Scopes as Types

Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser
Delft University of Technology

OOPSLA'18, Boston, MA, USA

Type System Specifications

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 . t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11} . T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

```
let rec typeof ctx t =
  match t with
    TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      let tyT2 = typeof ctx' t2 in
      TyArr(tyT1, typeShift (-1) tyT2)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if (=) tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
  | TmTAbs(fi,tyX,t2) →
      let ctx = addbinding ctx tyX TyVarBind in
      let tyT2 = typeof ctx t2 in
      TyAll(tyX,tyT2)
  | TmTApp(fi,t1,tyT2) →
      let tyT1 = typeof ctx t1 in
      (match tyT1 with
           TyAll(_,tyT12) → typeSubstTop tyT2 tyT12
         | _ → error fi "universal type expected")
```

Declarative
Typing Rules

Executable
Type Checker

Typing Rules $\boxed{\mathcal{G}, s \vdash e : t}$

$$\text{(STLC-Num)}\frac{}{s \vdash z : \mathtt{num}} \qquad \text{(STLC-Plus)}\frac{s \vdash e_1 : \mathtt{num} \qquad s \vdash e_2 : \mathtt{num}}{s \vdash e_1 \ + \ e_2 : \mathtt{num}}$$

$$\text{(STLC-Fun)}\frac{\nabla s_2 \qquad s_2 \xrightarrow{P} s_1 \qquad s_2 \overset{:}{\longrightarrow\!\blacksquare}\ x_i : t_1 \qquad s_2 \vdash e : t_2}{s_1 \vdash \mathbf{fun}\ (x_i\ :\ t_1)\ \{\ e\ \} : t_1\ \texttt{->}\ t_2}$$

$$\text{(STLC-Id)}\frac{DECL(x_i), P^*, \leq_\top, <_l \vdash p : s \overset{:}{\mapsto} x_j : t}{s \vdash x_i : t} \qquad \text{(STLC-App)}\frac{s \vdash e_1 : t_1\ \texttt{->}\ t_2 \qquad s \vdash e_2 : t_1}{s \vdash e_1\ e_2 : t_2}$$

$$\text{(STLC-Let)}\frac{s_1 \vdash e_1 : t_1 \qquad \nabla s_2 \qquad s_2 \xrightarrow{P} s_1 \qquad s_2 \overset{:}{\longrightarrow\!\blacksquare}\ x_i : t_1 \qquad s_2 \vdash e_2 : t_2}{s_1 \vdash \mathbf{let}\ x_i\ =\ e_1\ \mathbf{in}\ e_2 : t_2}$$

Declarative Specification and Name Resolution

Uniform Binding Representation

Abstraction over Execution Order

Binding: Lexical

```
let
  x : int = 7
in
  let
    f : int -> int = fun (x:int) { x * 3 }
  in
    f x
```

shadowed

Representation

- Typing environment
- Ordered list of name-types

Execution order

- Constructed top-down

Binding: Structural Records

```
let
   p : { x:int, y:int } = { y = -1, x = 2 }
in
   p.y
```

Representation

- Unordered map of fields-types

Execution order

- Interleaved type checking and name resolution

In general:

- Types expose the scope structure of the underlying data
- Often language-specific representations (e.g., class types)

