

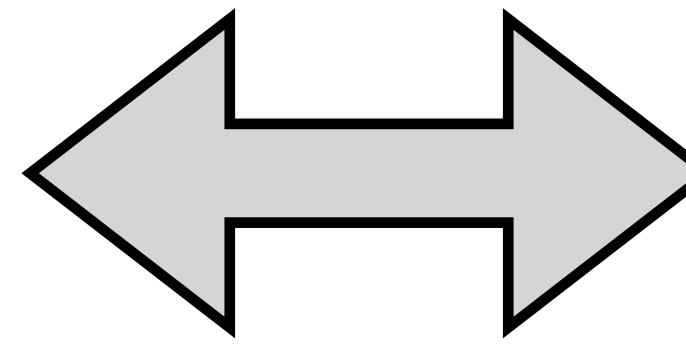
Scopes as Types

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

OOPSLA'18, Boston, MA, USA

Type System Specifications

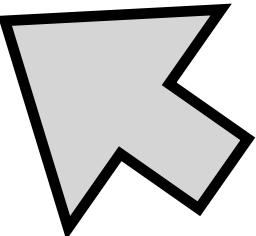
<i>Typing</i>	$\boxed{\Gamma \vdash t : T}$
$\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)
$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2}$	(T-ABS)
$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(T-APP)
$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1. t_2 : \forall X <: T_1. T_2}$	(T-TABS)
$\frac{\Gamma \vdash t_1 : \forall X <: T_{11}. T_{12} \quad \Gamma \vdash t_2 <: T_{11}}{\Gamma \vdash t_1 [t_2] : [X \mapsto T_2] T_{12}}$	(T-TAPP)
$\frac{\Gamma \vdash t : S \quad \Gamma \vdash S <: T}{\Gamma \vdash t : T}$	(T-SUB)



