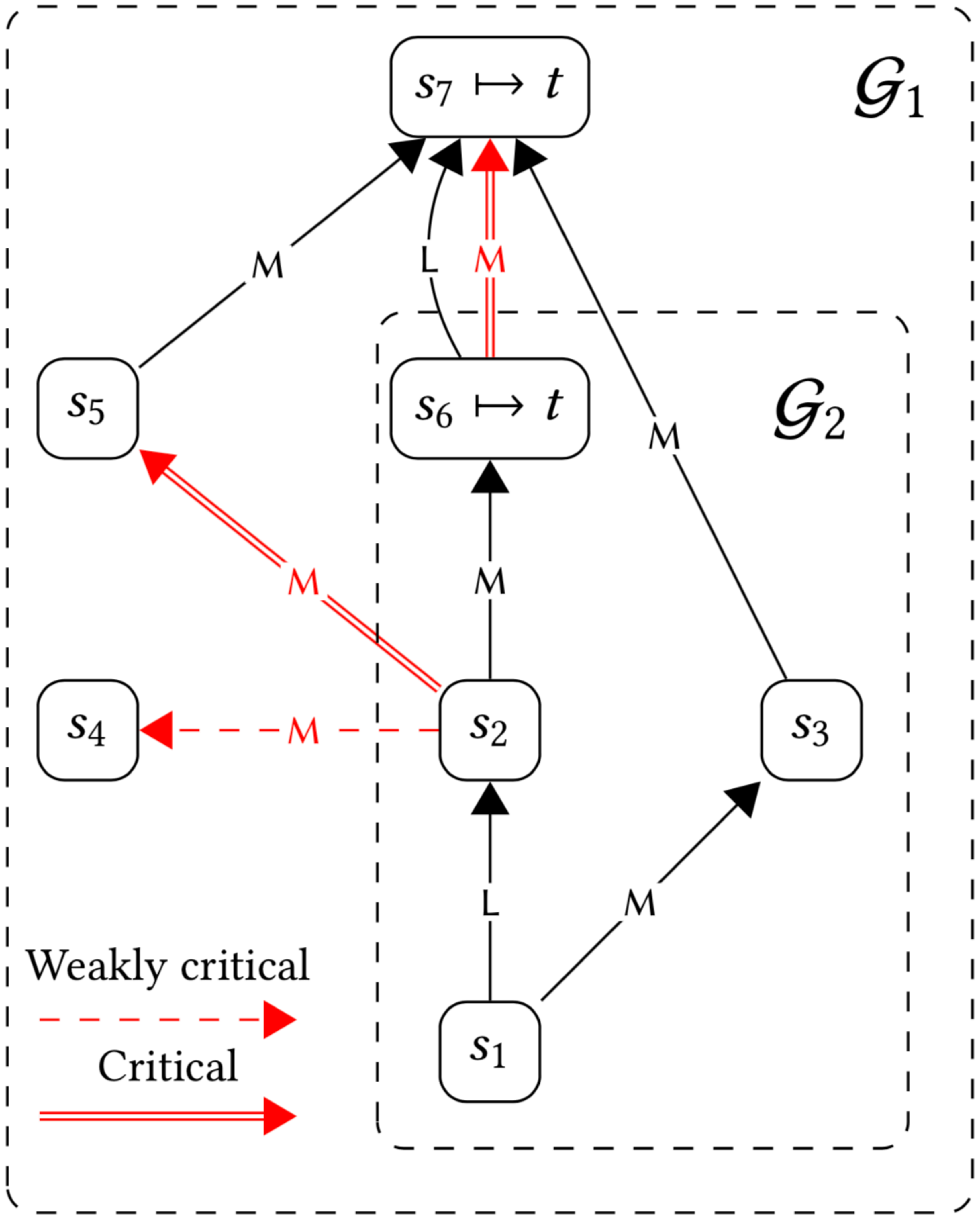


Fig. 11. (Weakly) critical edges for the query $s_1 \overset{LM*}{\twoheadrightarrow}_{\mathrm{R}} D$, assuming $t \in D$

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

- 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

- 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