Binding: Modules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```

Representation

- Global module table (MT)
- Name-interface pairs
- Often language-specific

Execution order

- Staged MT construction and module body checking

Common Binding Representations in Specifications

Binding Representation

Many different representations

Often language-specific

Ad-hoc, not reusable

Execution Order

Interleaving

Staging

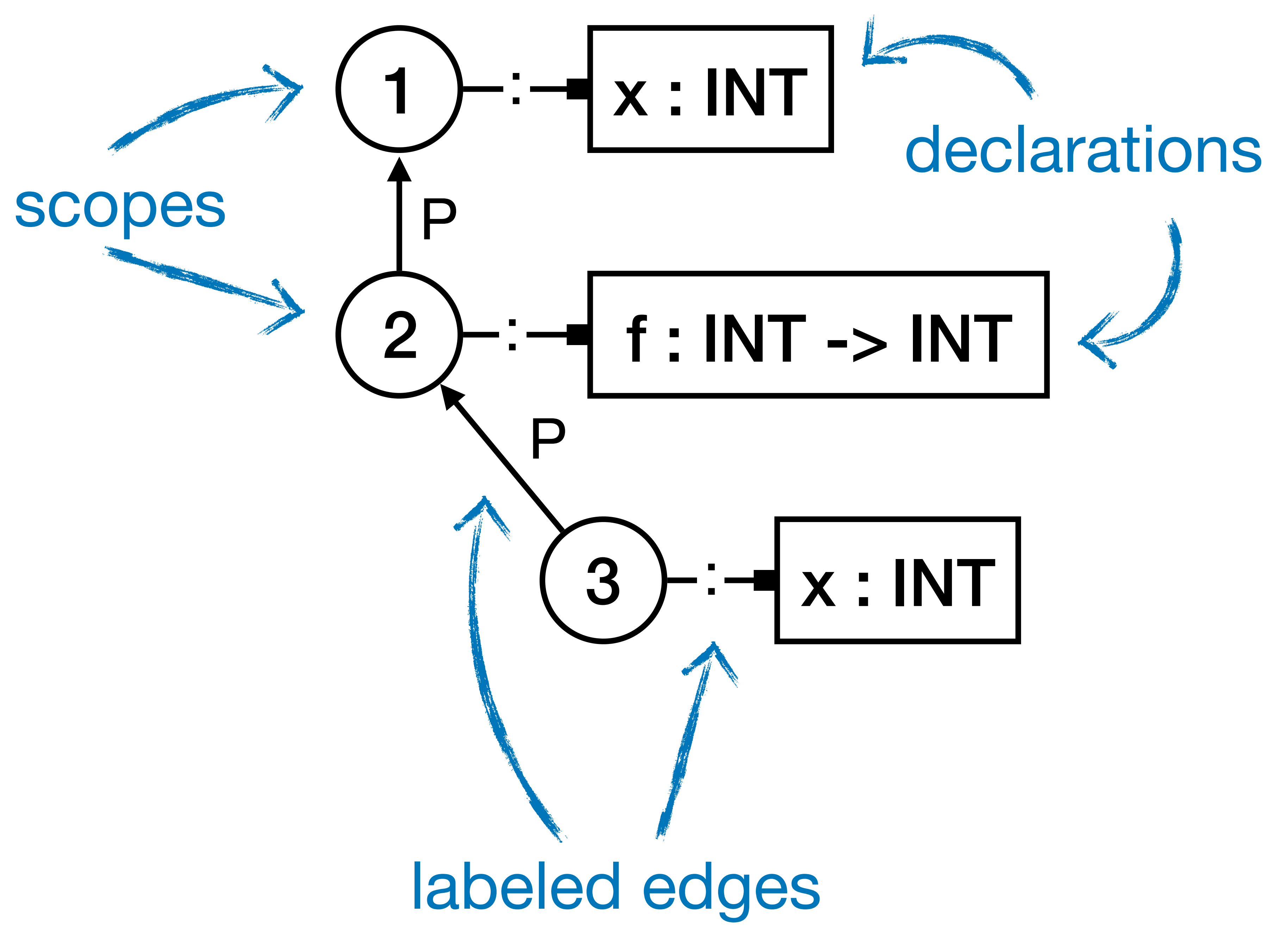
Not declarative

Our Approach

Scope Graphs	Statix

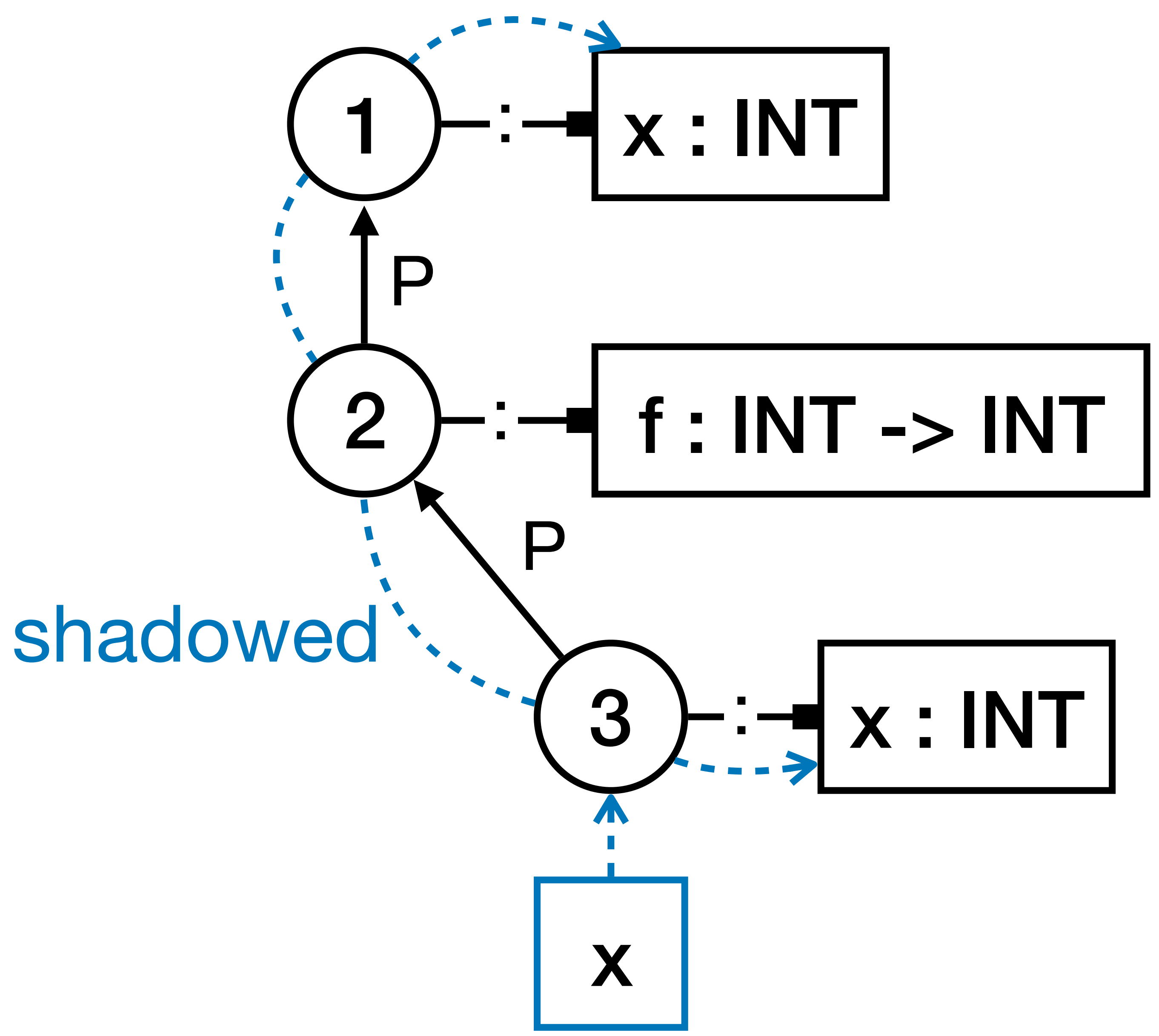
Scope Graph: Lexical

```
let
  x : int = 7
in
  let
    f : int -> int = fun (x:int) { x * 3 }
  in
    f x
```



Scope Graph: Lexical

```
let
  x : int = 7
in
                shadowed
  let
    f : int -> int = fun (x:int) { x * 3 }
  in
    f x
```



Query

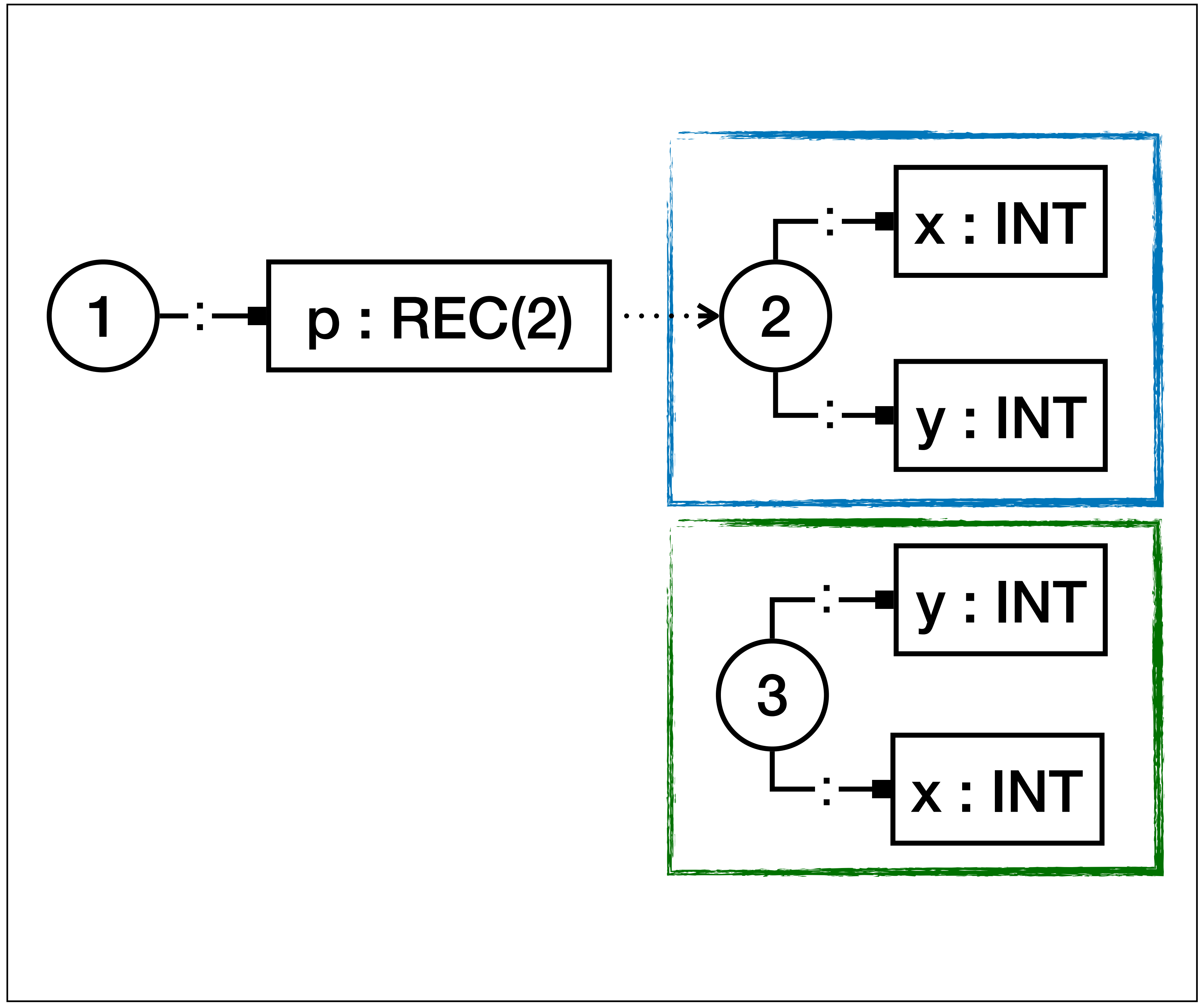
P* (allow any number of P steps)

$ < P (prefer local declarations over P-steps)

- Name resolution = querying the graph
- Visibility and shadowing = regular expression and order over edge labels

Scope Graph: Records

```
let
  p : { x:int, y:int } = { y = -1, x = 2 }
in
  p.y
```

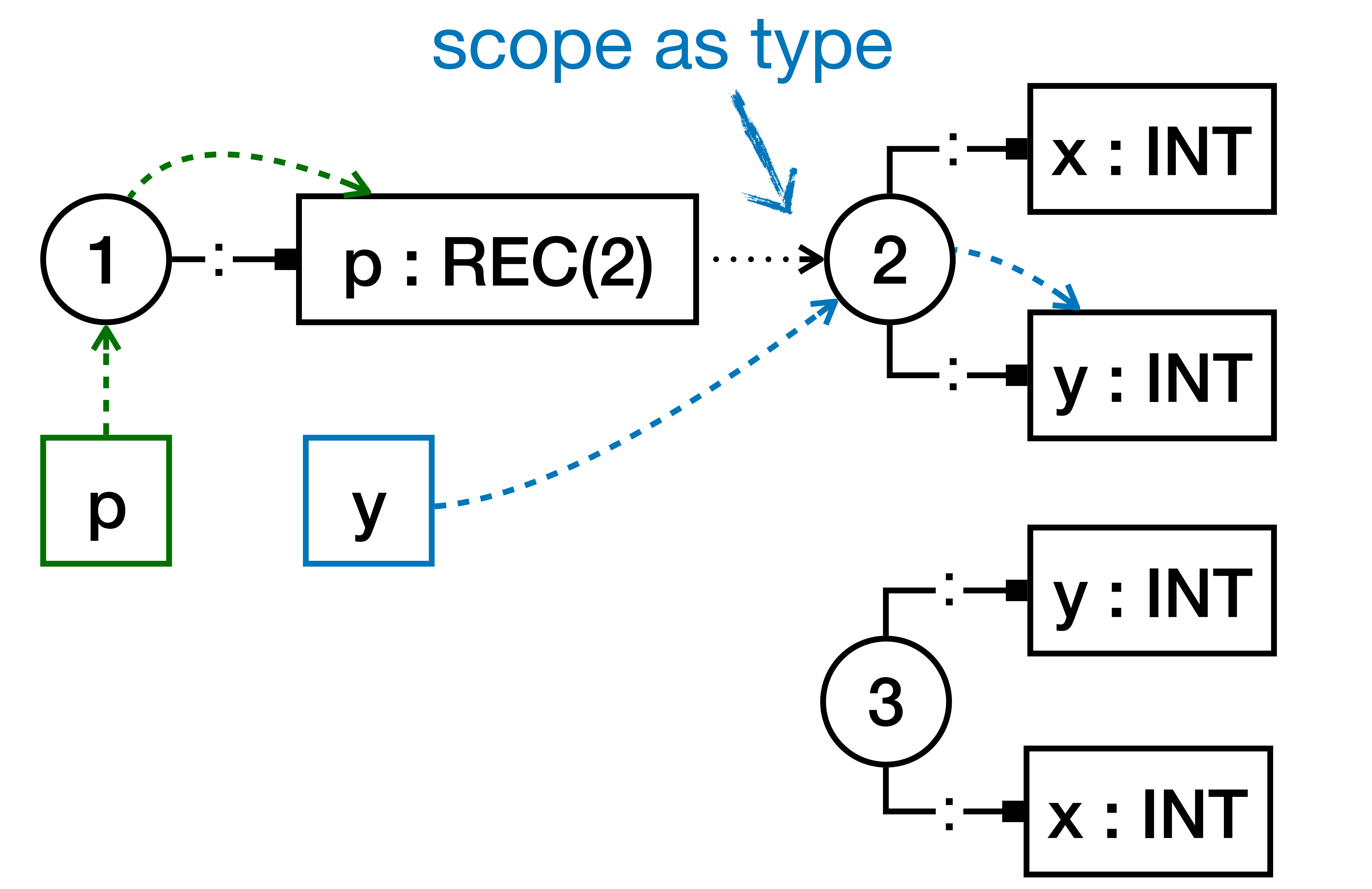


- Type structure described by scopes

Scope Graph: Records

```
let
  p : { x:int, y:int } = { y = -1, x = 2 }
in
  p.y
```