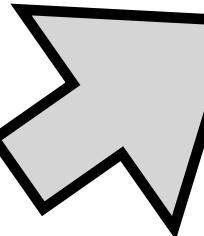
```

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



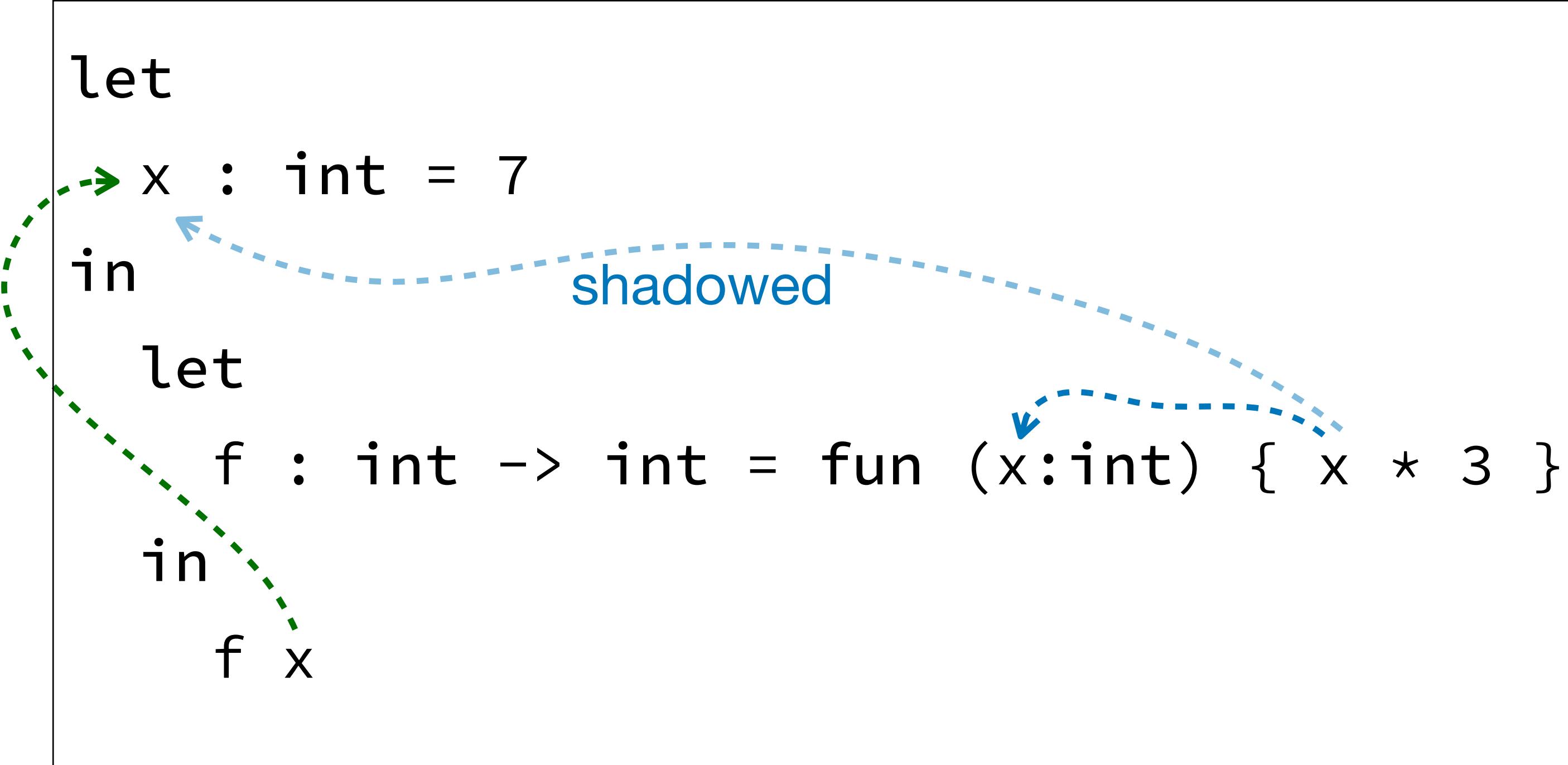
Typing Rules	$\boxed{\mathcal{G}, s \vdash e : t}$
(STLC-Num) $\frac{}{s \vdash z : \text{num}}$	(STLC-Plus) $\frac{s \vdash e_1 : \text{num} \quad s \vdash e_2 : \text{num}}{s \vdash e_1 + e_2 : \text{num}}$
(STLC-Fun) $\frac{\nabla s_2 \quad s_2 \xrightarrow{P} s_1 \quad s_2 \overset{:}{\rightarrow} x_i : t_1 \quad s_2 \vdash e : t_2}{s_1 \vdash \text{fun } (x_i : t_1) \{ e \} : t_1 \rightarrow t_2}$	
(STLC-Id) $\frac{DECL(x_i), P^*, \leq_T, <_I \vdash p : s \overset{:}{\rightarrow} x_j : t}{s \vdash x_i : t}$	(STLC-App) $\frac{s \vdash e_1 : t_1 \rightarrow t_2 \quad s \vdash e_2 : t_1}{s \vdash e_1 e_2 : t_2}$
(STLC-Let) $\frac{s_1 \vdash e_1 : t_1 \quad \nabla s_2 \quad s_2 \xrightarrow{P} s_1 \quad s_2 \overset{:}{\rightarrow} x_i : t_1 \quad s_2 \vdash e_2 : t_2}{s_1 \vdash \text{let } x_i = e_1 \text{ in } e_2 : t_2}$	

Declarative Specification and Name Resolution

Uniform Binding Representation

Abstraction over Execution Order

Binding: Lexical



Representation

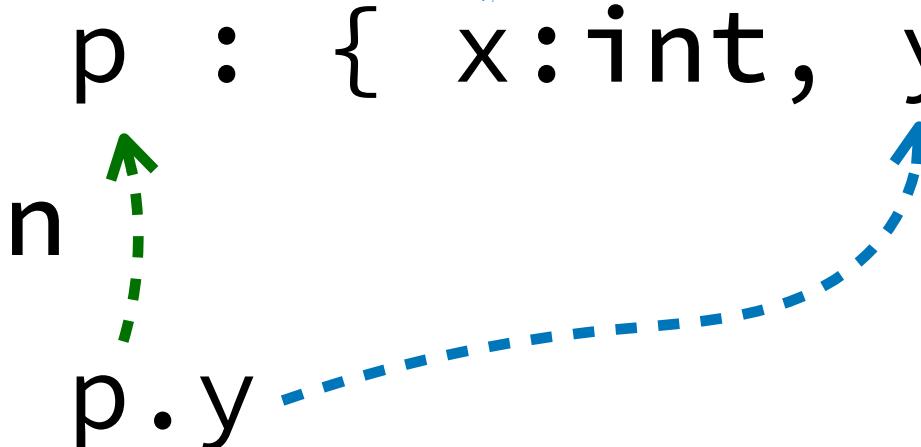
- Typing environment
- Ordered list of name-types

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

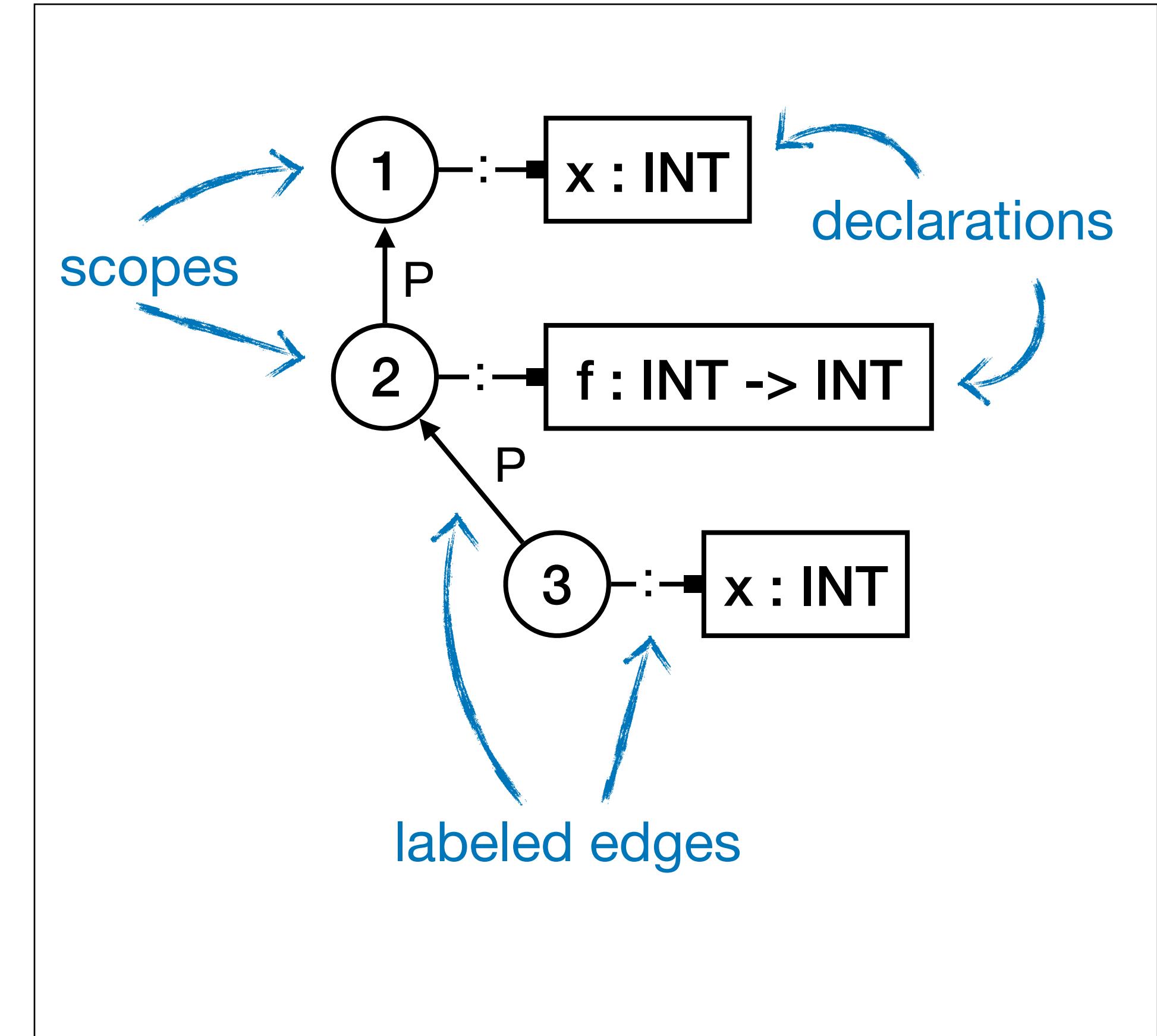
Our Approach

Scope Graphs

Statix

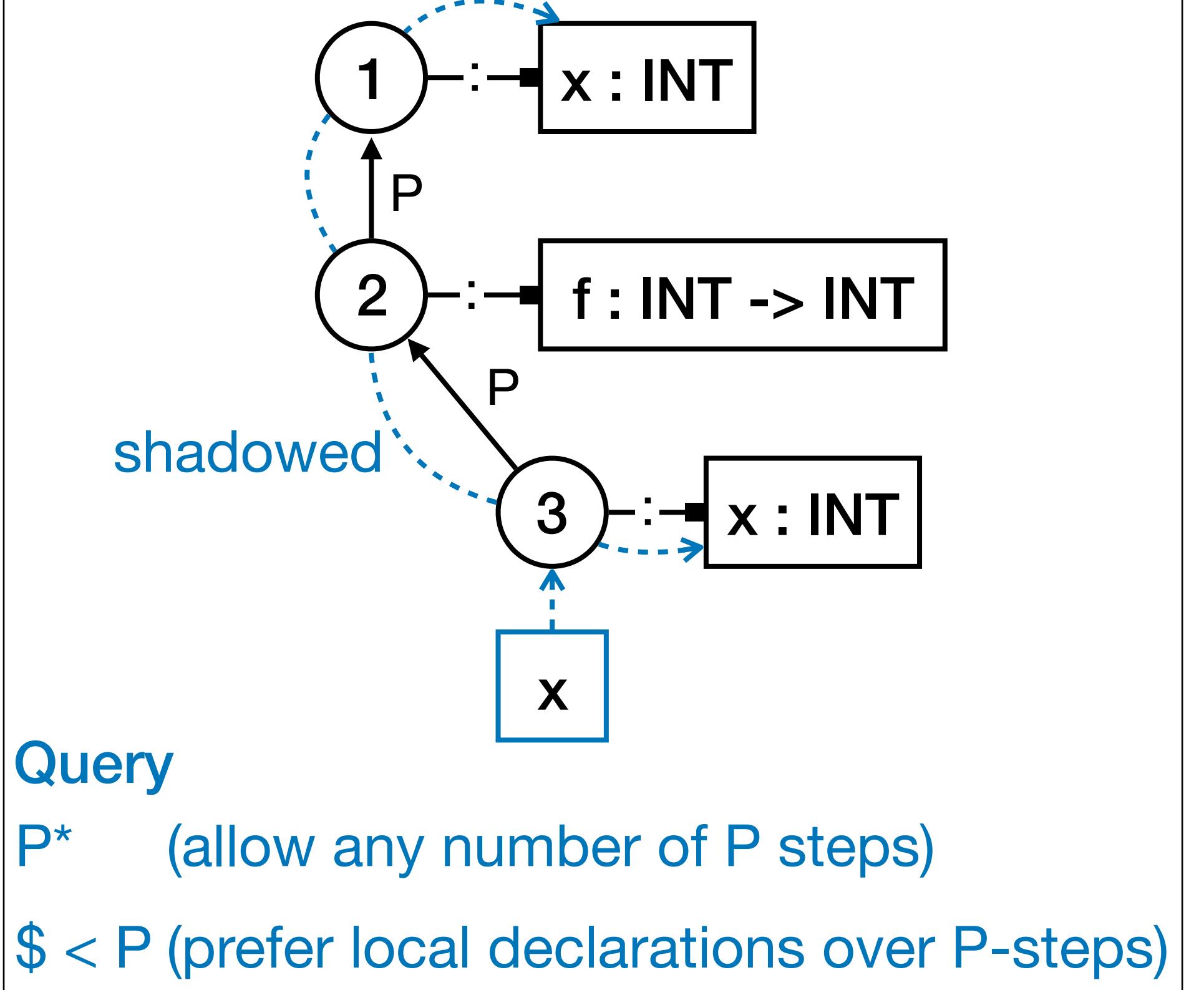
Scope Graph: Lexical

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



Scope Graph: Lexical

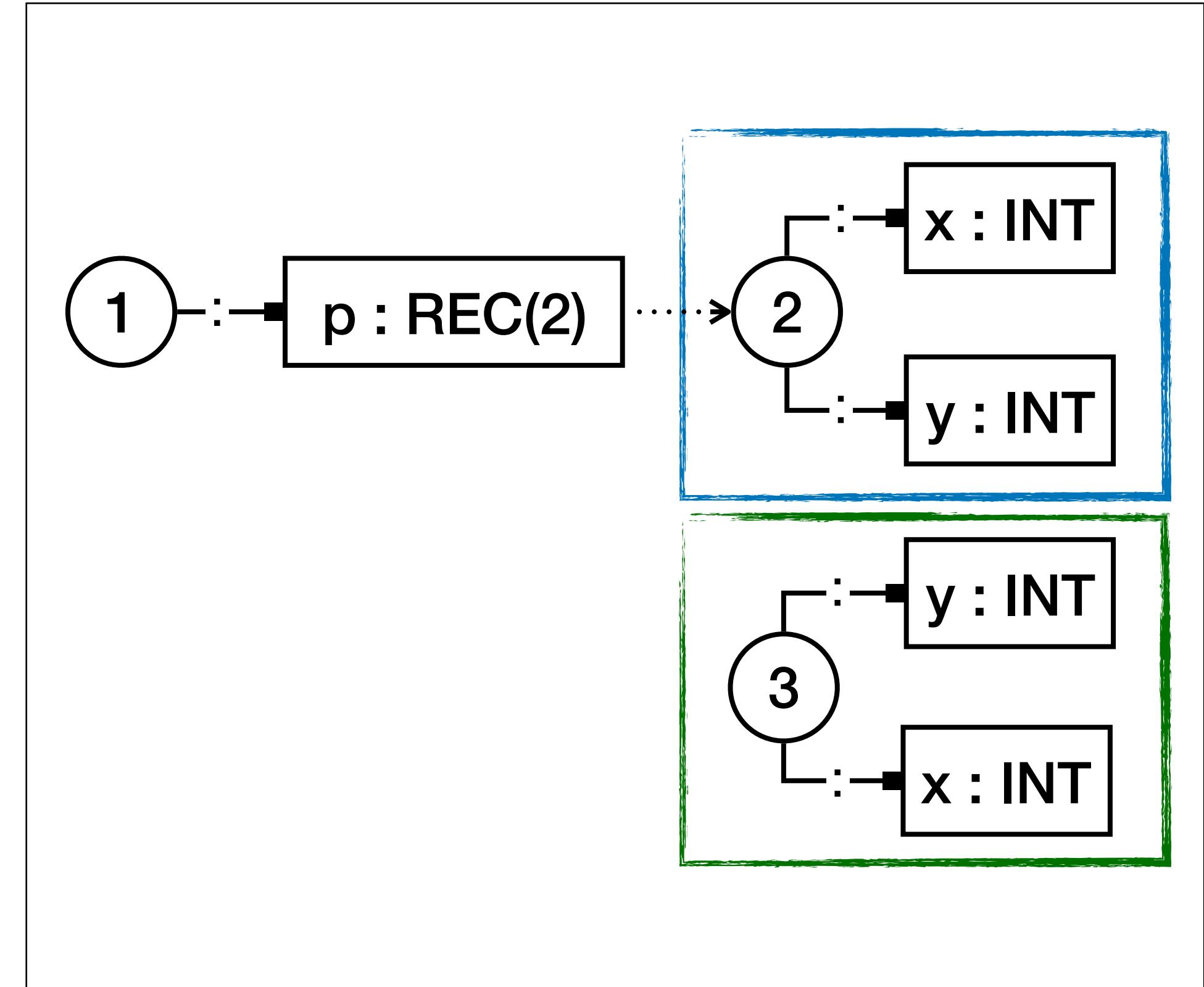
```
let
  x : int = 7
in
let
  f : int -> int = fun (x:int) { x * 3 }
in
  f x
```



- 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
```

