Statix

Checking & Generating Programs

Hendrik van Antwerpen
Delft University of Technology

FP Seminar, Chalmers, 30 August 2019

Type System Specifications

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

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

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, \mathtt{X} <: \mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X} <: \mathtt{T}_1 . \mathtt{t}_2 : \forall \mathtt{X} <: \mathtt{T}_1 . \mathtt{T}_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X} <: \mathtt{T}_{11} . \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{T}_2 <: \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \quad \text{(T-TApp)}$$

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

Declarative
Typing Rules

```
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")
```

Executable
Type Checker

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

$$\text{(STLC-Num)}\ \frac{}{s \vdash z : \mathtt{num}} \qquad \text{(STLC-Plus)}\ \frac{s \vdash \mathtt{e}_1 : \mathtt{num} \qquad s \vdash \mathtt{e}_2 : \mathtt{num}}{s \vdash \mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2 : \mathtt{num}}$$

$$\text{(STLC-Fun)}\ \frac{\nabla s_2 \qquad s_2 \xrightarrow{P} s_1 \qquad s_2 \xrightarrow{:}\blacksquare\ \mathtt{x}_i : \mathtt{t}_1 \qquad s_2 \vdash \mathtt{e} : \mathtt{t}_2}{s_1 \vdash \mathbf{fun}\ (\mathtt{x}_i\ :\ \mathtt{t}_1)\ \{\ \mathtt{e}\ \} : \mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2}$$

$$\text{(STLC-Id)}\ \frac{\mathit{DECL}(\mathtt{x}_i), \mathrm{P}^*, \leq_{\top}, <_l \vdash p : s \overset{:}{\mapsto} \mathtt{x}_j : \mathtt{t}}{s \vdash \mathtt{x}_i : \mathtt{t}} \qquad \text{(STLC-App)}\ \frac{s \vdash \mathtt{e}_1 : \mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2 \qquad s \vdash \mathtt{e}_2 : \mathtt{t}_1}{s \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{t}_2}$$

$$\text{(STLC-Let)}\ \frac{s_1 \vdash \mathtt{e}_1 : \mathtt{t}_1 \qquad \nabla s_2 \qquad s_2 \xrightarrow{P} s_1 \qquad s_2 \xrightarrow{:}\blacksquare\ \mathtt{x}_i : \mathtt{t}_1 \qquad s_2 \vdash \mathtt{e}_2 : \mathtt{t}_2}{s_1 \vdash \mathbf{let}\ \mathtt{x}_i\ \mathtt{=}\ \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2 : \mathtt{t}_2}$$

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	Execution Order
Many different representations	Interleaving
Often language-specific	Staging
Ad-hoc, not reusable	Not declarative