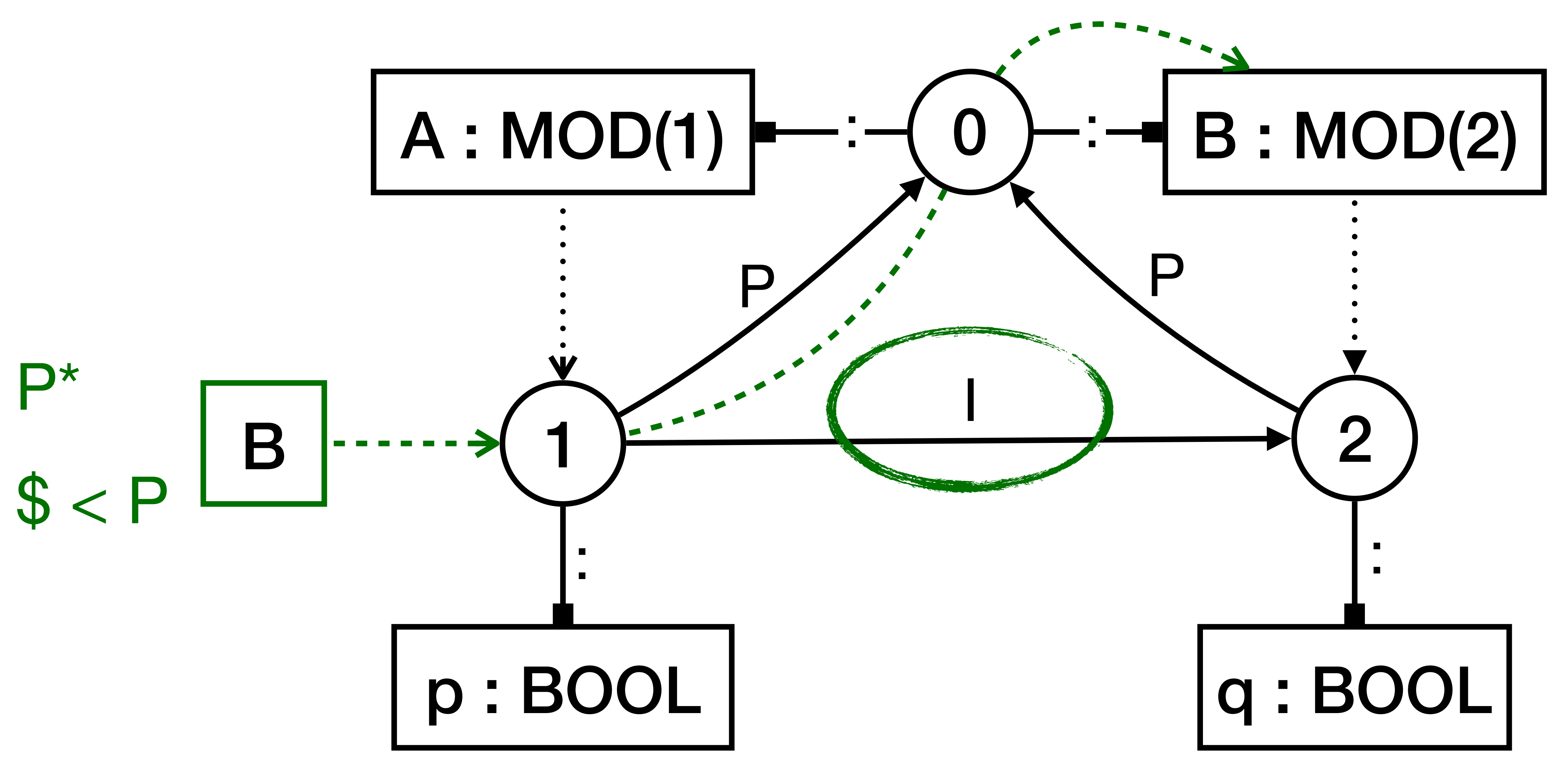
type-dependent
name resolution



- Type structure described by scopes
- Scopes as types = uniform approach to type-dependent name resolution!

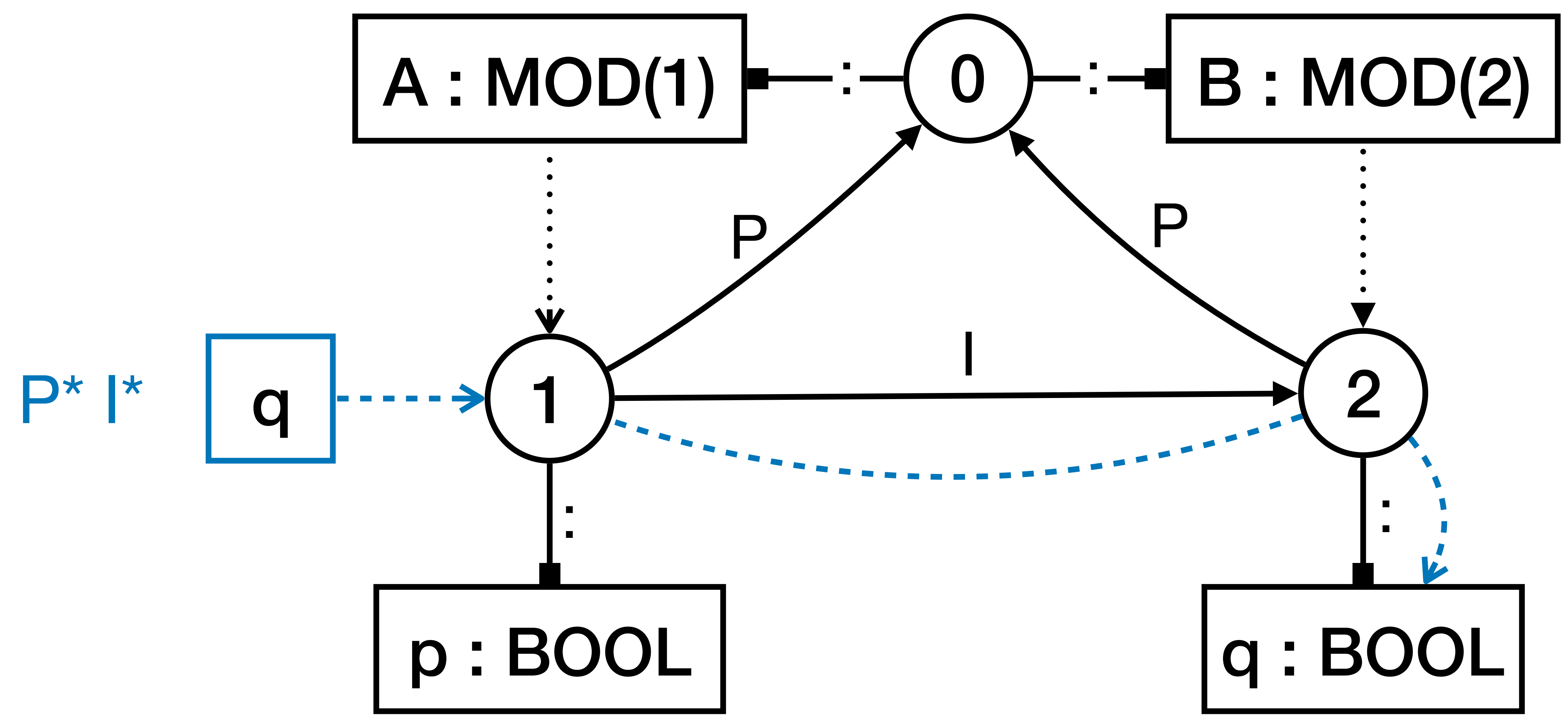
Scope Graph: Modules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```