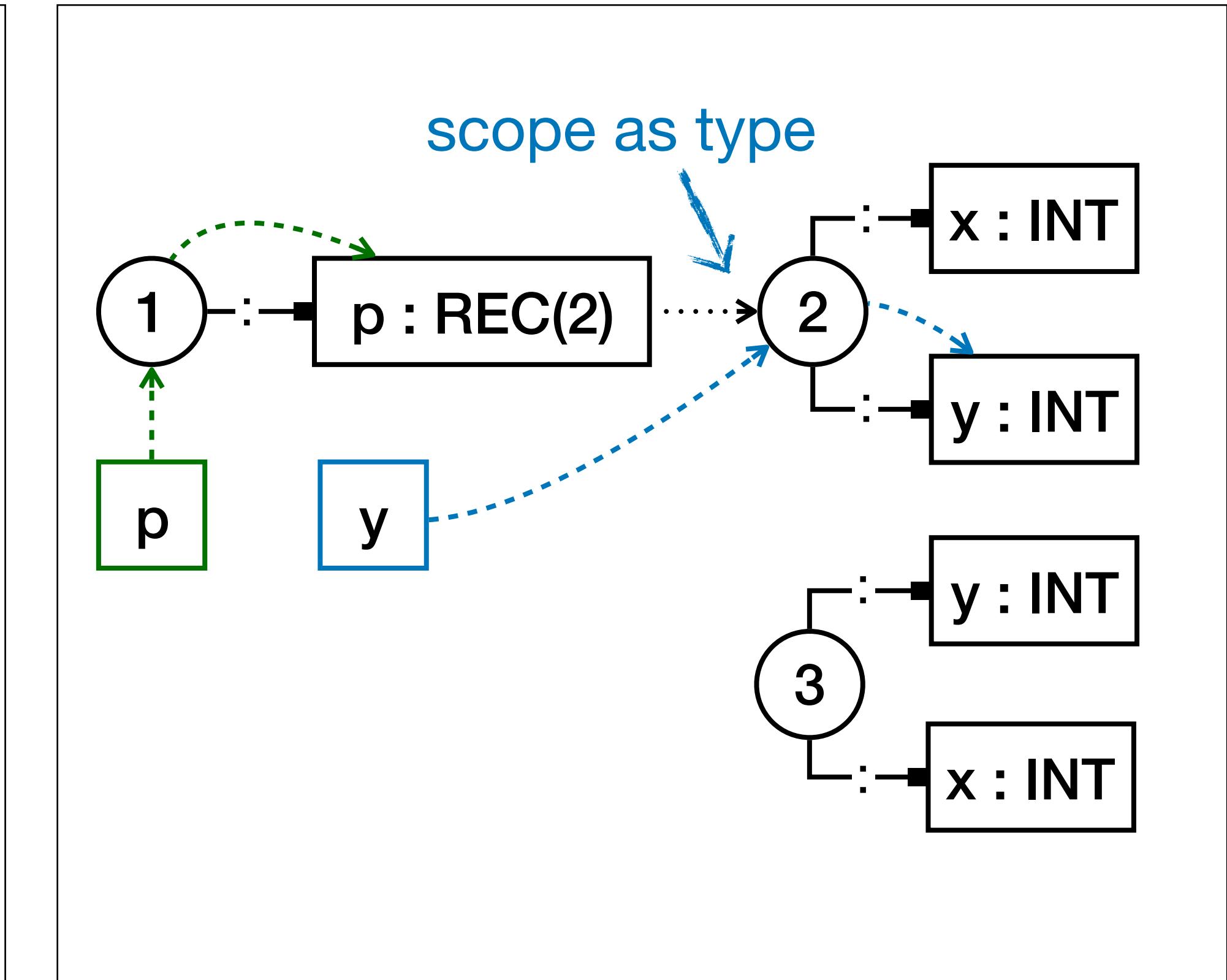


- Type structure described by scopes

Scope Graph: Records

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

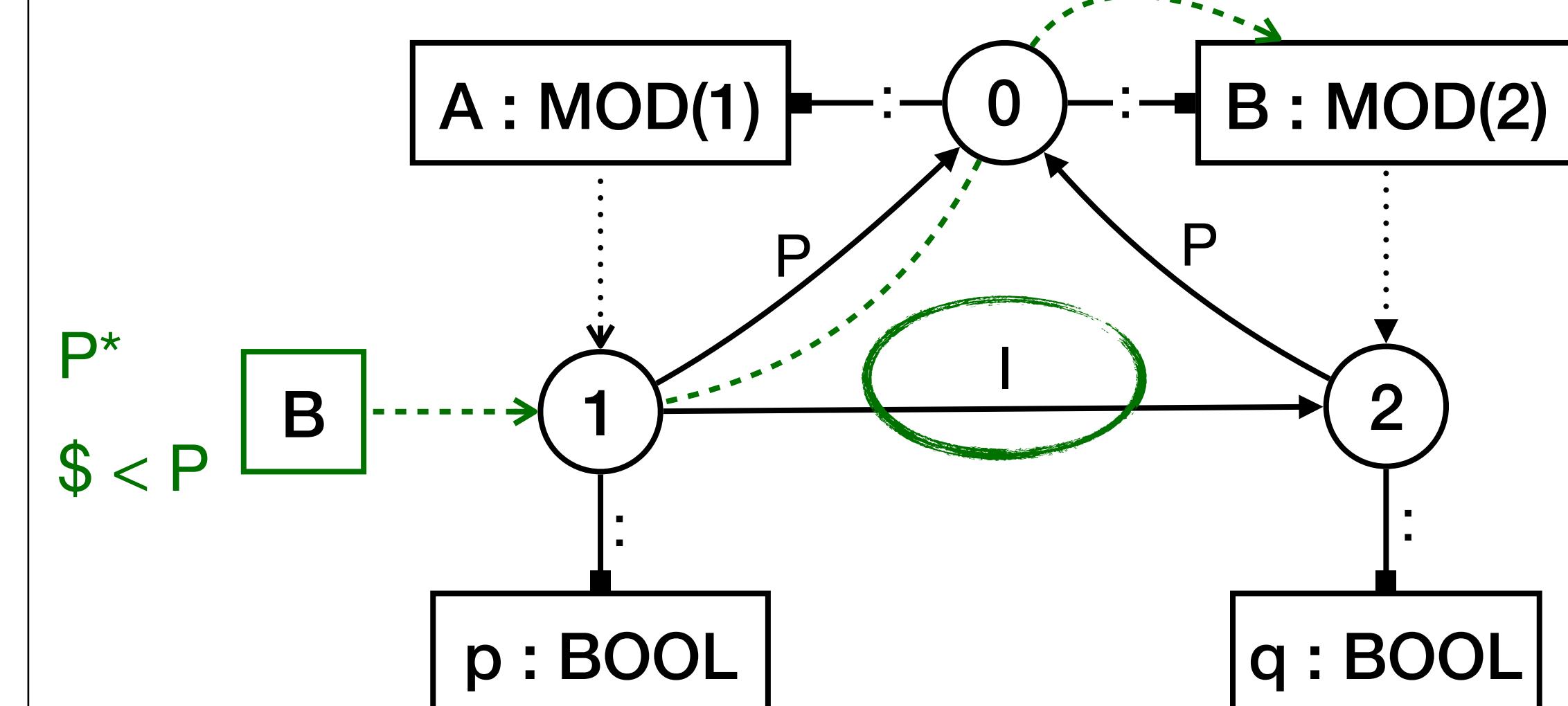
type-dependent name resolution



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

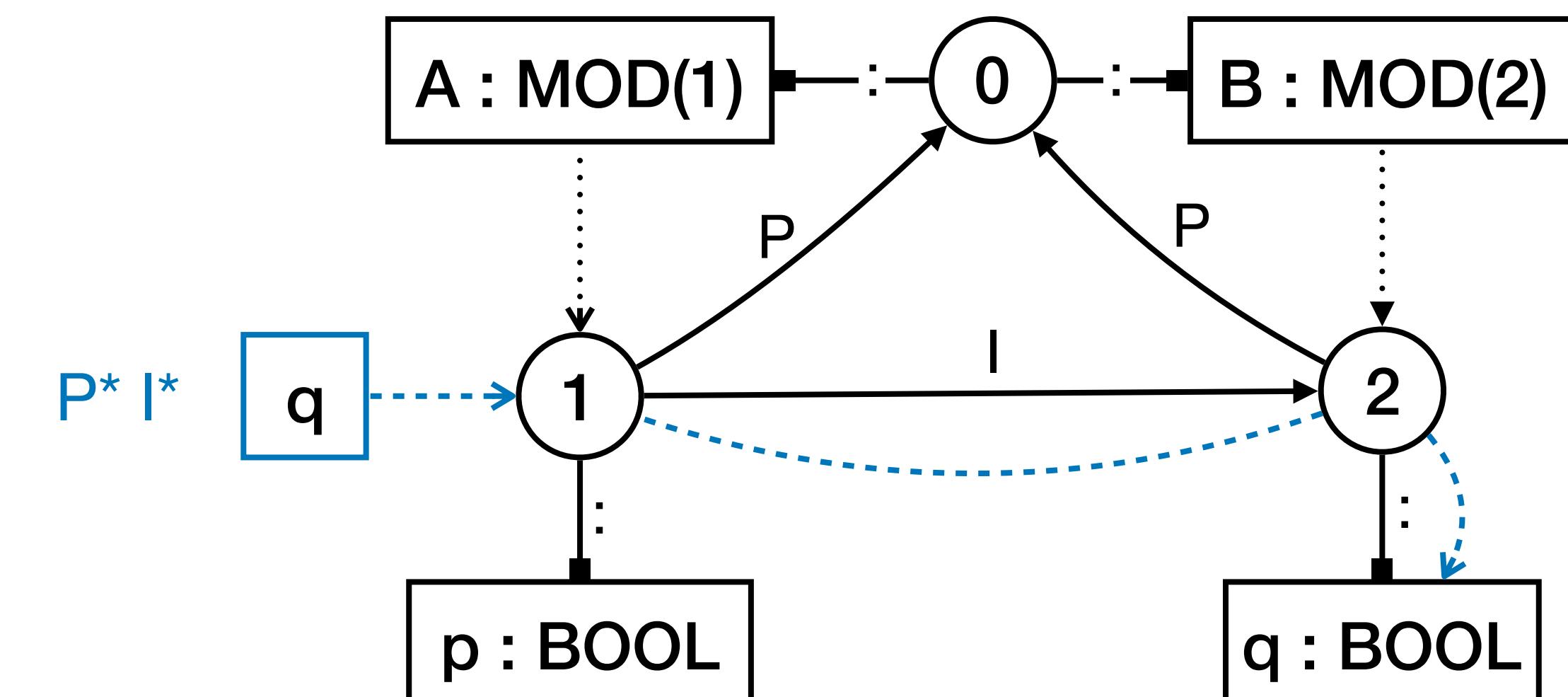
Scope Graph: Modules

```
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

Language independent

Capture different kinds of binding

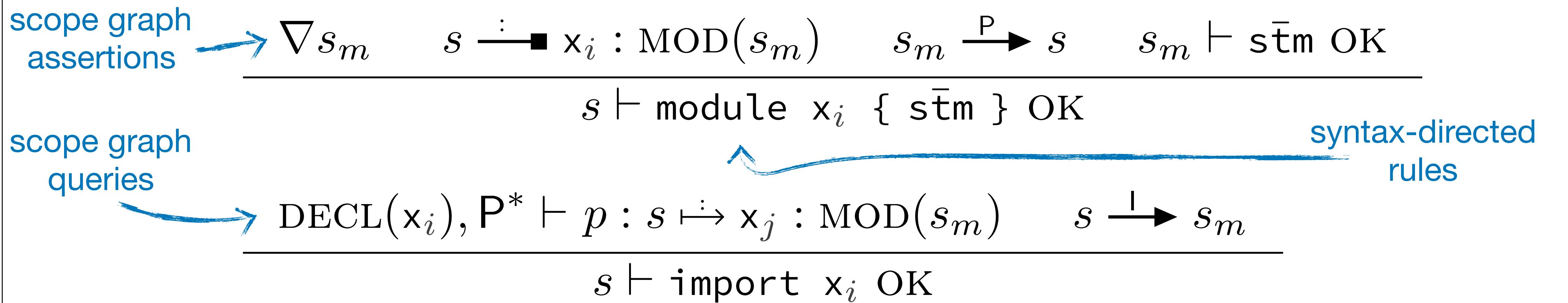