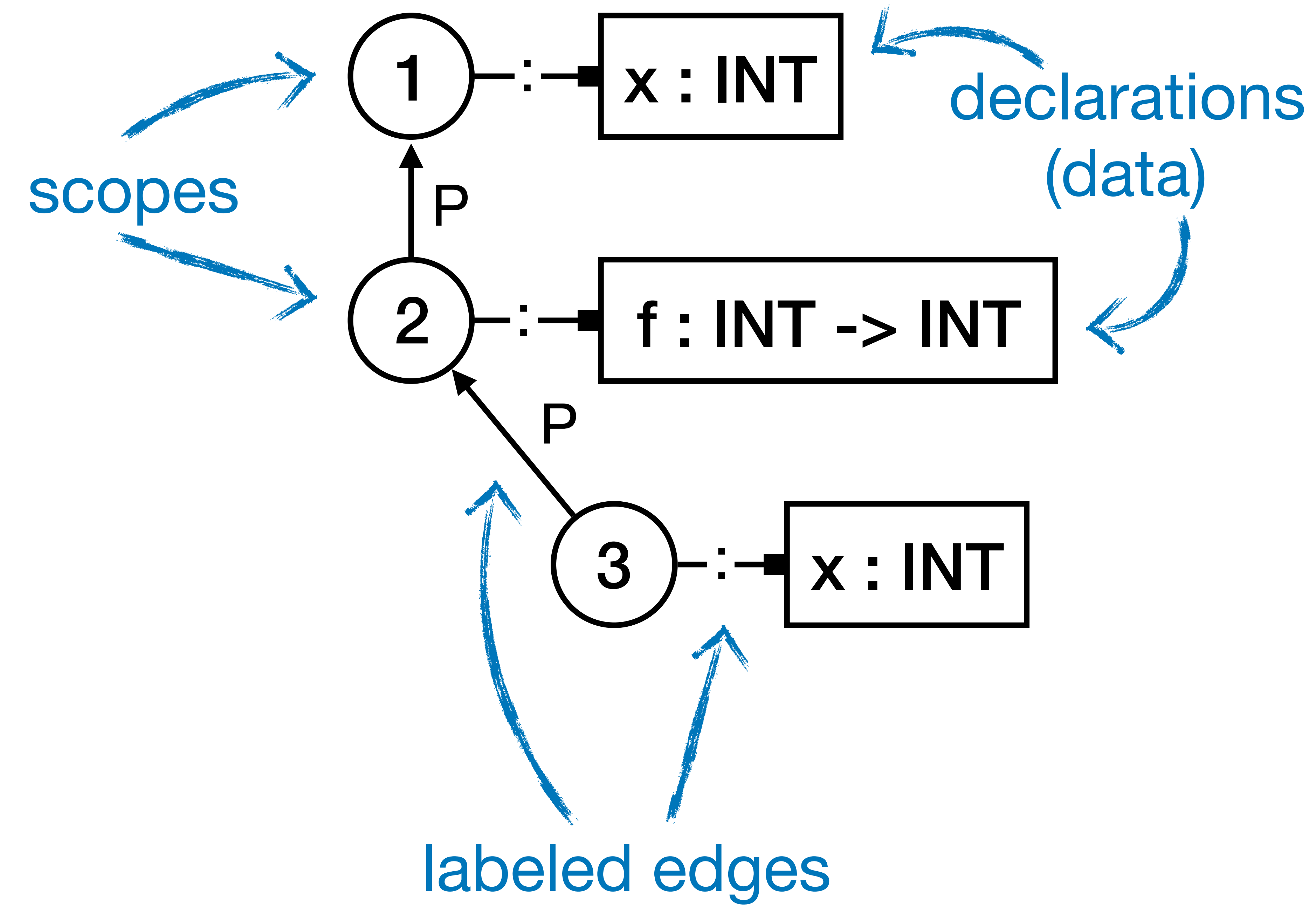
Our Approach

Scope Graphs

Statix

Scope Graph: Lexical

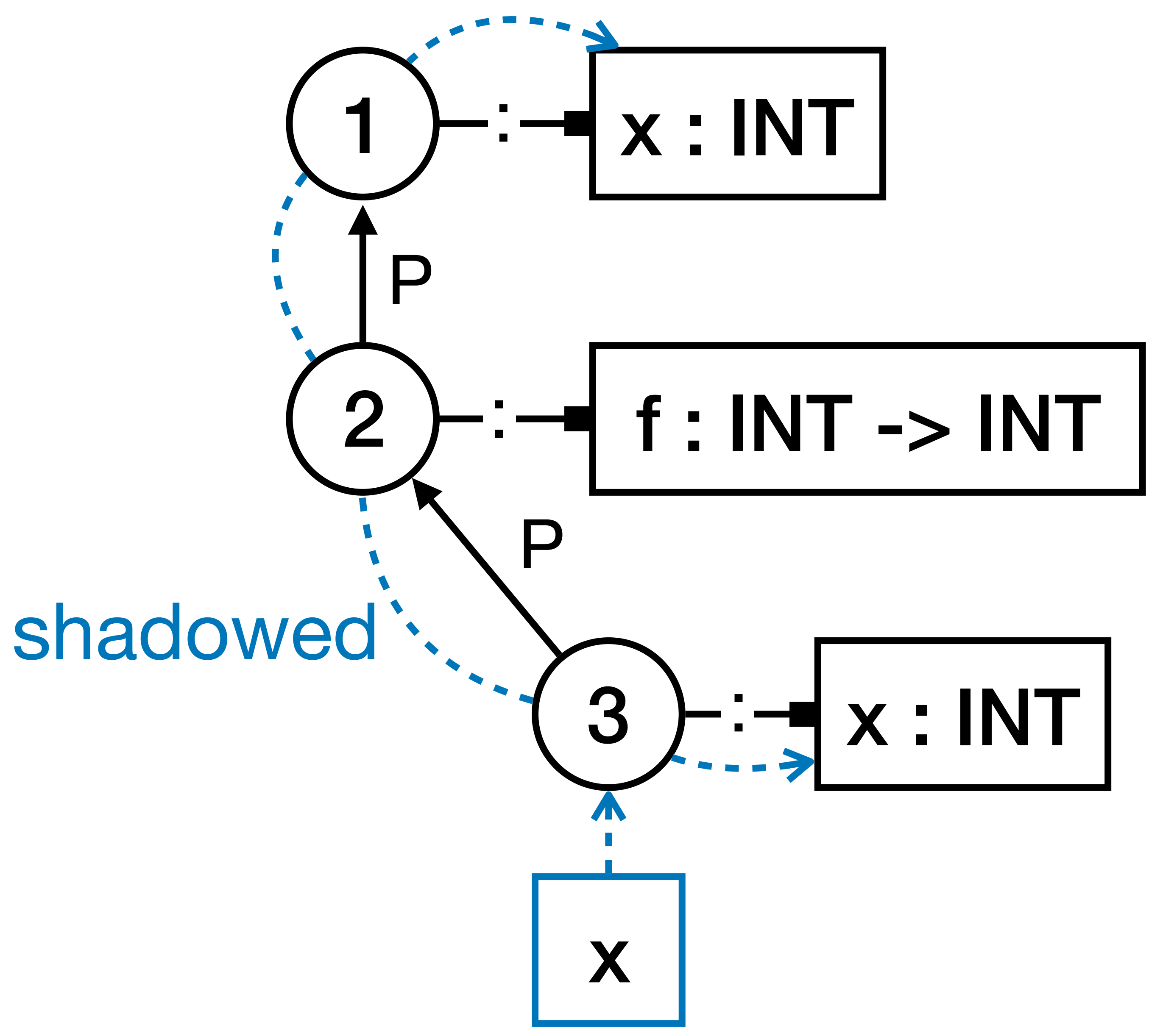
```
let
  x : int = 7
in
  let
    f : int -> int = fun (x:int) { x * 3 }
  in
    f x
```