Scope Graph: Modules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```

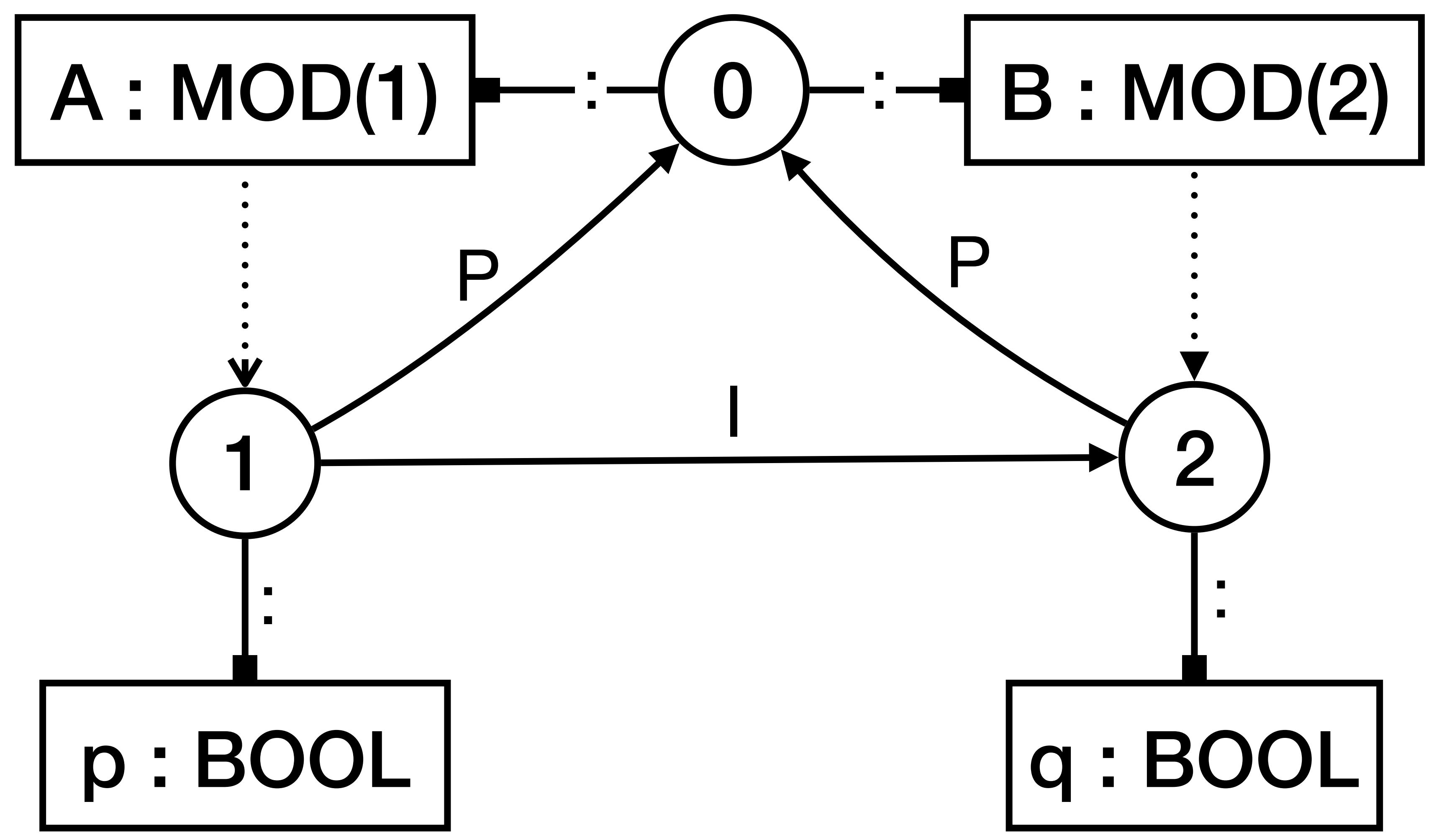


Our Approach

Scope Graphs	Statix
Language independent Capture different kinds of binding Resolve names by graph queries	

Statix: Declarative Rules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```



scope graph assertions

$$\frac{\nabla s_m \qquad s \xrightarrow{:} \blacksquare\ \mathtt{x}_i : \mathrm{MOD}(s_m) \qquad s_m \xrightarrow{\mathsf{P}} s \qquad s_m \vdash \overline{\mathtt{stm}}\ \mathrm{OK}}{s \vdash \mathtt{module}\ \mathtt{x}_i\ \{\ \overline{\mathtt{stm}}\ \}\ \mathrm{OK}}$$

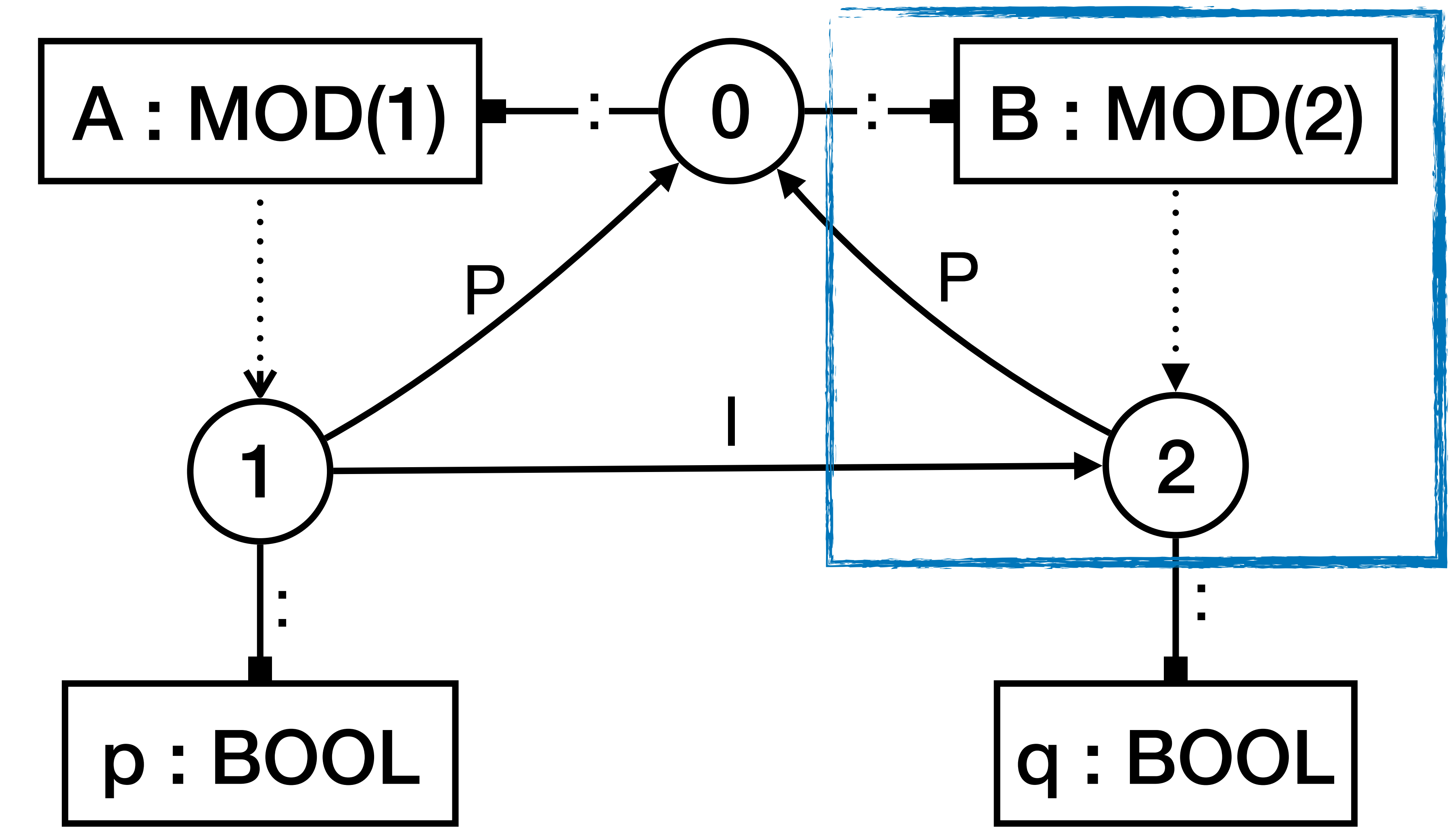
syntax-directed rules

scope graph queries

$$\frac{\mathrm{DECL}(\mathtt{x}_i), \mathsf{P}^* \vdash p : s \overset{:}{\mapsto} \mathtt{x}_j : \mathrm{MOD}(s_m) \qquad s \xrightarrow{\mathsf{I}} s_m}{s \vdash \mathtt{import}\ \mathtt{x}_i\ \mathrm{OK}}$$

Statix: Declarative Rules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```