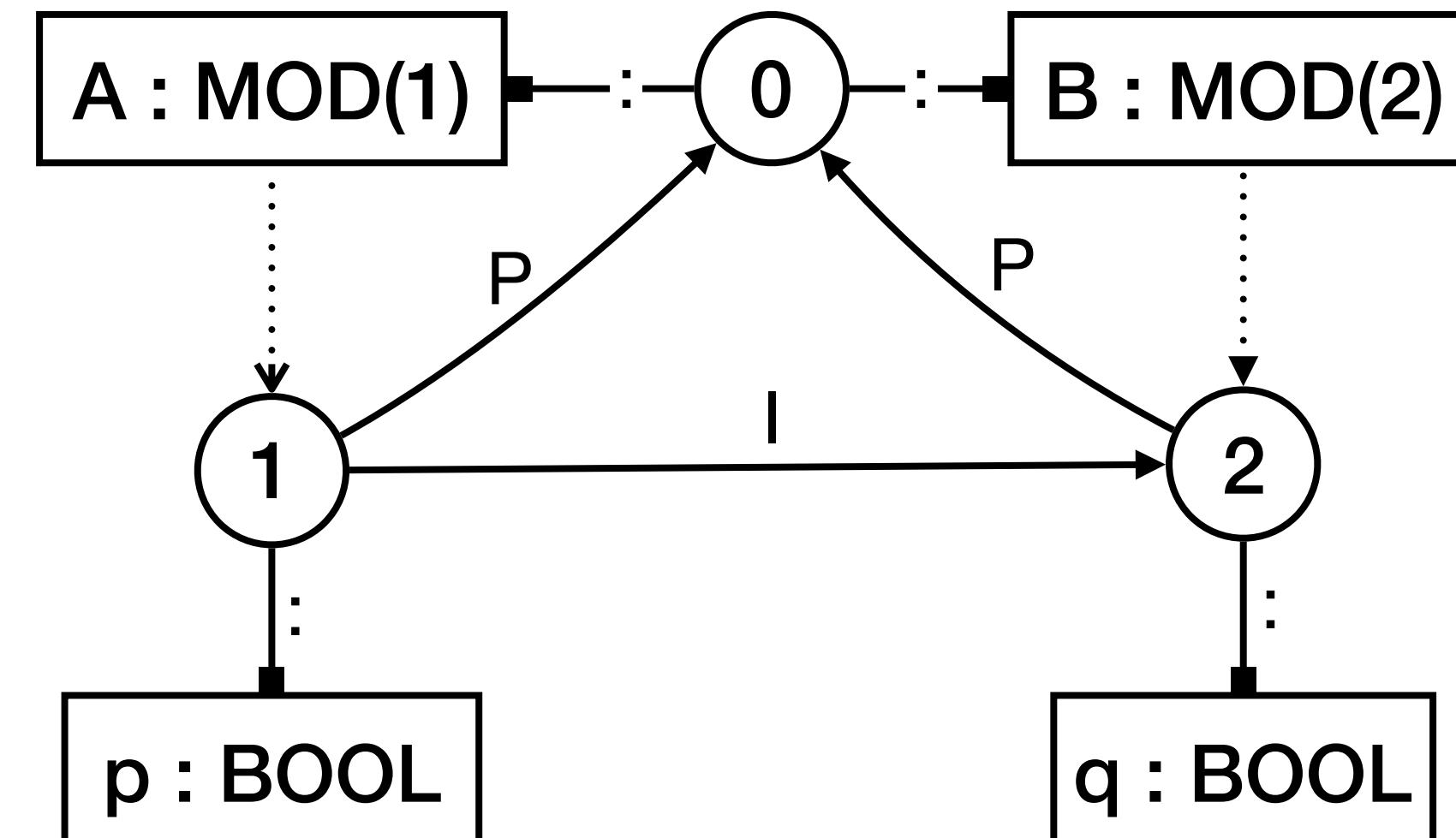
Resolve names by graph queries

Statix

Statix: Declarative Rules

```

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

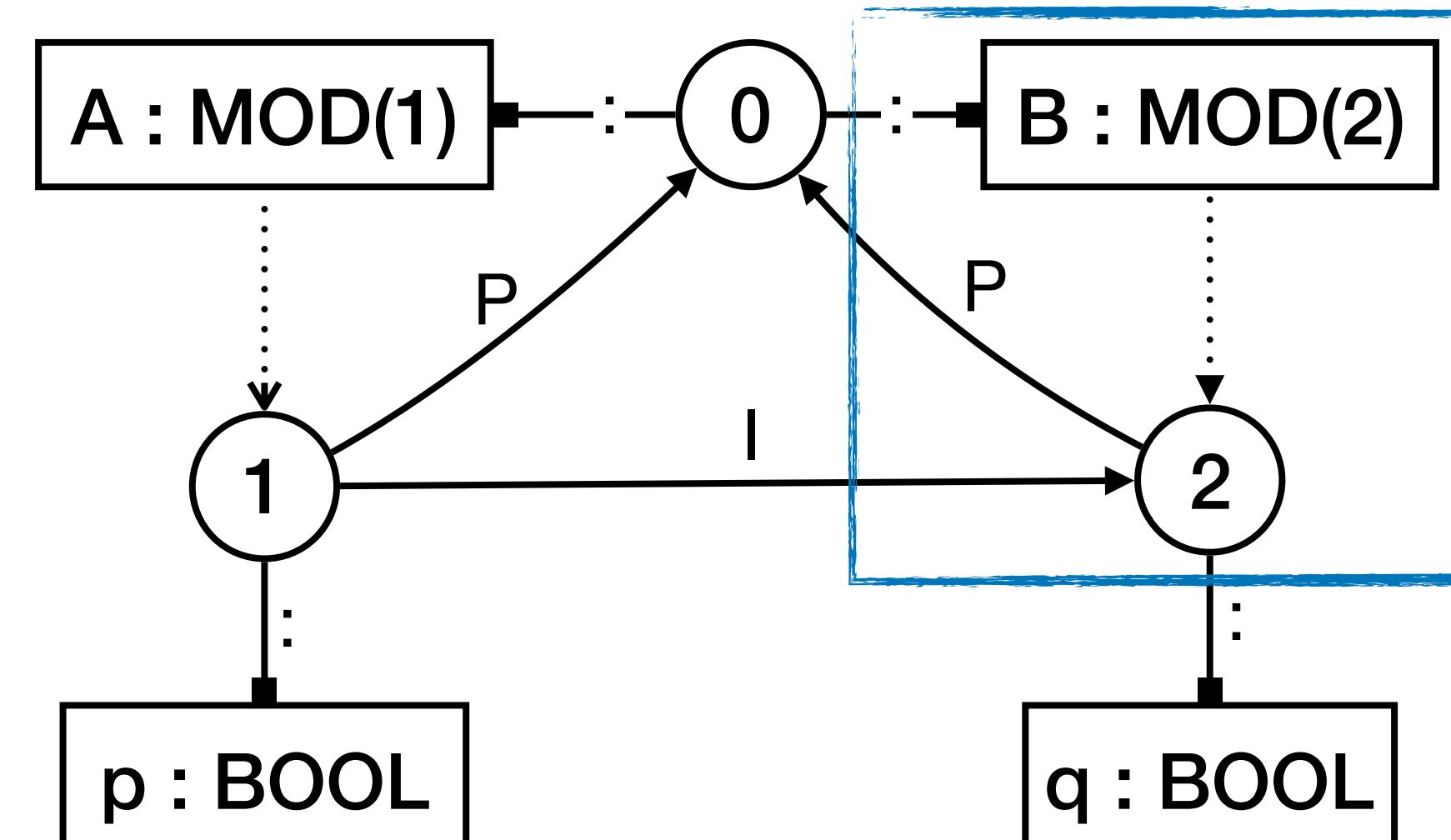


Statix: Declarative Rules

```

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

```



$$\frac{\nabla s_m \quad s \overset{\cdot}{\dashrightarrow} x_i : \text{MOD}(s_m) \quad s_m \xrightarrow{P} s \quad s_m \vdash \text{stmt OK}}{s \vdash \text{module } x_i \{ \text{stmt} \} \text{ OK}}$$