Scope Graph: Lexical

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

shadowed



Query

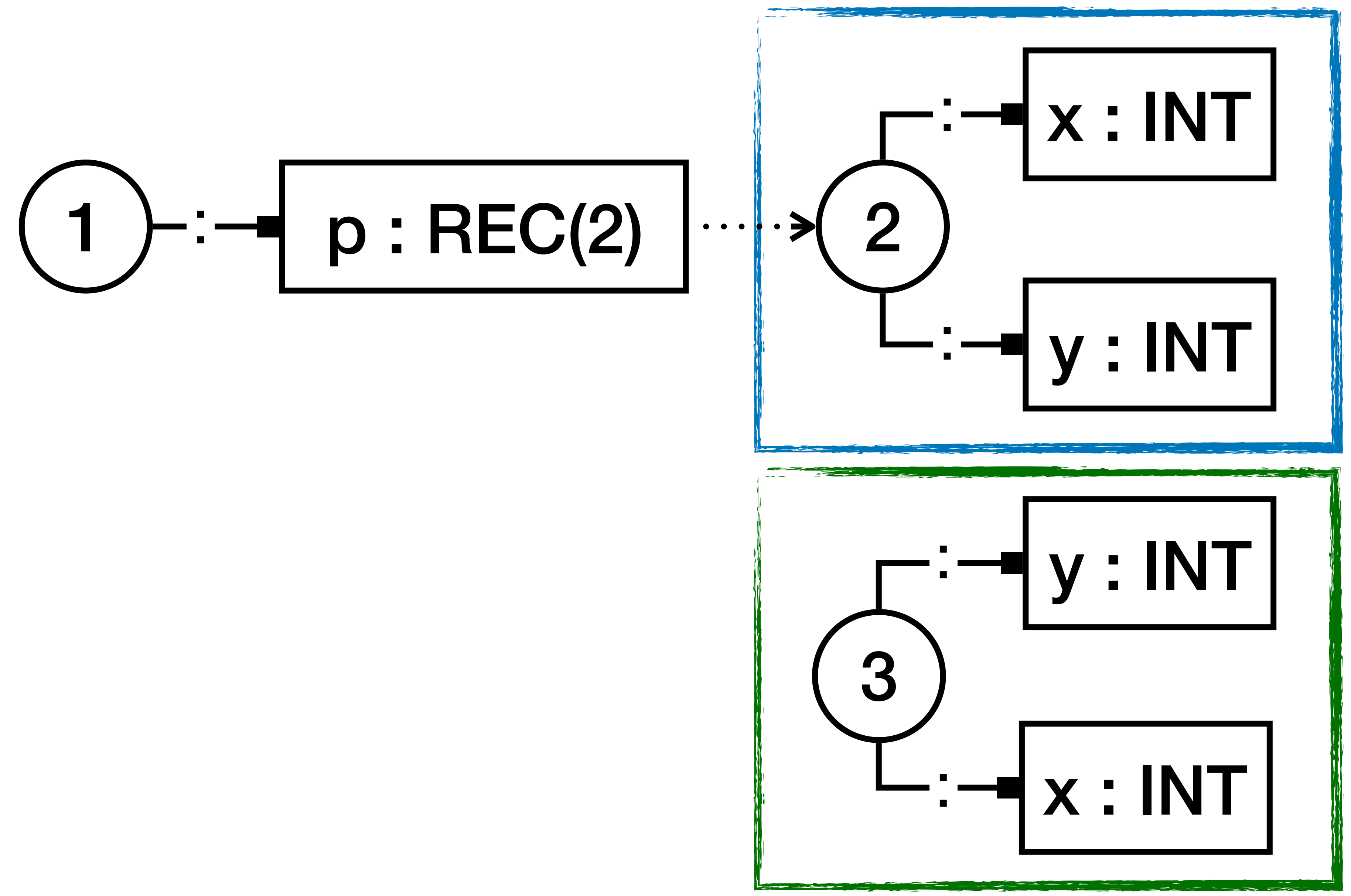
P* (allow any number of P steps)