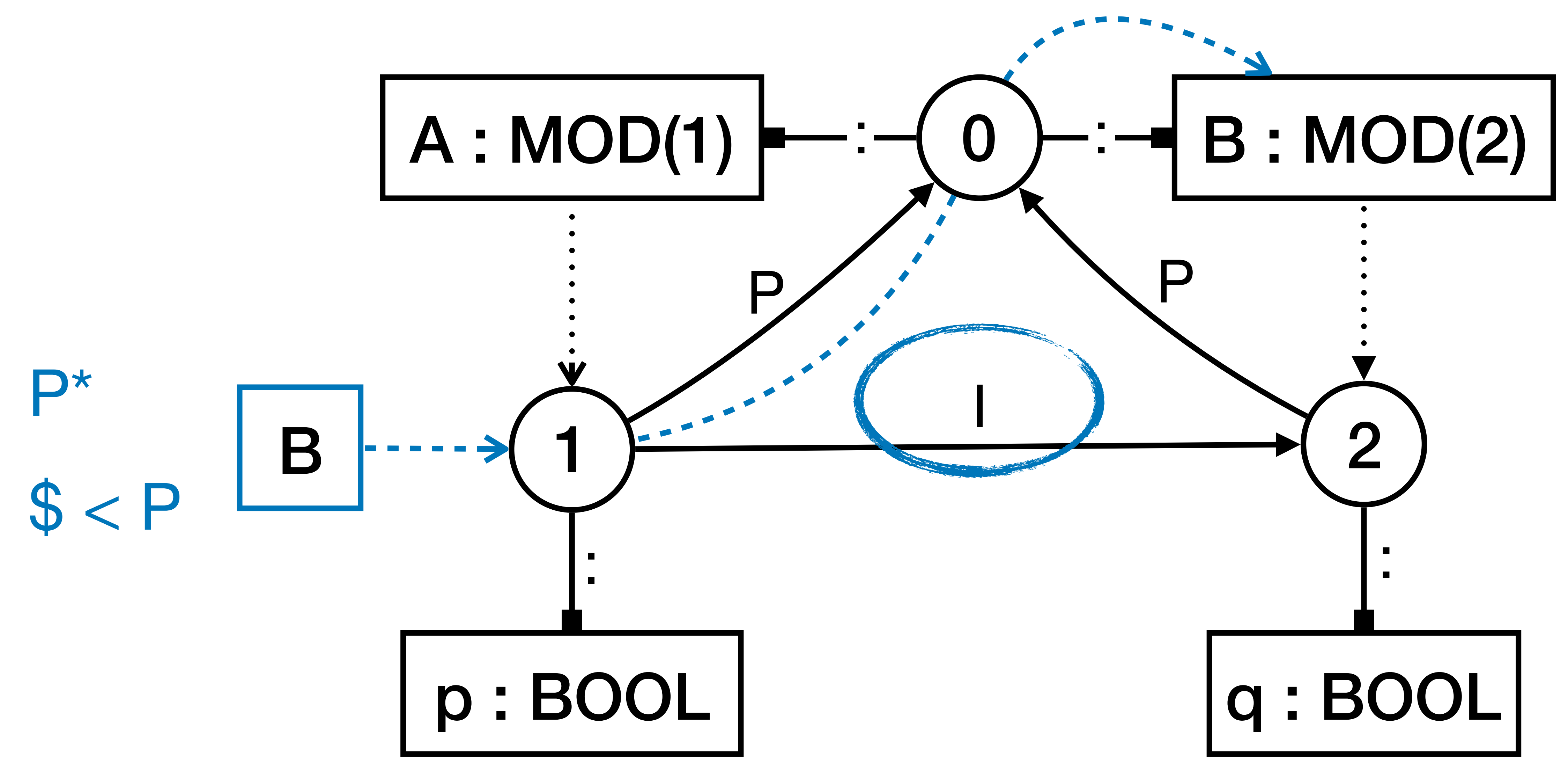


$$\frac{\nabla s_m \qquad s \xrightarrow{:} \blacksquare\ \mathtt{x}_i : \mathrm{MOD}(s_m) \qquad s_m \xrightarrow{\mathsf{P}} s \qquad s_m \vdash \overline{\mathtt{stm}}\ \mathrm{OK}}{s \vdash \mathtt{module}\ \mathtt{x}_i\ \{\ \overline{\mathtt{stm}}\ \}\ \mathrm{OK}}$$

$$\frac{\mathrm{DECL}(\mathtt{x}_i), \mathsf{P}^* \vdash p : s \stackrel{.}{\mapsto} \mathtt{x}_j : \mathrm{MOD}(s_m) \qquad s \xrightarrow{\mathsf{I}} s_m}{s \vdash \mathtt{import}\ \mathtt{x}_i\ \mathrm{OK}}$$

Statix: Declarative Rules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```

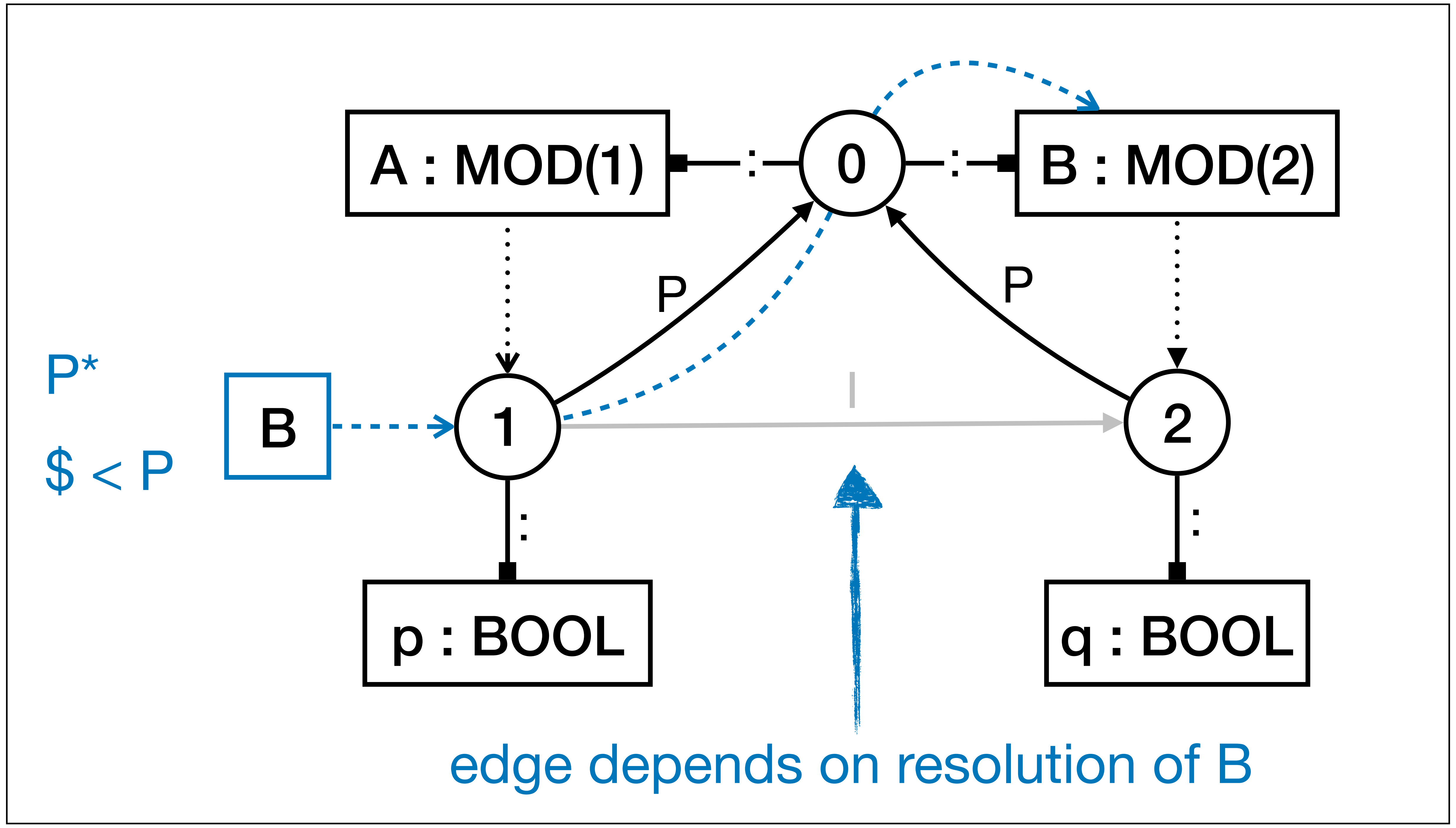


$$\frac{\nabla s_m \qquad s \xrightarrow{:} \mathtt{x}_i : \mathrm{MOD}(s_m) \qquad s_m \xrightarrow{\mathsf{P}} s \qquad s_m \vdash \overline{\mathtt{stm}}\ \mathrm{OK}}{s \vdash \mathtt{module}\ \mathtt{x}_i\ \{\ \overline{\mathtt{stm}}\ \}\ \mathrm{OK}}$$

$$\frac{\boxed{\mathrm{DECL}(\mathtt{x}_i), \mathsf{P}^* \vdash p : s \overset{:}{\mapsto} \mathtt{x}_j : \mathrm{MOD}(s_m)} \qquad \boxed{s \xrightarrow{\mathsf{I}} s_m}}{s \vdash \boxed{\mathtt{import}\ \mathtt{x}_i}\ \mathrm{OK}}$$