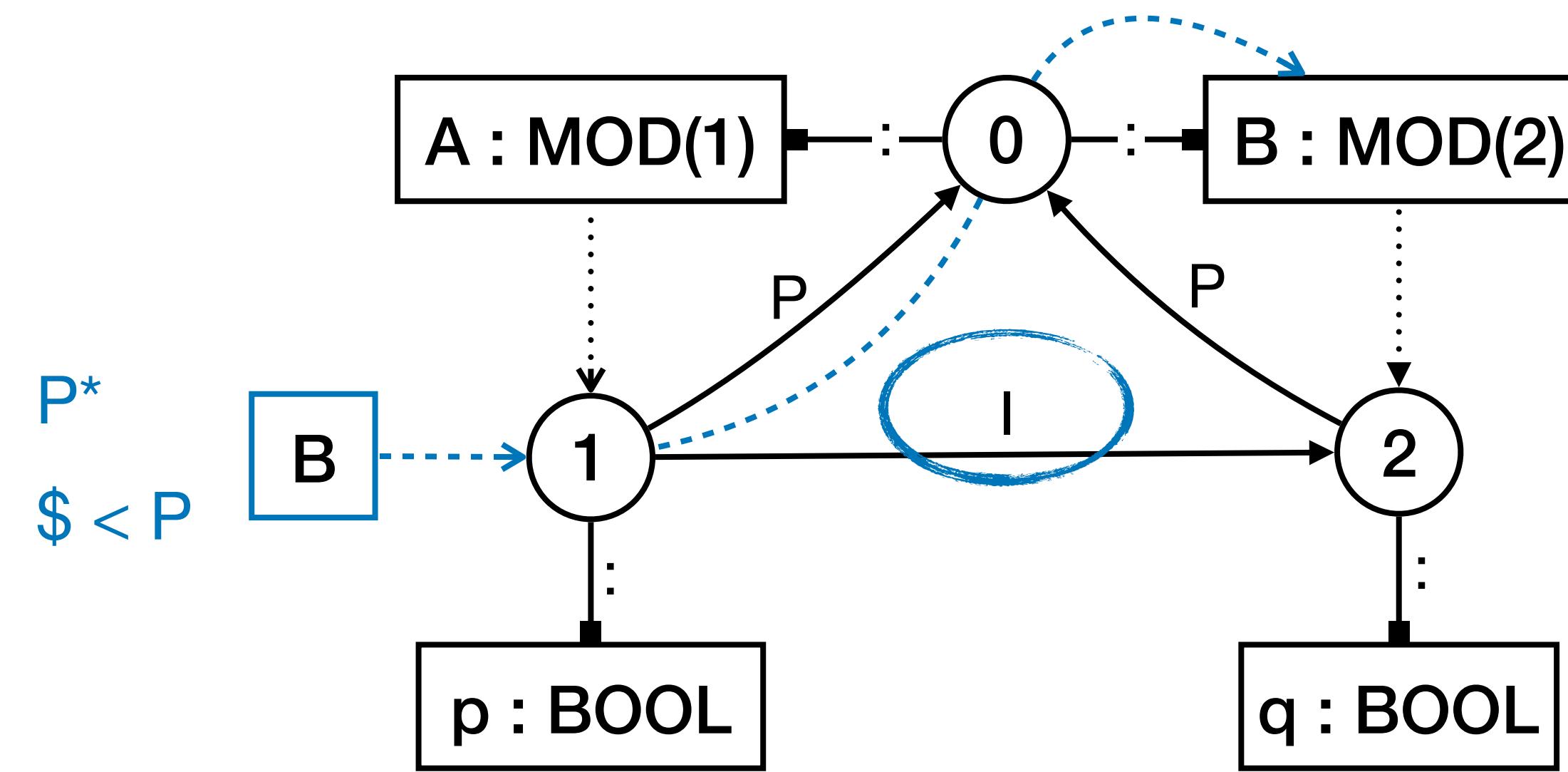
$$\frac{\text{DECL}(x_i), P^* \vdash p : s \mapsto x_j : \text{MOD}(s_m) \quad s \xrightarrow{I} s_m}{s \vdash \text{import } x_i \text{ OK}}$$

Statix: Declarative Rules

```

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

```



$$\frac{\nabla s_m \quad s \xrightarrow{\cdot} x_i : \text{MOD}(s_m) \quad s_m \xrightarrow{P} s \quad s_m \vdash \text{stmt OK}}{s \vdash \text{module } x_i \{ \text{stmt} \} \text{ OK}}$$

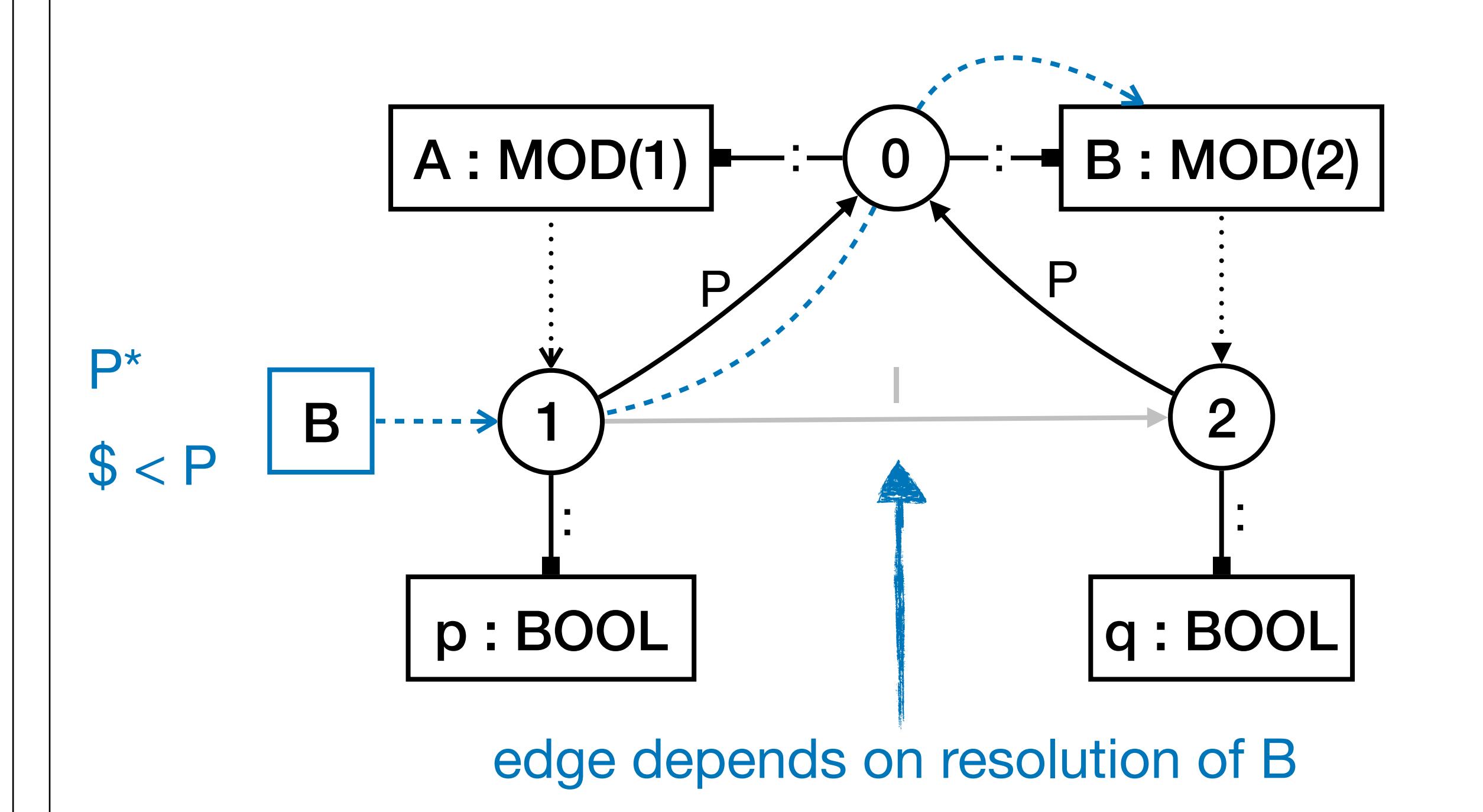
$\text{DECL}(x_i), P^* \vdash p : s \xrightarrow{\cdot} x_j : \text{MOD}(s_m)$

$s \xrightarrow{\perp} s_m$

$s \vdash \text{import } x_i \text{ OK}$

Interleave Graph Construction and Queries

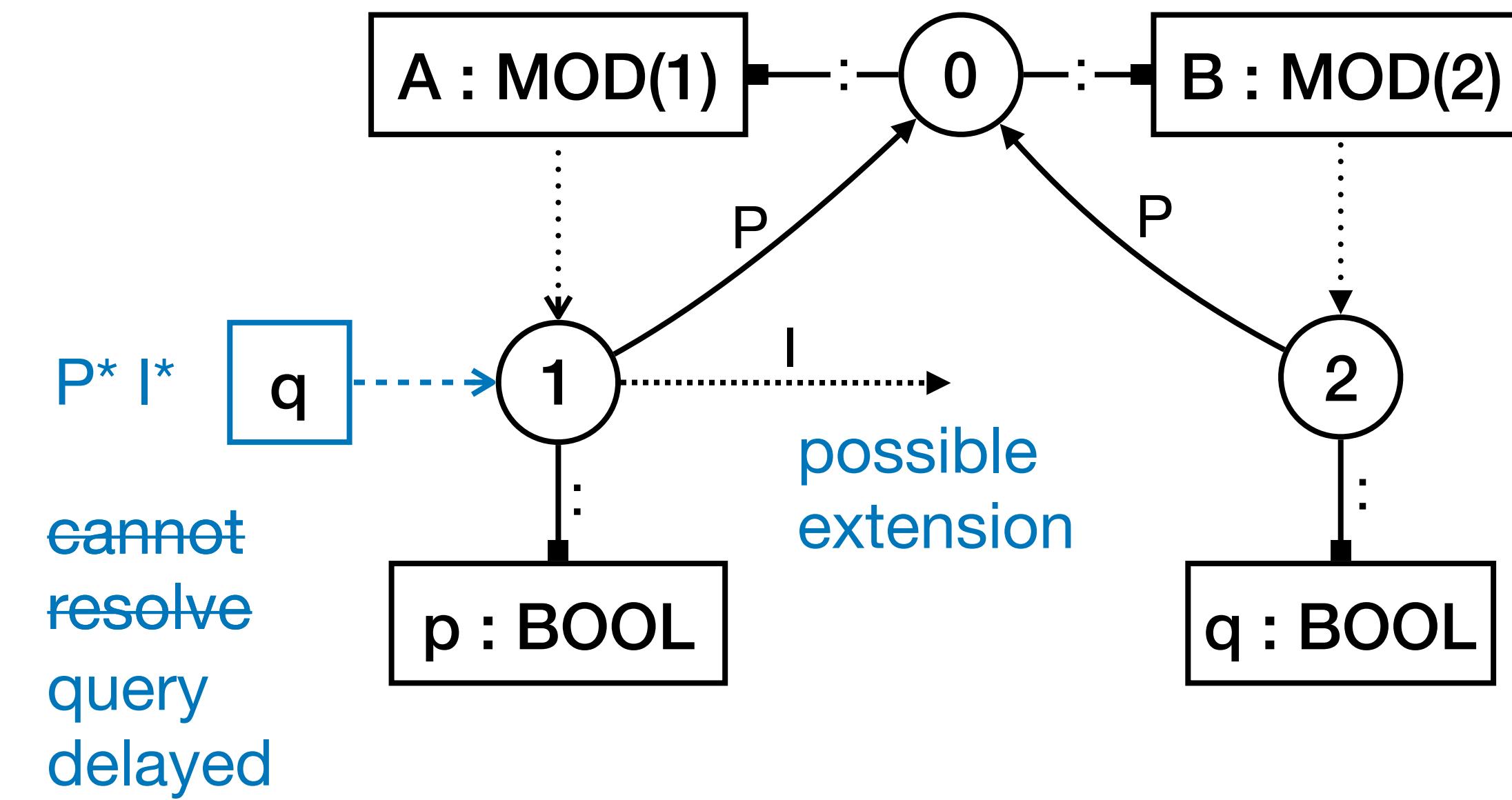
```
module A {  
    import B  
  
    def p : bool = ~q  
}  
  
module B {  
    def q : bool = true  
}
```