$ < 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
```

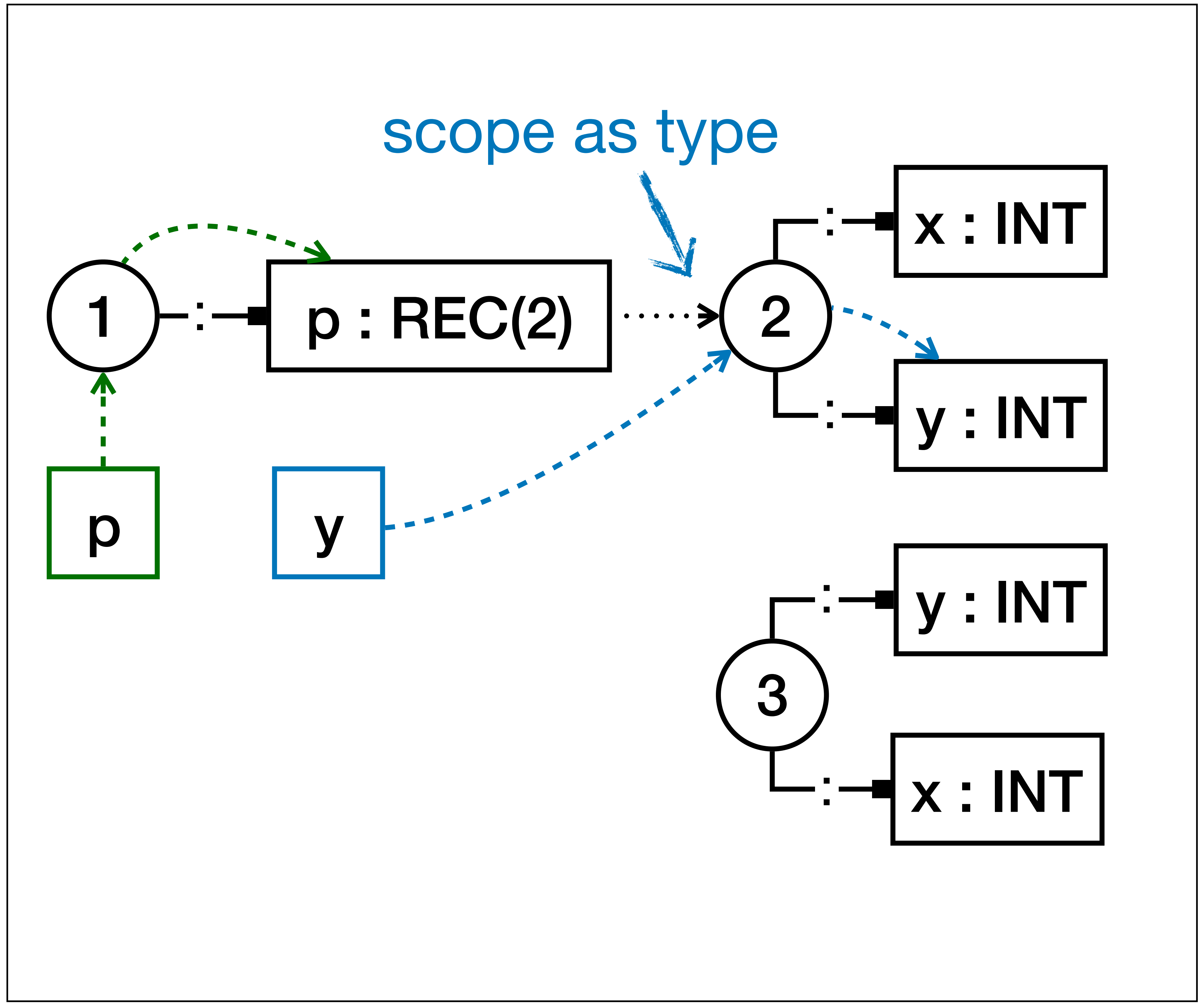


- Type structure described by scopes

Scope Graph: Records

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