Interleave Graph Construction and Queries

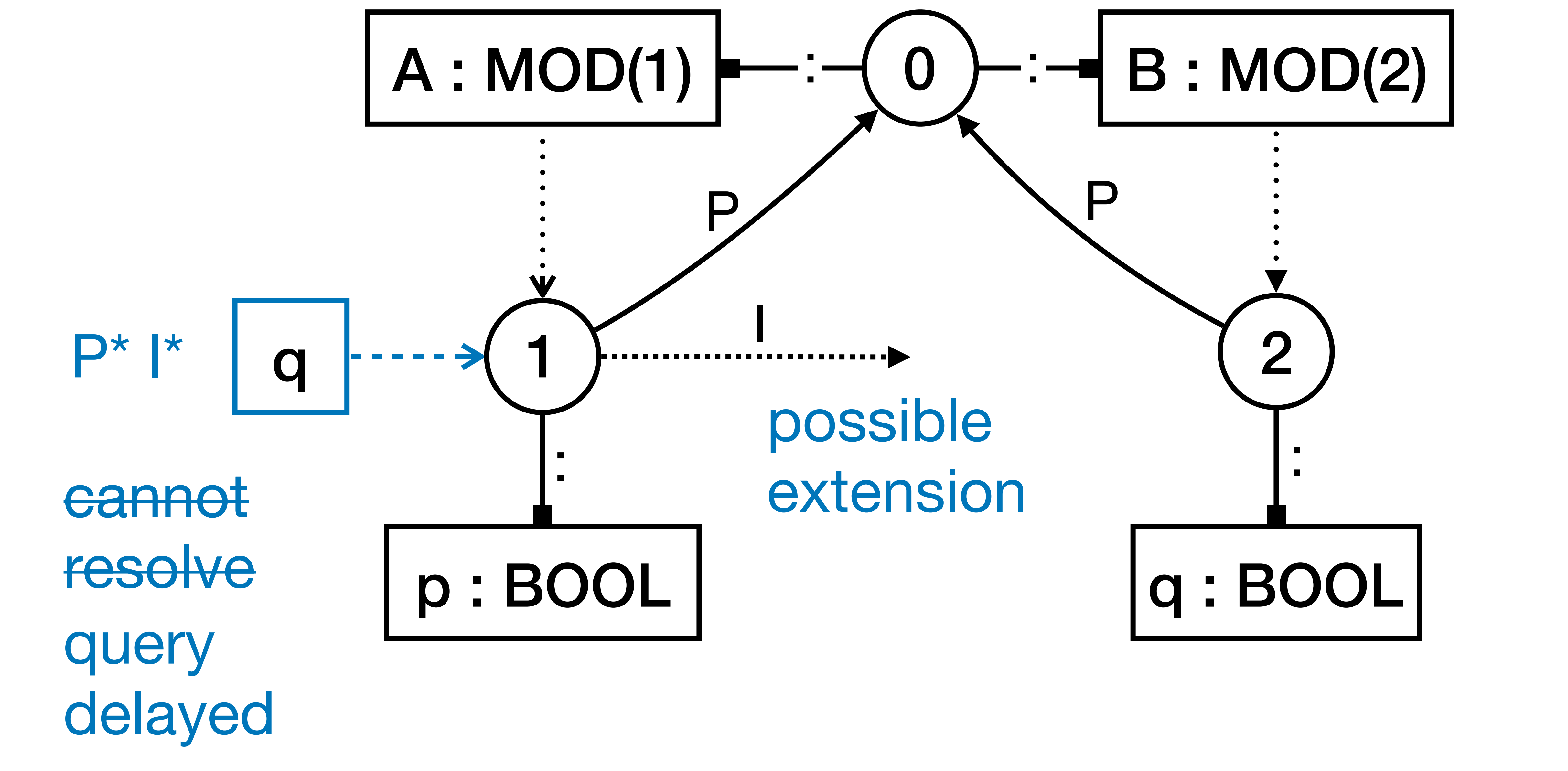
```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```



- Interleave scope graph construction and querying
- Query in incomplete graph? If the result holds in final graph!

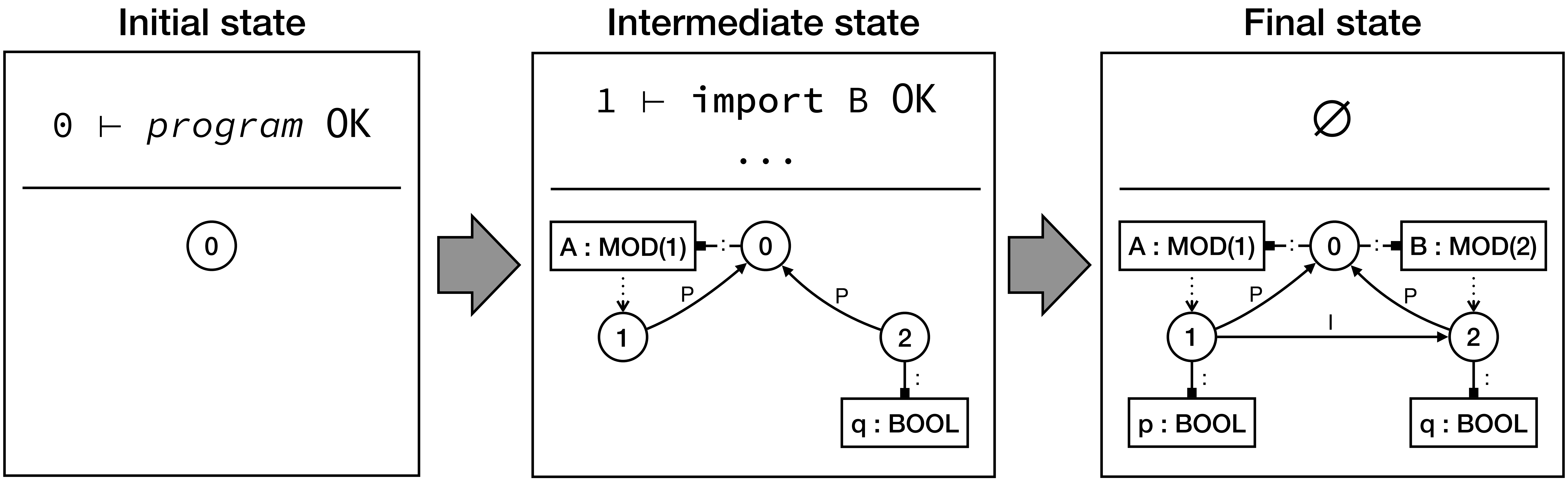
Queries in Incomplete Graphs

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```



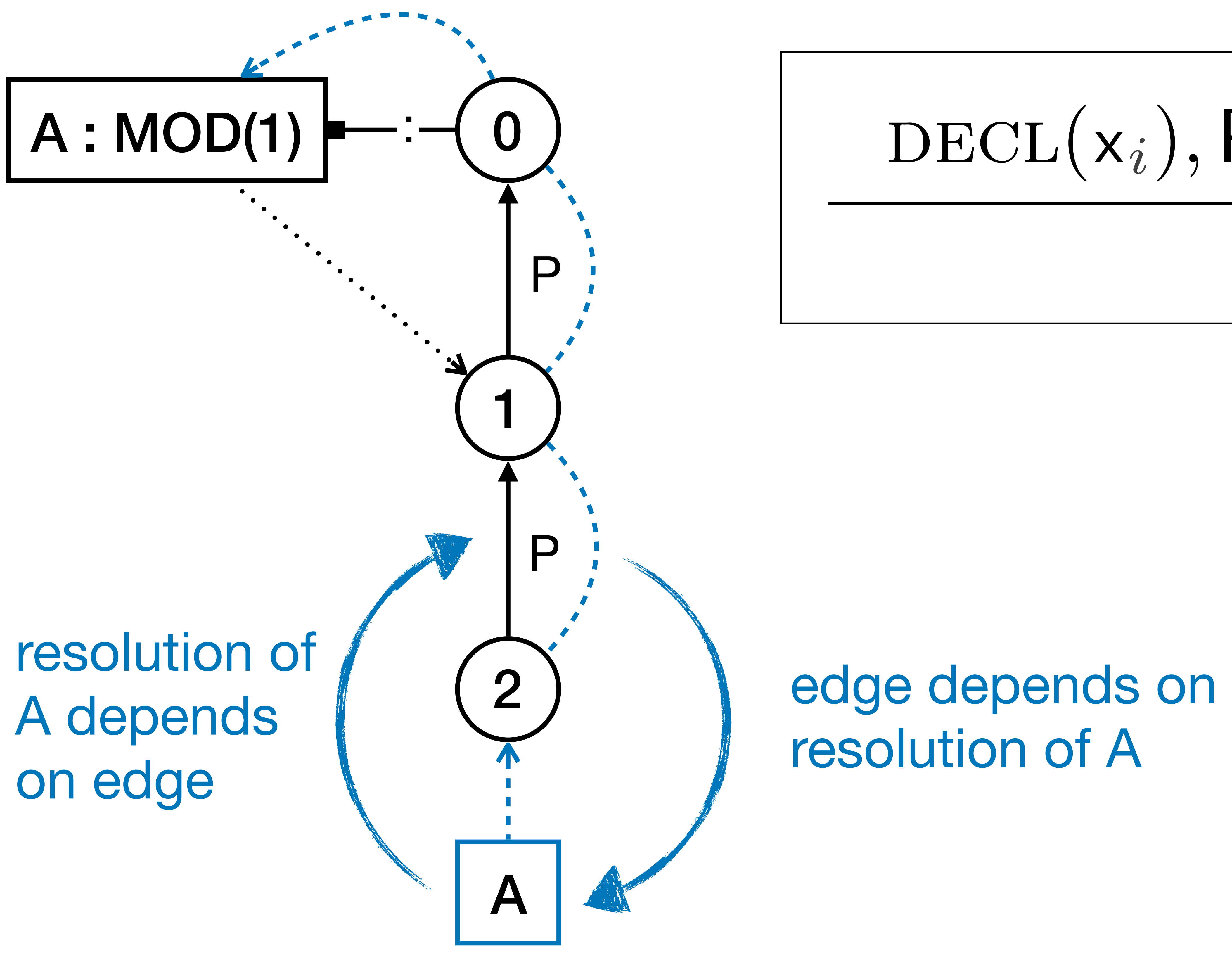
- Query attempt in incomplete graph
- Approximate possible edge extensions
- Delay if extension may change result

Execution by Rewriting



- Non-deterministic constraint solving
- Committed choice, no backtracking
- Rule-based simplification
- Ensure correct scope graph queries

Unorderable Constraints get Stuck



$$\frac{\textsc{decl}(\mathtt{x}_i), \mathsf{P}^* \vdash p : s \mapsto \mathtt{x}_j : \textsc{mod}(s_m) \qquad s \xrightarrow{\mathsf{P}} s_m}{s \vdash \mathtt{stuck}\ \ \mathtt{x}_i\ \textsc{ok}}$$

Our Approach

Scope Graphs

Language independent

Capture different kinds of binding

Resolve names by graph queries

Statix

Declarative specifications

Abstracts over execution order

Constraint solver

Implementation

Program