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

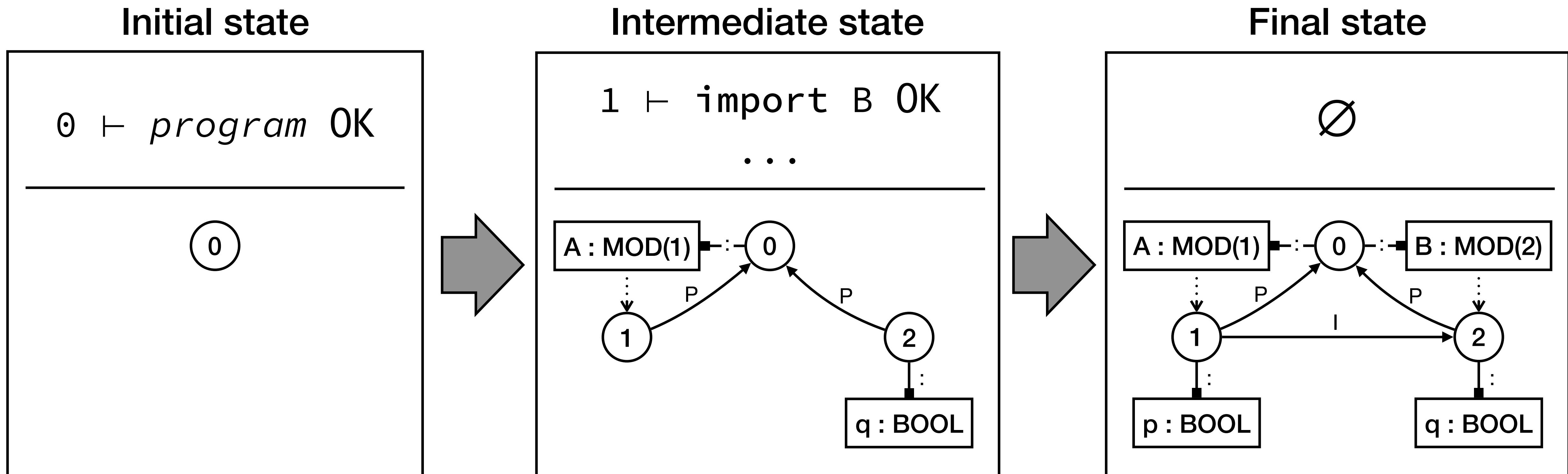
Queries in Incomplete Graphs

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



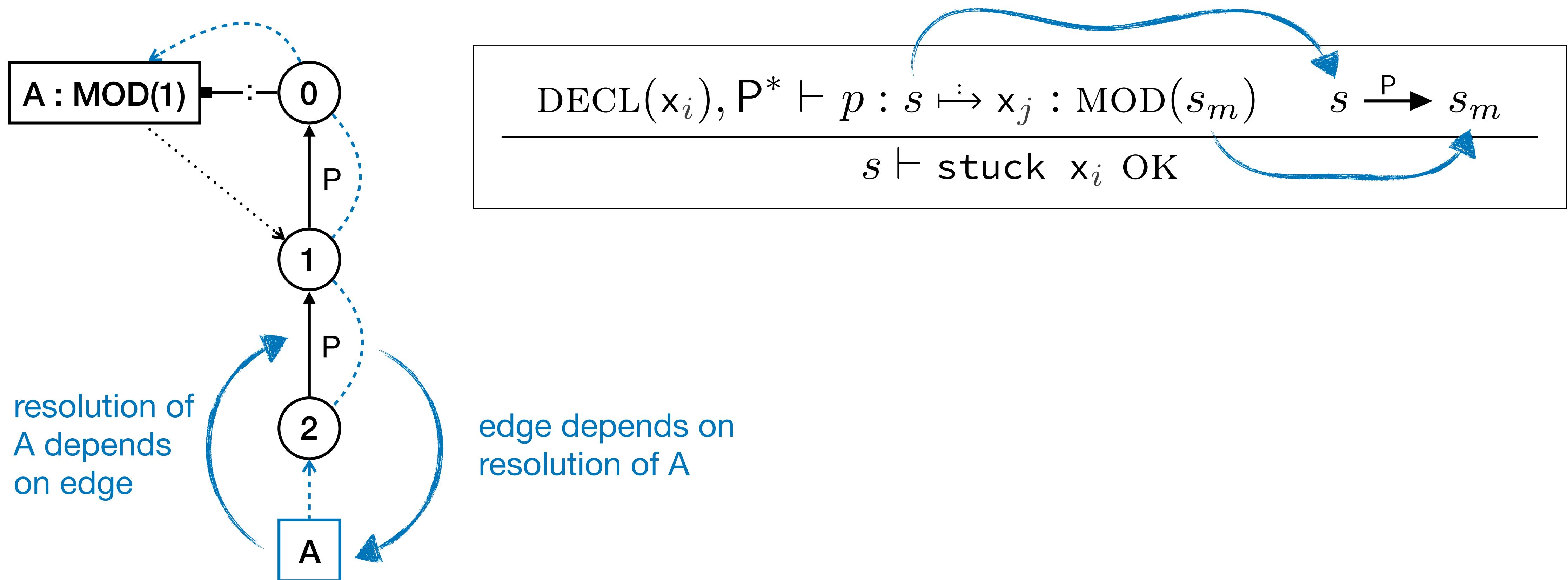
- Query attempt in incomplete graph
- Approximate possible edge extensions
- Delay if extension may change result