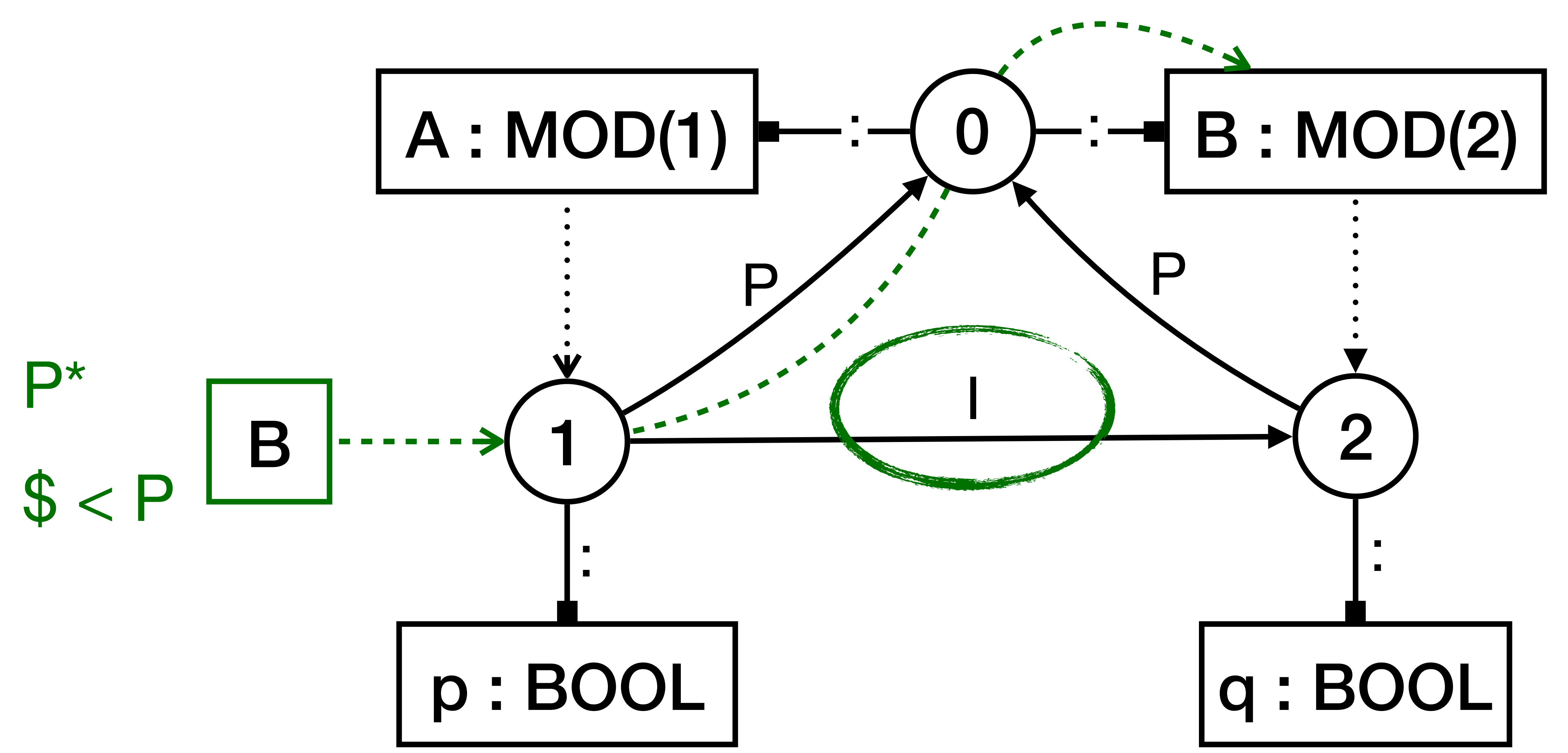
type-dependent
name resolution



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

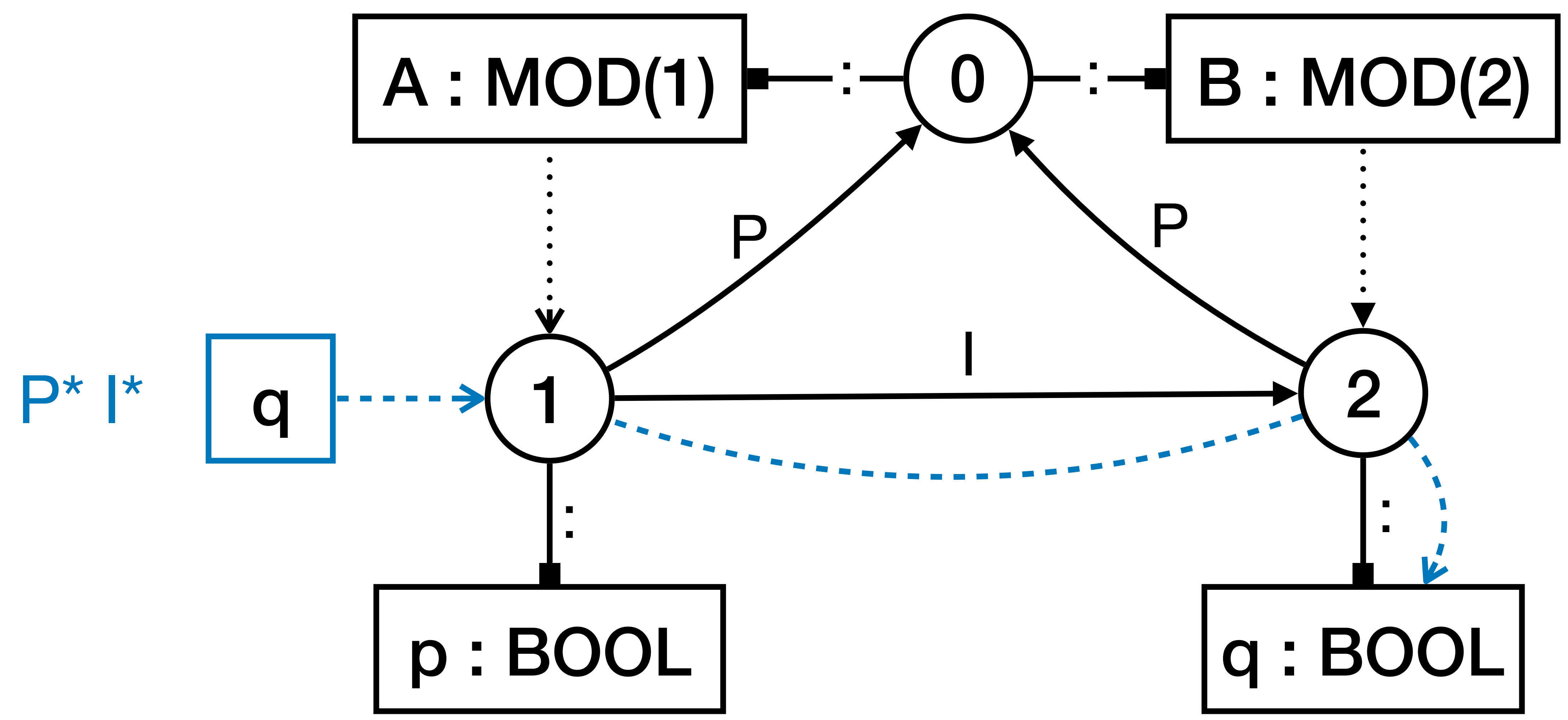
Scope Graph: Modules