```
type point = {x : num, y : num} in
let  mkpoint = fun(x : num) { {x = x, y = x} } in
type color = num in
type colorpoint =
       {k : color} extends point in
let  addColor =
       fun(c : num) {
         fun(p : colorpoint) {
           ({c = c} extends p) : colorpoint
         }
       } in
     (addColor 6 ({c = 5} extends mkpoint 4)) : colorpoint
```

Statix Specification

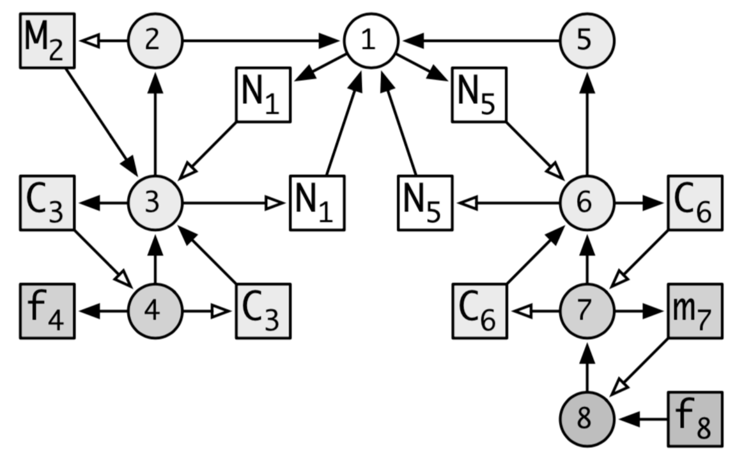
```
typeOfExp : scope * Exp ->  Type

typeOfExp(s, Num(_)) = NUM().

typeOfExp(s, Plus(e1, e2)) = NUM() :-
  typeOfExp(s, e1) == NUM(),
  typeOfExp(s, e2) == NUM().

typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}
  typeOfTypeExp(s, te) == S,
  new s_fun, s_fun -P-> s,
  s_fun -> Var{x@x} with typeOfDecl S,
  typeOfExp(s_fun, e) == T.

typeOfExp(s, Var(x)) = T :-
  query typeOfDecl filter pathMatch[P*(R|E)*] and { d :- varOrFld(x, d) }
                   min pathLt[$ < P, $ < R, $ < E, R < P, R < E] and true
                   in s |-> [(_, (_, T))].

typeOfExp(s, App(e1, e2)) = T :- {S U}
  typeOfExp(s, e1) == FUN(S, T),
  typeOfExp(s, e2) == U,
  subType(U, S).
```

Solver

Typed Program

```
type point = {x : num, y : num} in
let  mkpoint = fun(x : num) { {x = x, y = x} } in
type color = num in
type colorpoint =
       {k : color} extends point in
let  addColor =
       fun(c : num) {
         fun(p : colorpoint) {
           ({c = c} extends p) : colorpoint
         }
       } in
     (addColor 6 ({c = 5} extends mkpoint 4)) : colorpoint
```

Scope Graph



Case Studies: Setup

Statix Specification

```
typeOfExp : scope * Exp ->  Type

typeOfExp(s, Num(_)) = NUM().

typeOfExp(s, Plus(e1, e2)) = NUM() :-
  typeOfExp(s, e1) == NUM(),
  typeOfExp(s, e2) == NUM().

typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}
  typeOfTypeExp(s, te) == S,
  new s_fun, s_fun -P-> s,
  s_fun -> Var{x@x} with typeOfDecl S,
  typeOfExp(s_fun, e) == T.

typeOfExp(s, App(e1, e2)) = T :- {S U}
  typeOfExp(s, e1) == FUN(S, T),
  typeOfExp(s, e2) == U,
  subType(U, S).

typeOfExp(s, Rec(finits)) = REC(rs) :-
  new rs, fieldInitsOK(s, finits, rs).

typeOfExp(s, ERec(e1, e2)) = REC(rs) :- {rs1 rs2}
  typeOfExp(s, e1) == REC(rs1),
  typeOfExp(s, e2) == REC(rs2),
  new rs, rs -R-> rs1, rs -E-> rs2.

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

typeOfExp(s, FAccess(e, x)) = T :- {rs d}
  typeOfExp(s, e) == REC(rs),
  typeOfDecl of Fld{x@x} in rs |-> [(_, (_, T))].

typeOfExp(s, TypeLet(x, te, e)) = S :- {s_let}
  new s_let, s_let -P-> s,
  s_let -> Type{x@x} with typeOfDecl typeOfTypeExp(s, te),
  typeOfExp(s_let, e) == S.

typeOfExp(s, Let(x, e1, e2)) = S :- {s_let}
  new s_let, s_let -P-> s,
  s_let -> Var{x@x} with typeOfDecl typeOfExp(s, e1),
  typeOfExp(s_let, e2) == S.
```

Typing and Name Resolution Tests

```
test record literal [[
  {x = 1, y = 2, h = {}}
]] analysis succeeds

test record extension [[
  {p = 5} extends {x = 1, y = 2, h = {}}
]] analysis succeeds

test record projection [[
  {x = 1, y = 2, h = {}}.x
]] analysis succeeds

test record mis projection [[
  {x = 1, y = 2, h = {}}.z
]] analysis fails

test record projection [[
  type point = {x : num, y : num} in
    fun(p : point) { p.x + p.y }
]] analysis succeeds

test record application [[
  type point = {x : num, y : num} in
    (fun(p : point) { p.x + p.y } {x = 1, y = 2}) : num
]] analysis succeeds

test record subtype [[
  type point = {x : num, y : num} in
    (fun(p : point) { p.x + p.y } {x = 1, y = 2, z = 3}) : num
]] analysis succeeds