Execution by Rewriting



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

Unorderable Constraints get Stuck



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

```

package mb.statix.solver;

import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collectors;

import javax.annotation.Nullable;

import org.immutables.value.Value;
import org.metaborg.util.functions.Predicated;
import com.google.common.collect.ImmutableMultimap;
import com.google.common.collect.ImmutableSet;
import com.google.common.collect.Maps;
import com.google.common.collect.Sets;

import mb.nabl2.terms.ITerm;
import mb.nabl2.terms.ITermVar;
import mb.nabl2.terms.Unification.IUnifier;
import mb.nabl2.util.TermFormatter;
import mb.statix.solver.log.IDebugContext;
import mb.statix.solver.log.LazyDebugContext;
import mb.statix.solver.log.Log;

public class Solver {

    private Solver() {
    }

    public static class SolverResult {
        final State _state;
        final Iterable<IConstraint> _constraints;
        final Completeness _completeness;
        final IDebugContext _debug;
        final Delay _delayed;
        final LazyDebugContext _proxyDebug;
        final Log _log;

        public SolverResult(State state, Iterable<IConstraint> constraints, Completeness completeness, IDebugContext debug, Delay delayed) {
            _state = state;
            _constraints = constraints;
            _completeness = completeness;
            _debug = debug;
            _delayed = delayed;
            _proxyDebug = proxyDebug(debug);
            _log = log(debug);
        }
    }

    private void solveInternal(State state, Iterable<IConstraint> constraints, Completeness completeness, IDebugContext debug) throws InterruptedException {
        return solve(state, constraints, completeness, v -> false, z -> false, debug);
    }

    public static SolverResult solve(State state, Iterable<IConstraint> constraints, Completeness completeness, IDebugContext debug) throws InterruptedException {
        return solveInternal(state, constraints, completeness, debug);
    }

    // set-up
    final Set<IConstraint> constraints = Sets.newConcurrentHashSet(_constraints);
    State state = _state;
    Completeness completeness = _completeness;
    completeness.add(completeness.addAll(constraints));

    // fixed point
    final Set<IConstraint> failed = Sets.newHashSet();
    final Log delayedLog = new Log();
    final Map<IConstraint, Delay> delays = Maps.newHashMap();
    boolean progress = true;
    int reduced = 0;
    int delayed = 0;
    outer: while(progress) {
        progress = false;
        Iterator<IConstraint> iterator = constraints.iterator();
        delays.clear();
        final Iterator<IConstraint> it = constraints.iterator();
        while(it.hasNext()) {
            if(Thread.interrupted()) {
                throw new InterruptedException();
            }
            final IConstraint constraint = it.next();
            proxyDebug.info("Solving '{}', constraint:{}", Solver.shallowTermFormatter(state.unifier()));
            delayedContext subDebug = proxyDebug.subContext();
            try {
                Optional<ConstraintResult> maybeResult =
                    Optional.ofNullable(resultFor(constraint));
                state = solve(state, completeness, new ConstraintContext(completeness, isRigid, isClosed, subDebug));
                progress = true;
                it.remove();
                completeness = completeness.remove(constraint);
                reduced += 1;
                if(maybeResult.isPresent()) {
                    final ConstraintResult result = maybeResult.get();
                    state = result.state();
                    if(result.constraints().isEmpty()) {
                        final List<IConstraint> newConstraints = result.constraints().stream()
                            .map(c -> c.withCause(constraint).collect(Collectors.toList()));
                        subDebug.info("Simplified to '{}', toString(newConstraints, state.unifier())");
                        constraints.addAll(newConstraints);
                        completeness = completeness.addAll(newConstraints);
                    }
                } else {
                    subDebug.error("Failed!");
                    failed.add(constraint);
                    if(proxyDebug.isRoot()) {
                        printTrace(constraint, state.unifier(), subDebug);
                    } else {
                        proxyDebug.info("Break early because of errors.");
                        break outer;
                    }
                }
            } catch(Delay d) {
                subDebug.info("Delayed");
                delayedLog.absorb(proxyDebug.clear());
                delays.put(constraint, d);
                delayed += 1;
            }
        }
    }

    delayedLog.flush(debug);
    debug.info("Solved {} constraints ({} failed) with {} delayed and {} remaining constraint(s).", reduced, delayed,
        failed.size(), constraints.size());
    return SolverResult.of(state, completeness, failed, delays);
}

public static Optional<SolverResult> entails(State state, Iterable<IConstraint> constraints, final Completeness completeness, final IDebugContext debug) throws InterruptedException, Delay {
    return entails(state, constraints, completeness, ImmutableSet.of(), debug);
}

public static Optional<SolverResult> entails(State state, Iterable<IConstraint> constraints, final Completeness completeness, final ITermVar[] localVars, final IDebugContext debug) throws InterruptedException, Delay {
    debug.info("Entails: {}, constraints: {}", Solver.toString(constraints, state.unifier()));
    final Set<ITermVar> localVars = ImmutableSet.copyOf(Arrays.asList(localVars));
    final Set<ITermVar> rigidVars = Sets.difference(state.vars(), localVars);
    final SolverResult result = Solver.solve(state, constraints, completeness, rigidVars::contains,
        localVars::contains, debug, subDebug);
    if(result.hasErrors()) {
        debug.info("Constraint not entailed");
        return Optional.empty();
    } else if(result.delays().isEmpty()) {
        debug.info("Constraint entailed");
        return Optional.of(result);
    } else {
        debug.info("Cannot decide constraint entailment (unsolved constraints)");
        throw result.delay(); // FIXME Remove local vars and scopes
    }
}

private static void printTrace(IConstraint constraint, IUnifier unifier, IDebugContext debug) {
    Nullable<IConstraint> constraint = constraint;
    while(constraint != null) {
        debug.error(" * ", constraint.toString(Solver.shallowTermFormatter(unifier)));
        constraint = constraint.cause().orElse(null);
    }
}

private static String toString(Iterable<IConstraint> constraints, IUnifier unifier) {
    final StringBuilder sb = new StringBuilder();
    boolean first = true;
    for(IConstraint constraint : constraints) {
        if(first) {
            first = false;
        } else {
            sb.append(", ");
        }
        sb.append(constraint.toString(Solver.shallowTermFormatter(unifier)));
    }
    return sb.toString();
}

@Value.Immutable
public static abstract class ASolverResult {
    @Value.Parameter public abstract State state();
    @Value.Parameter public abstract Completeness completeness();
    @Value.Parameter public abstract Set<IConstraint> errors();

    public boolean hasErrors() {
        return !errors().isEmpty();
    }

    @Value.Parameter public abstract Map<IConstraint, Delay> delays();

    public Delay delay() {
        ImmutableSet.Builder<ITermVar> vars = ImmutableSet.builder();
        ImmutableMultimap.Builder<ITerm, ITerm> scopes = ImmutableMultimap.builder();
        delays().values().stream().forEach(d -> {
            d.vars().addAll(vars);
            d.scopes().putAll(scopes);
        });
        return new Delay(vars.build(), scopes.build());
    }
}

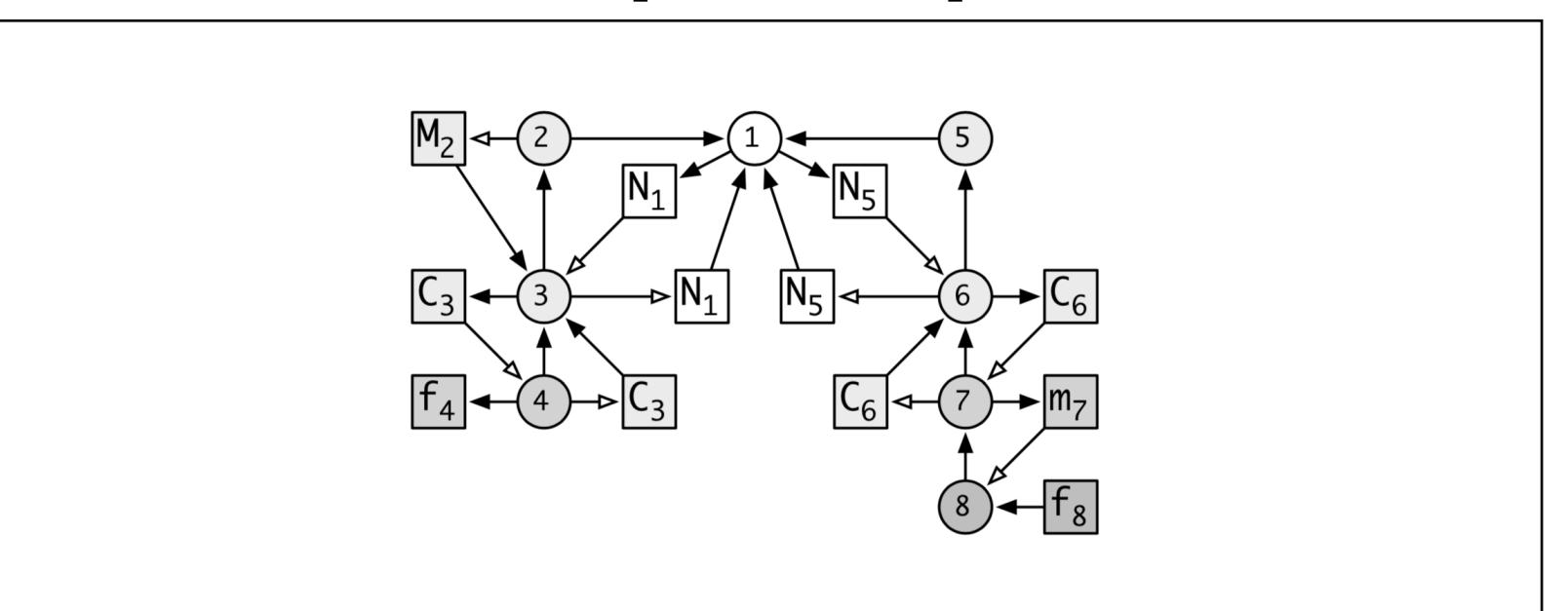
public static TermFormatter shallowTermFormatter(final IUnifier unifier) {
    return t -> unifier.toString(t, 3);
}