```
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

Language independent

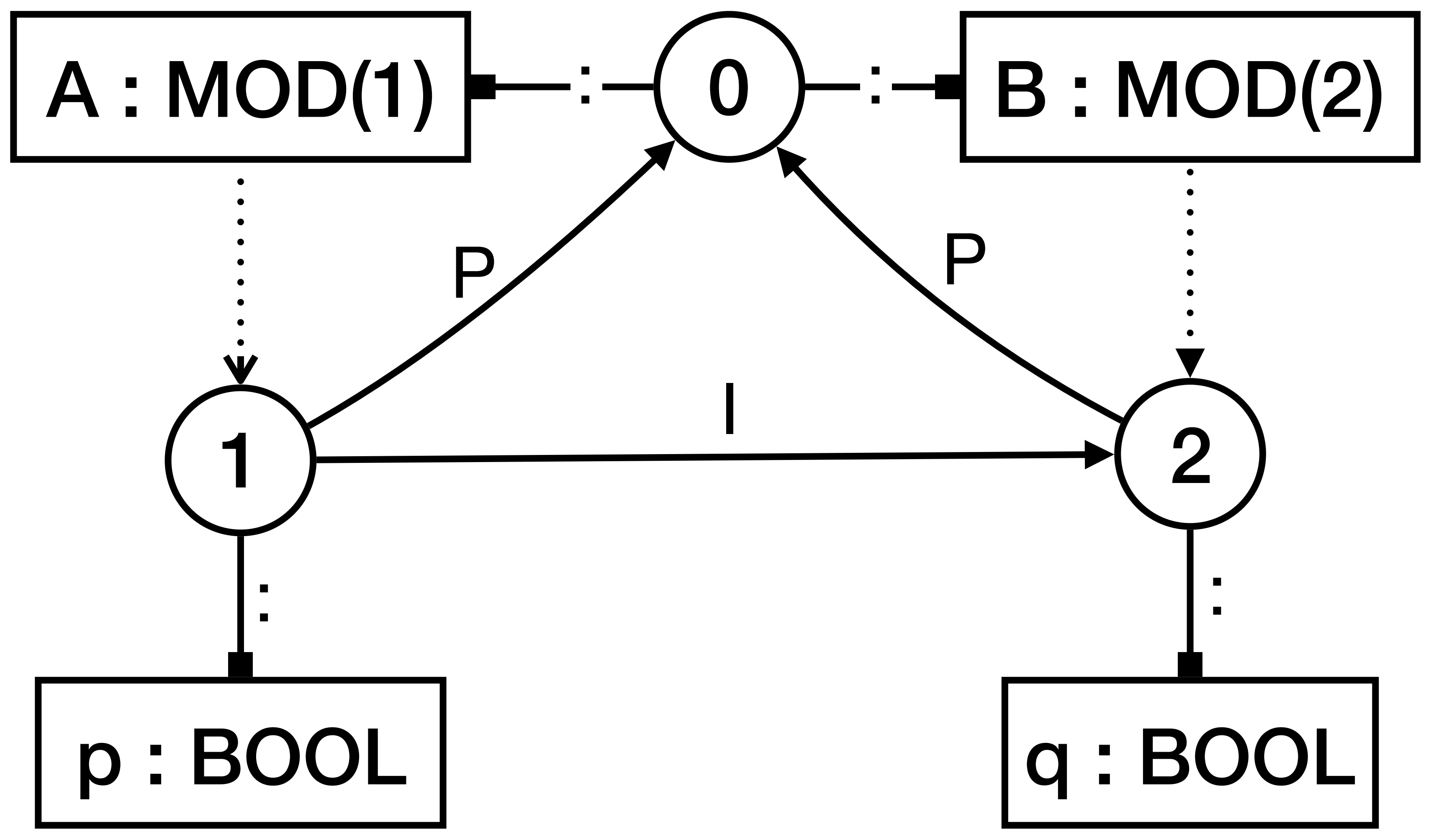
Capture different kinds of binding

Resolve names by graph queries

Statix

Statix: Declarative Rules

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



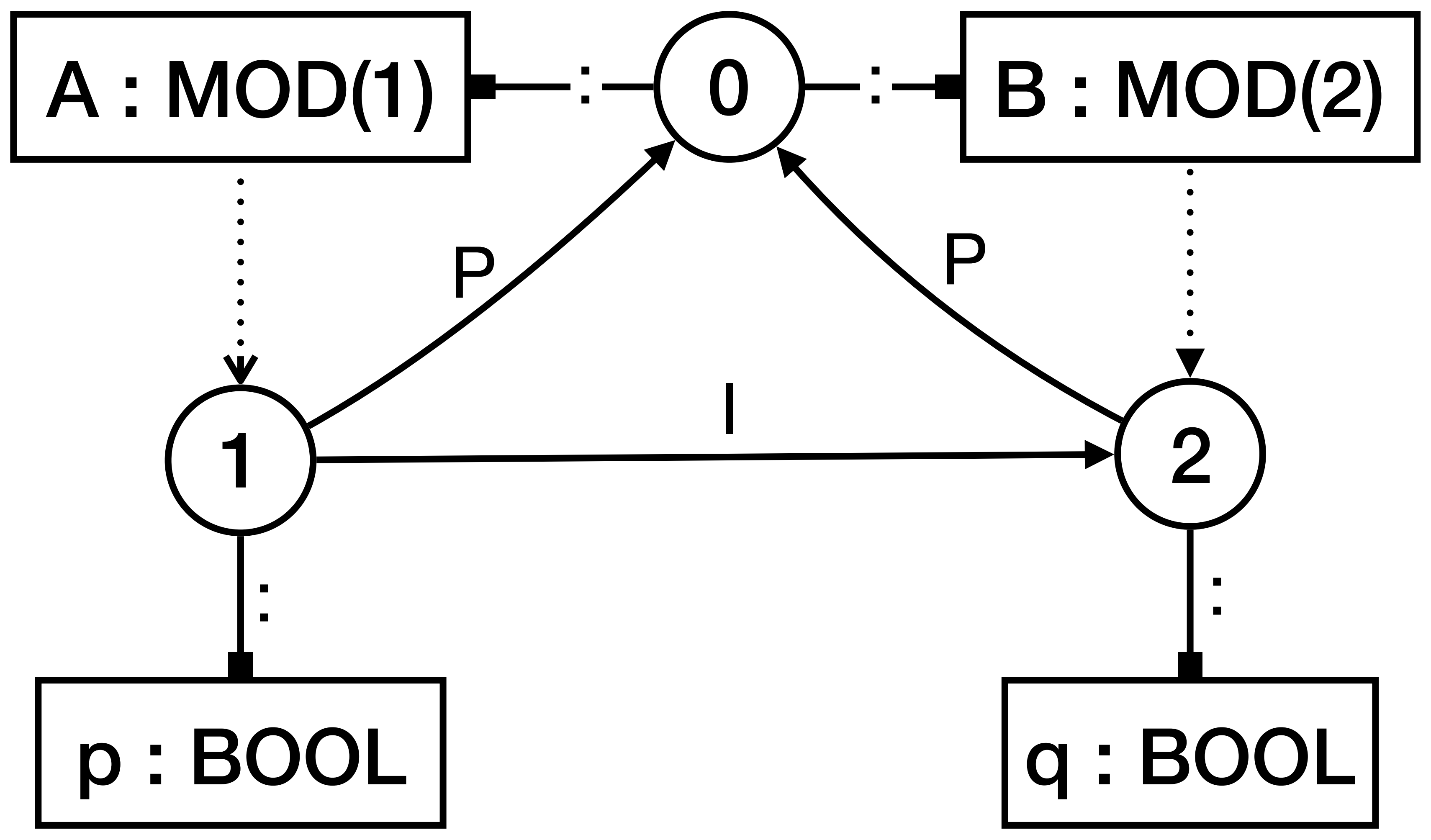

```
progOk : list(Mod)
progOk(mods) :- {s_prog}
  new s_prog,
  modsOk(s_prog, mods).

modsOk maps modOk(*, list(*))

modOk : scope * Mod
modOk(s, Mod(x, defs)) :- {s_mod}
  new s_mod,
  s_mod -P-> s,
  s_mod -> Mod{x@x} with typeOf MOD(s_mod),
  defsOk(s_mod, defs).

defsOk maps defOk(*, list(*))

defOk : scope * Def
defOk(s, Import(x)) :- {d s'}
  typeOf of Mod{x@x} in s |-> [(_,(d,MOD(s')))],
  s -I-> s'.
```