test record not a subtype [[
  type point = {x : num, y : num} in
    (fun(p : point) { p.x + p.y } {x = 1, z = 3}) : num
]] analysis fails
```

Test Results

SPT Test Runner

Tests 26 / 26

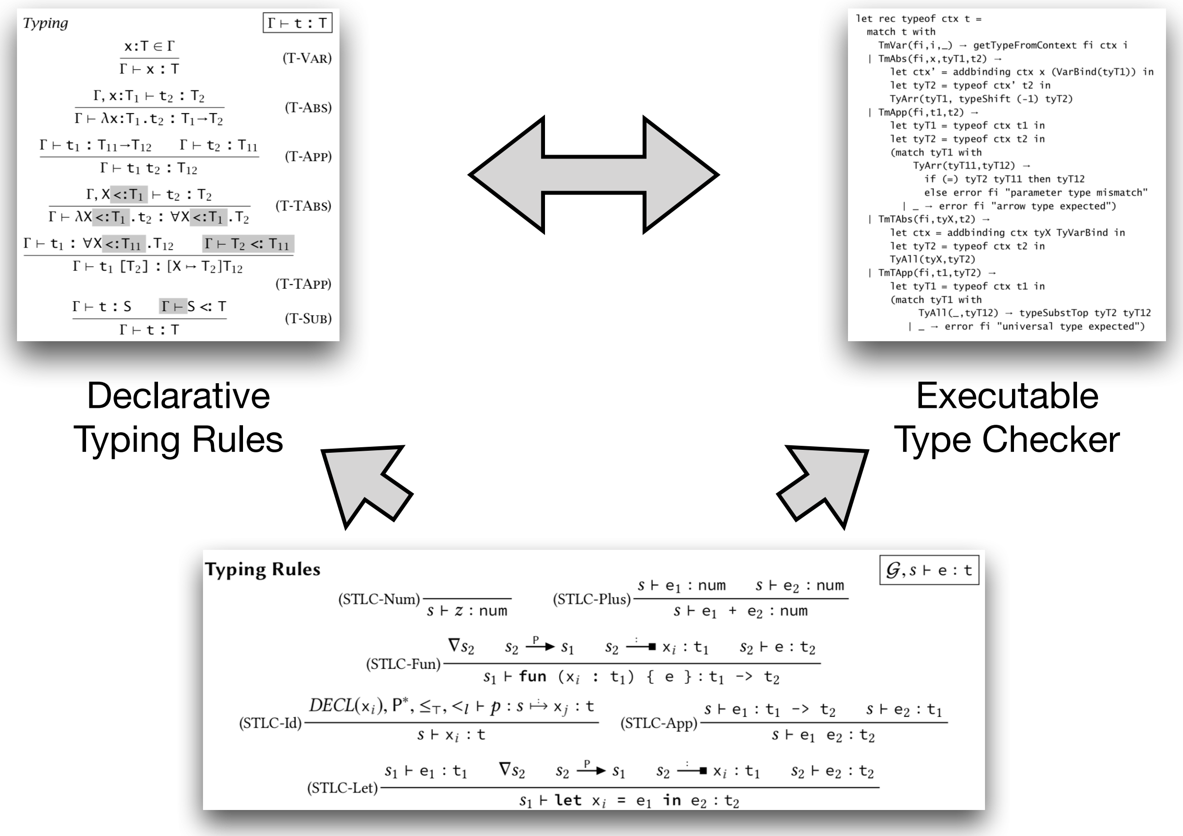
- records
  - duplicate field name 1 (0.06s)
  - duplicate field name 2 (0.07s)
  - duplicate record field extension 1 (0.06s)
  - duplicate record field extension 2 (0.10s)
  - duplicate record field fun type (0.10s)
  - let within with (0.05s)
  - nested record not a subtype (0.14s)
  - nested records (0.20s)
  - record application (0.19s)
  - record extension (0.15s)
  - record extension (1.65s)
  - record extension 1 (0.51s)
  - record extension 2 (0.56s)
  - record extension 3 (0.61s)
  - record extension 4 (0.56s)
  - record extension subtype (0.10s)
  - record literal (0.10s)
  - record mis projection (0.17s)
  - record not a subtype (0.16s)
  - record projection (0.13s)
  - record projection (0.15s)
  - record subtype (0.26s)
  - type ascription subtype (0.07s)
  - with extended record (0.08s)
  - with simple record (0.04s)
  - with within let (0.05s)

Case Studies: Languages

STLC with Structural Records	System F	Featherweight Generic Java
Structural types Structural subtyping	Parametric polymorphism	Nominal types Class inheritance Nominal subtyping Generics

Declarative, Executable Type System Specifications

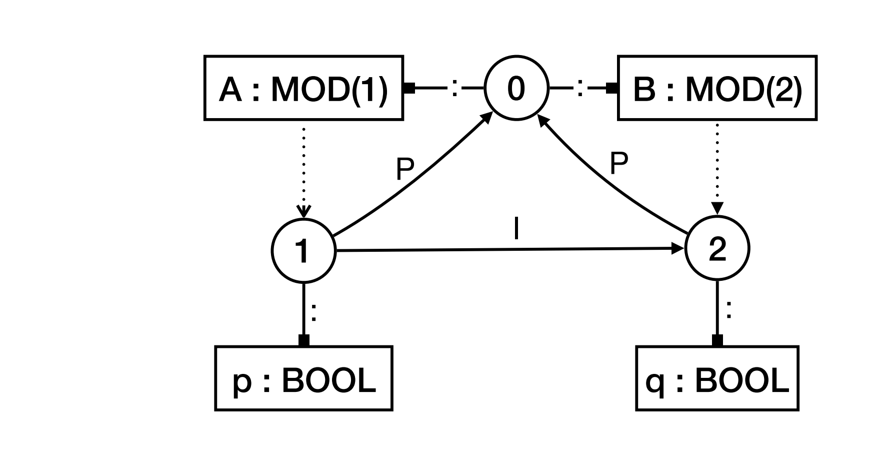
Type System Specifications



x

Statix: Declarative Rules

```
module A {
  import B
  def p : bool = ~q
}
module B {
 def q : bool = true
}
```



scope graph assertions

$$\frac{\nabla s_m \quad s \xrightarrow{:} x_i : \mathrm{MOD}(s_m) \quad s_m \xrightarrow{P} s \quad s_m \vdash \overline{\mathtt{stm}}\ \mathrm{OK}}{s \vdash \mathtt{module}\ \mathtt{x}_i\ \{\ \overline{\mathtt{stm}}\ \}\ \mathrm{OK}}$$

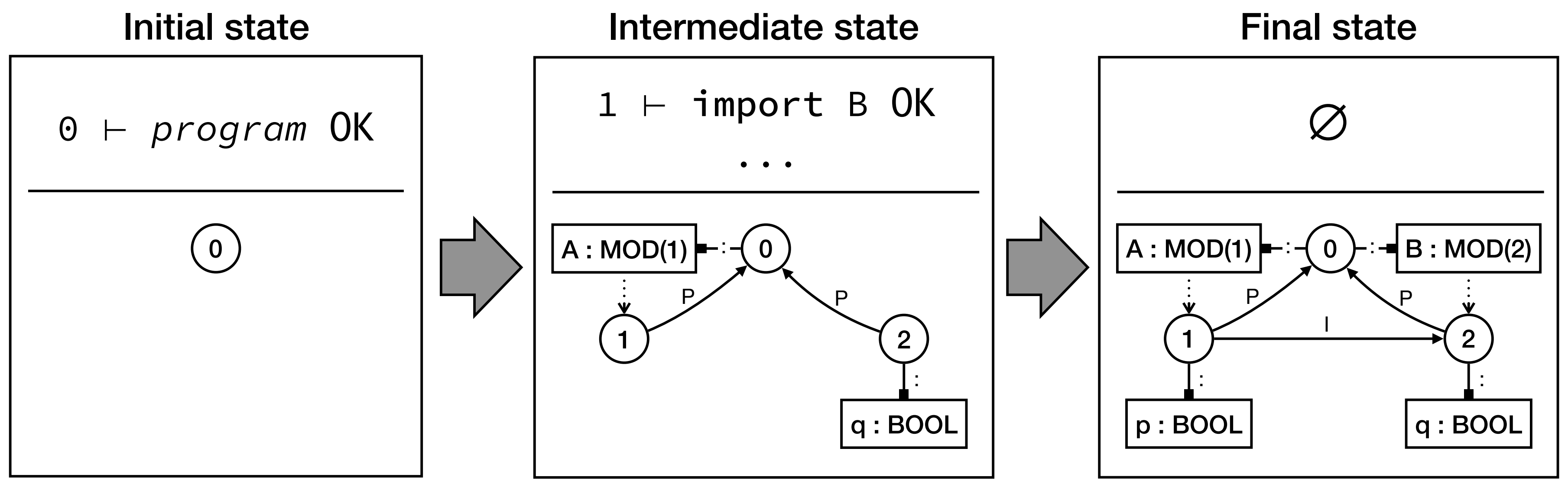
syntax-directed rules

scope graph queries

$$\frac{\mathrm{DECL}(x_i), \mathrm{P}^* \vdash p : s \mapsto x_j : \mathrm{MOD}(s_m) \quad s \xrightarrow{I} s_m}{s \vdash \mathtt{import}\ \mathtt{x}_i\ \mathrm{OK}}$$

x

Execution by Rewriting



- Non-deterministic constraint solving
- Rule-based simplification
- Ensure correct scope graph queries

x

Case Studies: Languages

STLC with Structural Records	System F	Featherweight Generic Java
Structural types Structural subtyping	Parametric polymorphism	Nominal types Class inheritance Nominal subtyping Generics

x

(slide intentionally left blank)

Comparison to previous work [Van Antwerpen, 2016]

Scope graph

- Holds any data, not just declarations
- Powerful queries
    - Subsumes previous name resolution relation

Constraint language

- Allow user-defined relations over types
    - Was restricted to AST-to-constraint mappings
    - Now support parametric polymorphism, structural type comparisons

* H. van Antwerpen, P. Néron, A. Tolmach, E. Visser, and G. Wachsmuth. 2016. *A constraint language for static semantic analysis based on scope graphs.* In PEPM '16. ACM, New York, NY, USA, 49-60.

Contributions

Scope Graphs	Statix	Evaluation
Generalization of previous work* Previous limitations: • Structural types • Parametric polymorphism/ generics Improvements: • Generalize declarations to arbitrary data • Generalize resolution to queries	A new constraint language to write declarative type checkers Scope graph assertions and queries are built-in concepts A formal declarative semantics for Statix An constraint solving algorithm to execute specifications as type checkers	An implementation of Statix Statix specification for: • STLC+REC • System F • FGJ Test suites for evaluation languages

* H. van Antwerpen, P. Néron, A. Tolmach, E. Visser, and G. Wachsmuth. 2016. *A constraint language for static semantic analysis based on scope graphs.* In PEPM '16. ACM, New York, NY, USA, 49-60.