```

Typed Program

```
ype point = {x : num, y : num} in
et mkpoint = fun(x : num) { {x = x, y = x} } in
ype color = num in
ype colorpoint =
    {k : color} extends point in
et 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(finites)) = REC(rs) :-  

    new rs, fieldInitsOK(s, finites, 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 I-> [(_, _, 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

Structural types
Structural subtyping

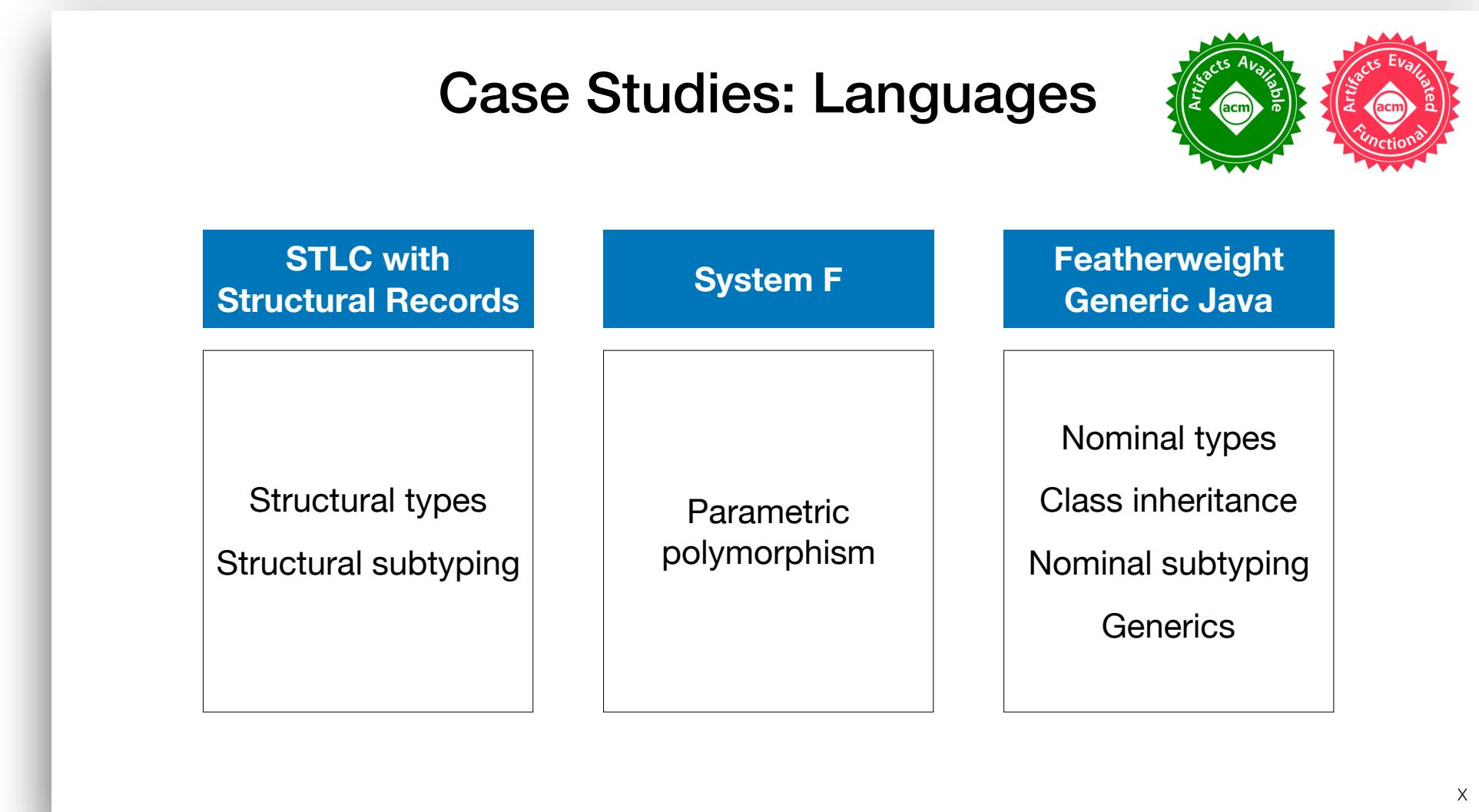
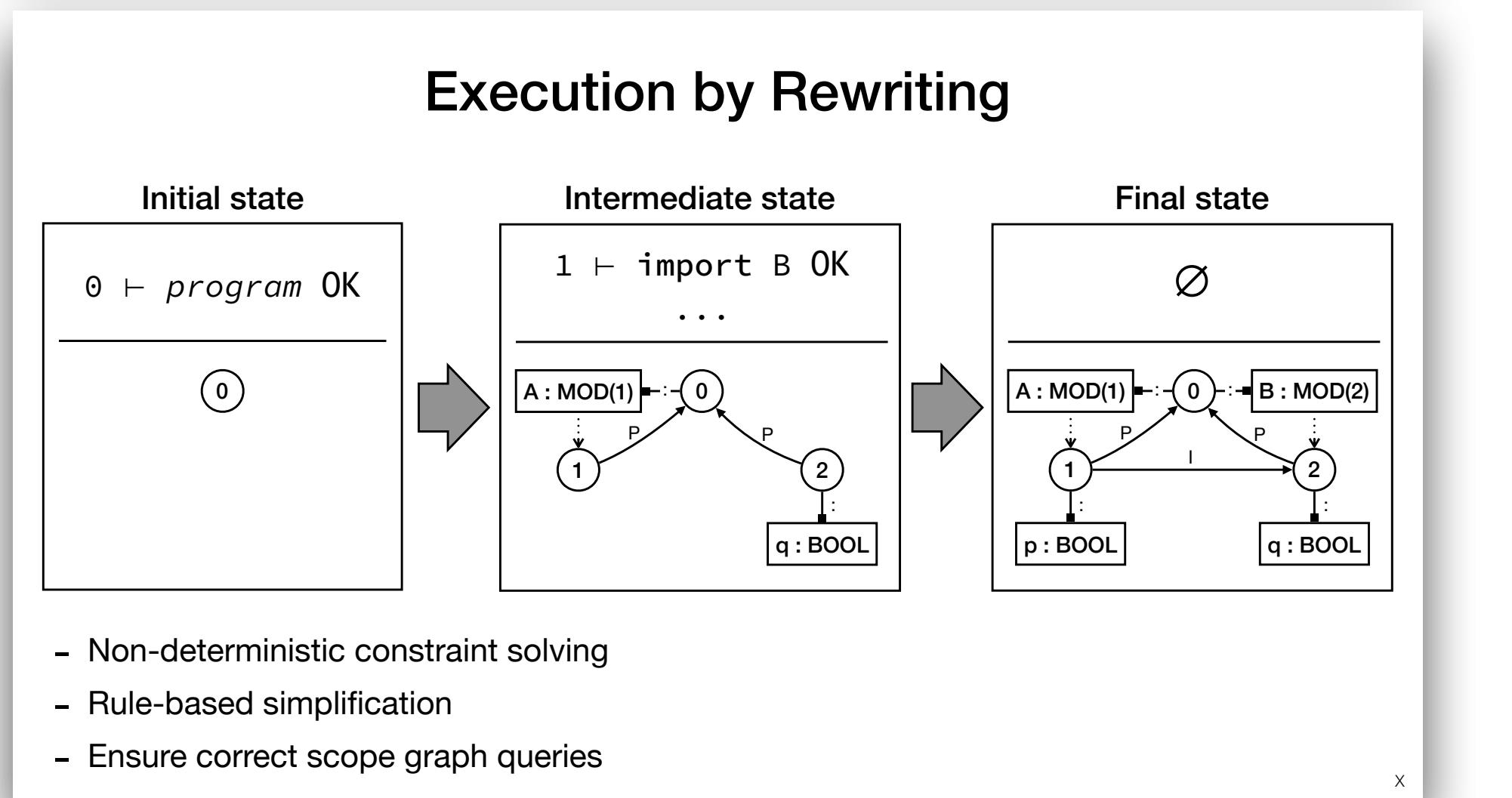
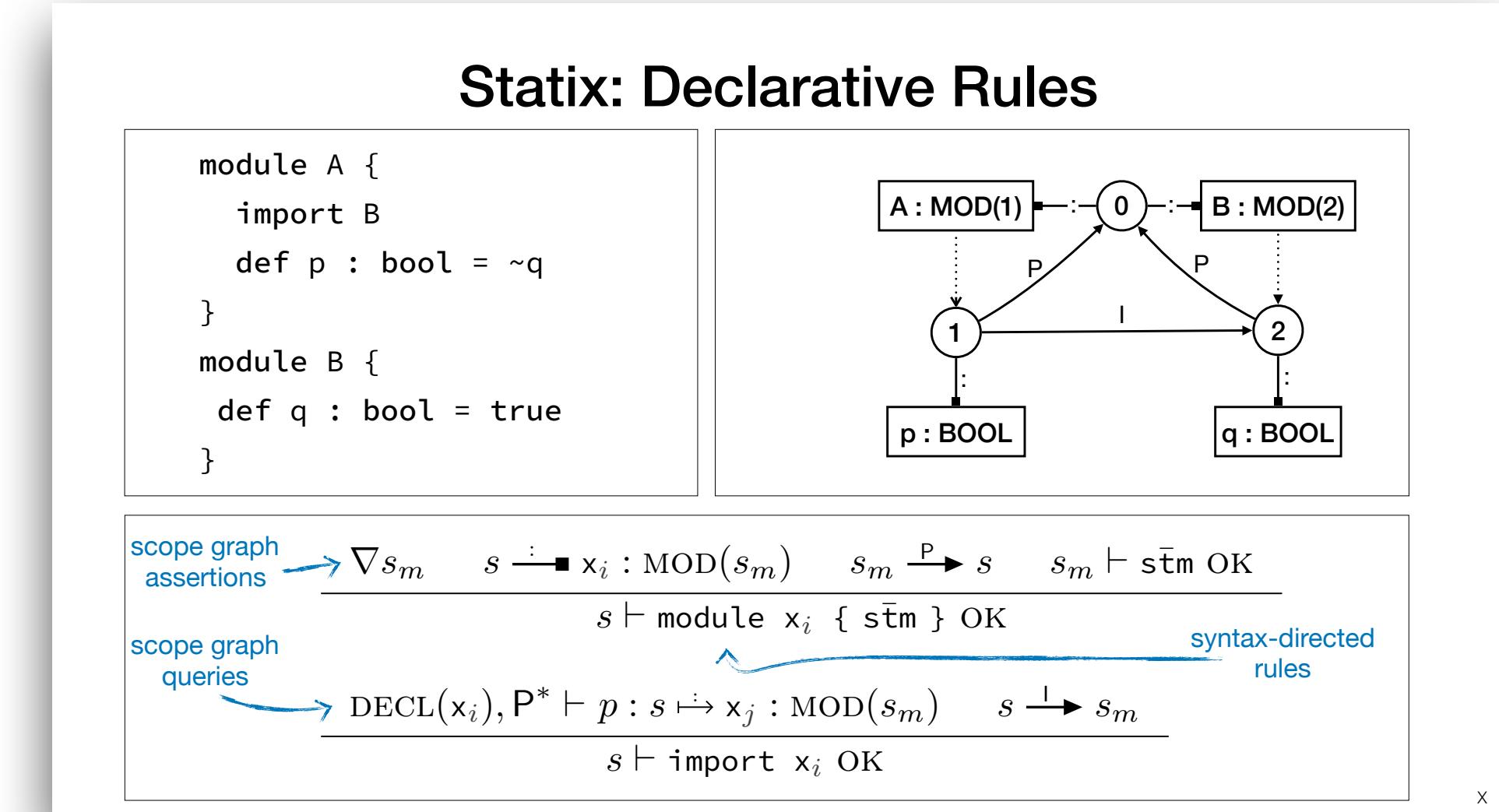
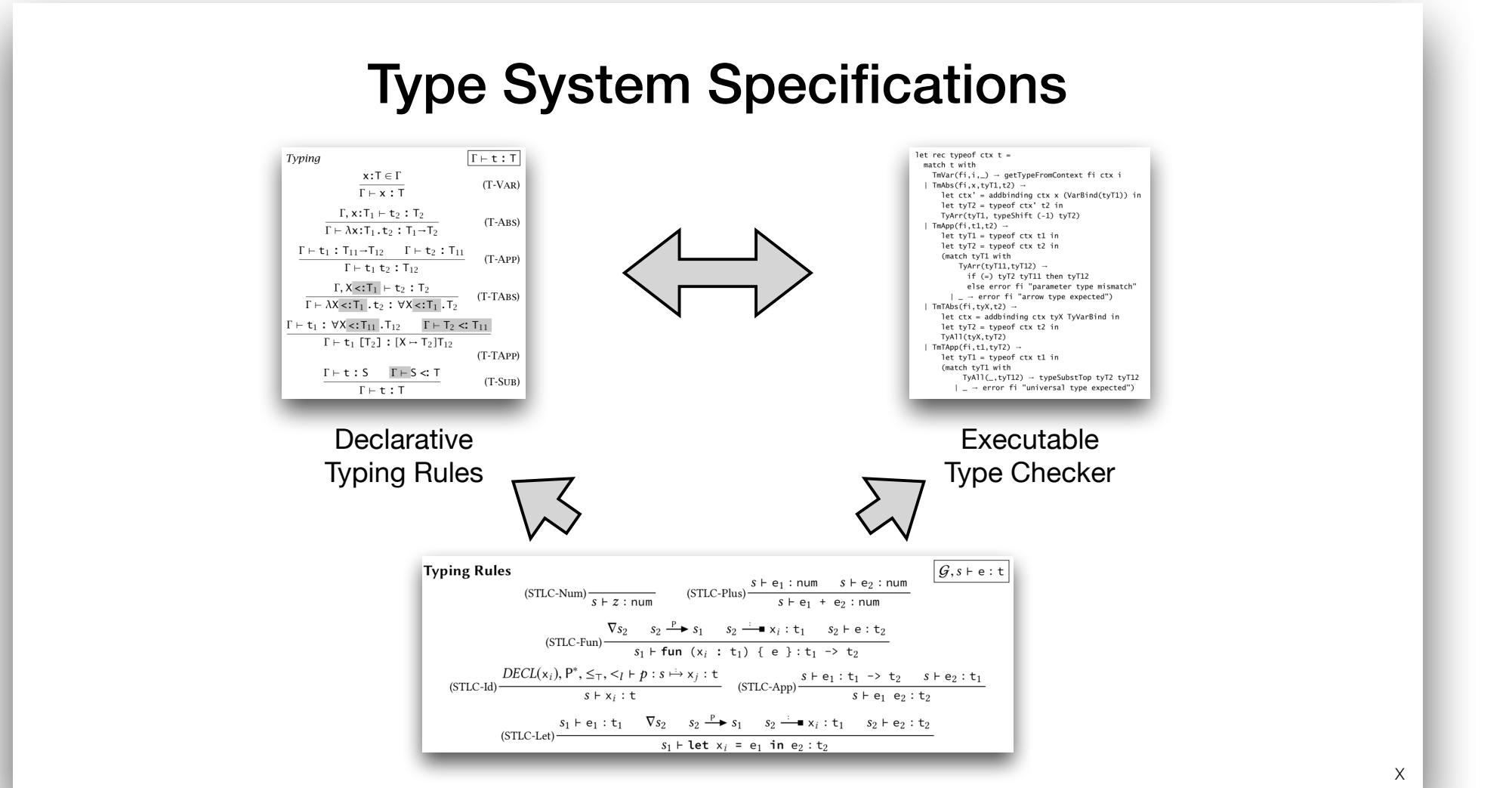
System F

Parametric polymorphism

Featherweight Generic Java

Nominal types
Class inheritance
Nominal subtyping
Generics

Declarative, Executable Type System Specifications



(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

Generalization of previous work*

Previous limitations:

- Structural types
- Parametric polymorphism/
generics

Improvements:

- Generalize declarations to
arbitrary data
- Generalize resolution to queries

Statix

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

Evaluation

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.