predicate signature and rules

syntax-directed rules

Statix: Declarative Rules

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




```
progOk : list(Mod)
progOk(mods) :- {s_prog}
  new s_prog,
  modsOk(s_prog, mods).

modsOk maps modOk(*, list(*))

modOk : scope * Mod
modOk(s, Mod(x, defs)) :- {s_mod}
  new s_mod,
  s_mod -P-> s,
  s_mod -> Mod{x@x} with typeOf MOD(s_mod),
  defsOk(s_mod, defs).

defsOk maps defOk(*, list(*))

defOk : scope * Def
defOk(s, Import(x)) :- {d s'}
  typeOf of Mod{x@x} in s |-> [(_,(d,MOD(s')))],
  s -I-> s'.
```

scope assertion

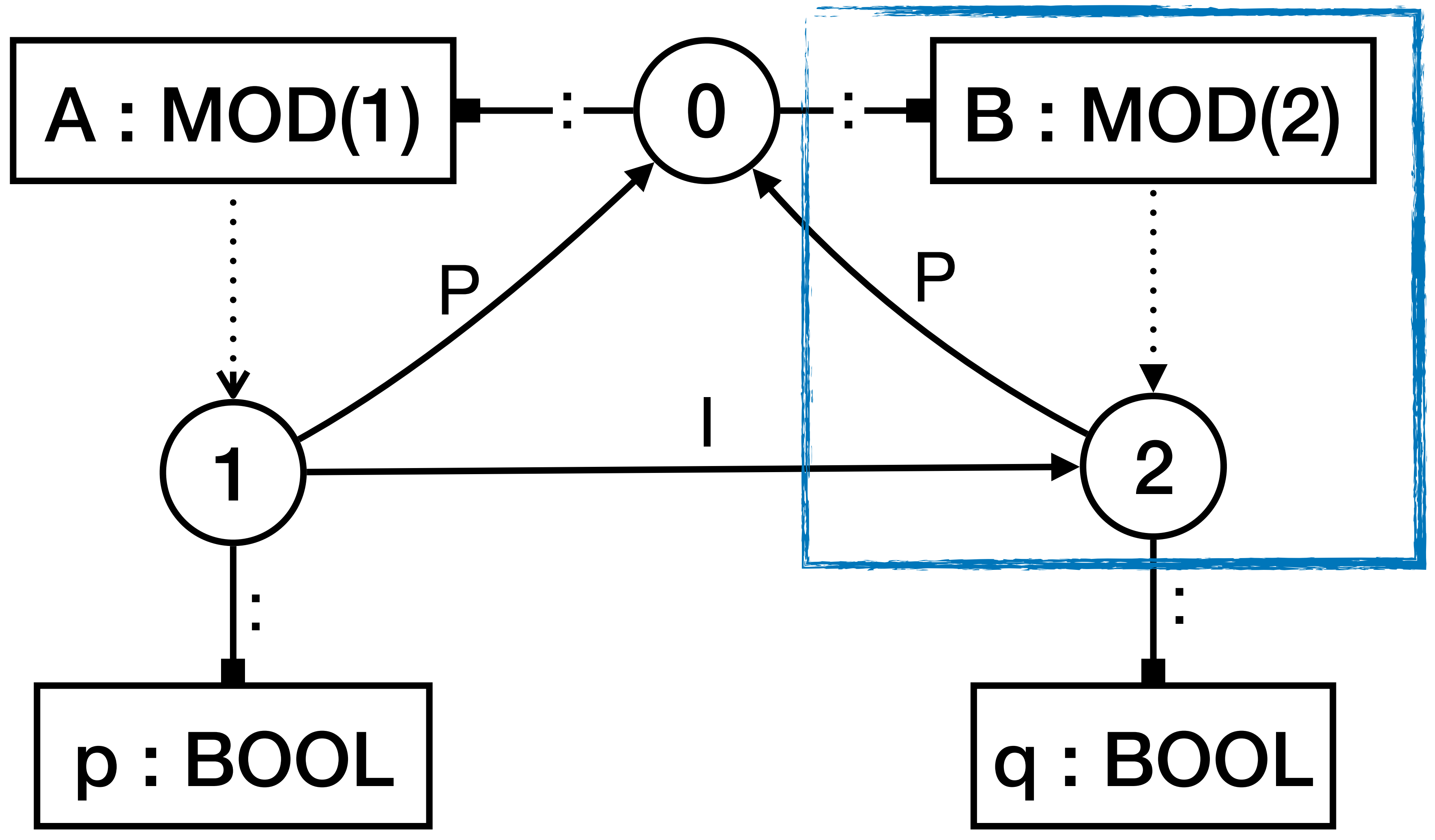
edge assertion

data assertion

resolution query

Statix: Declarative Rules

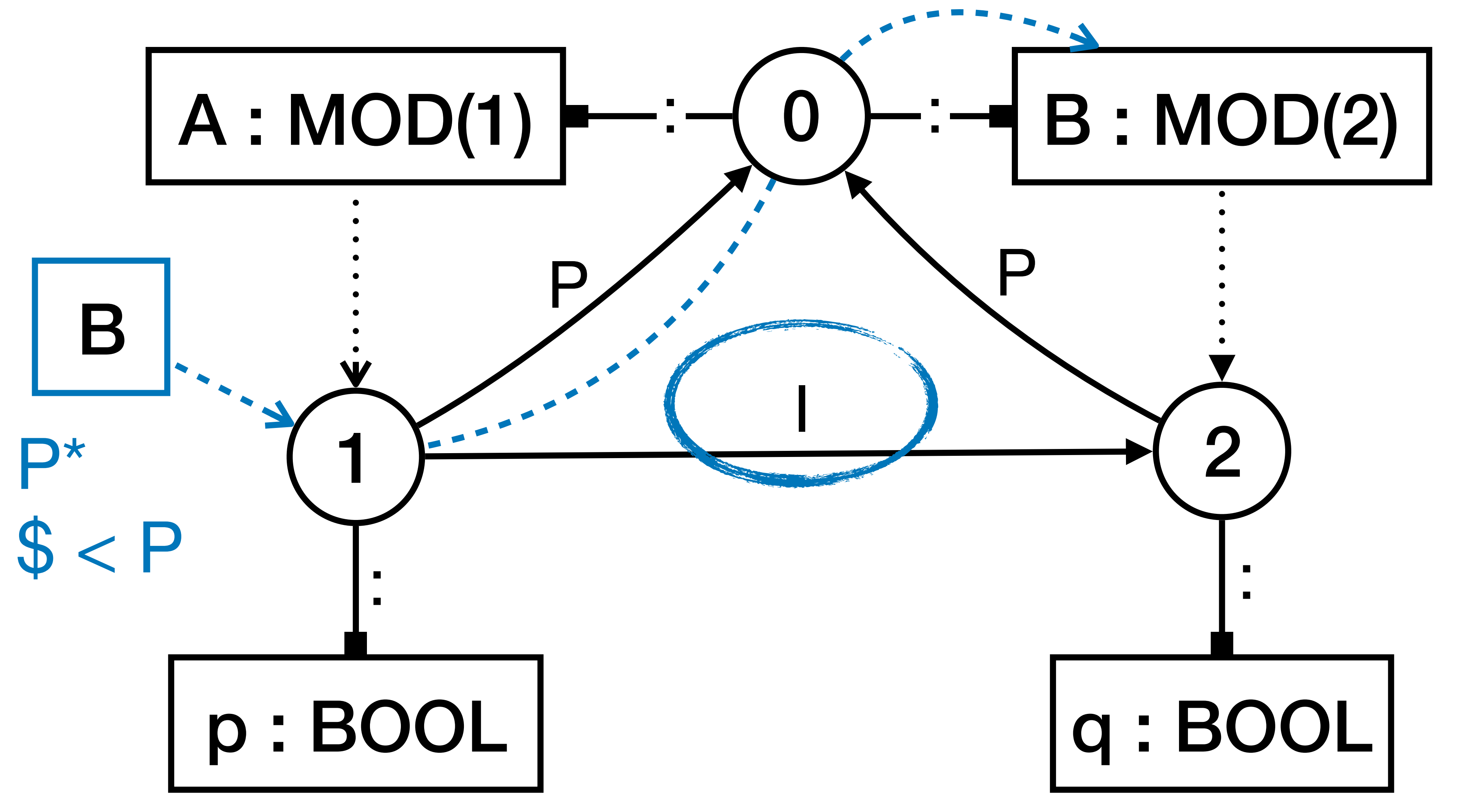
```
module A {
  import B
  def p : bool = ~q
}
module B {
  def q : bool = true
}
```




```
progOk : list(Mod)
progOk(mods) :- {s_prog}
  new s_prog,
  modsOk(s_prog, mods).

modsOk maps modOk(*, list(*))

modOk : scope * Mod
modOk(s, Mod(x, defs)) :- {s_mod}
  new s_mod,
  s_mod -P-> s,
  s_mod -> Mod{x@x} with typeOf MOD(s_mod),
  defsOk(s_mod, defs).

defsOk maps defOk(*, list(*))

defOk : scope * Def
defOk(s, Import(x)) :- {d s'}
  typeOf of Mod{x@x} in s |-> [(_,(d,MOD(s')))],
  s -I-> s'.
```

Statix: Declarative Rules

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




```
progOk : list(Mod)
progOk(mods) :- {s_prog}
  new s_prog,
  modsOk(s_prog, mods).

modsOk maps modOk(*, list(*))

modOk : scope * Mod
modOk(s, Mod(x, defs)) :- {s_mod}
  new s_mod,
  s_mod -P-> s,
  s_mod -> Mod{x@x} with typeOf MOD(s_mod),
  defsOk(s_mod, defs).

defsOk maps defOk(*, list(*))

defOk : scope * Def
defOk(s, Import(x)) :- {d s'}
  typeOf of Mod{x@x} in s |-> [(_,(d,MOD(s')))],
  s -I-> s'.
```

Our Approach

Scope Graphs

Language independent

Capture different kinds of binding

Resolve names by graph queries

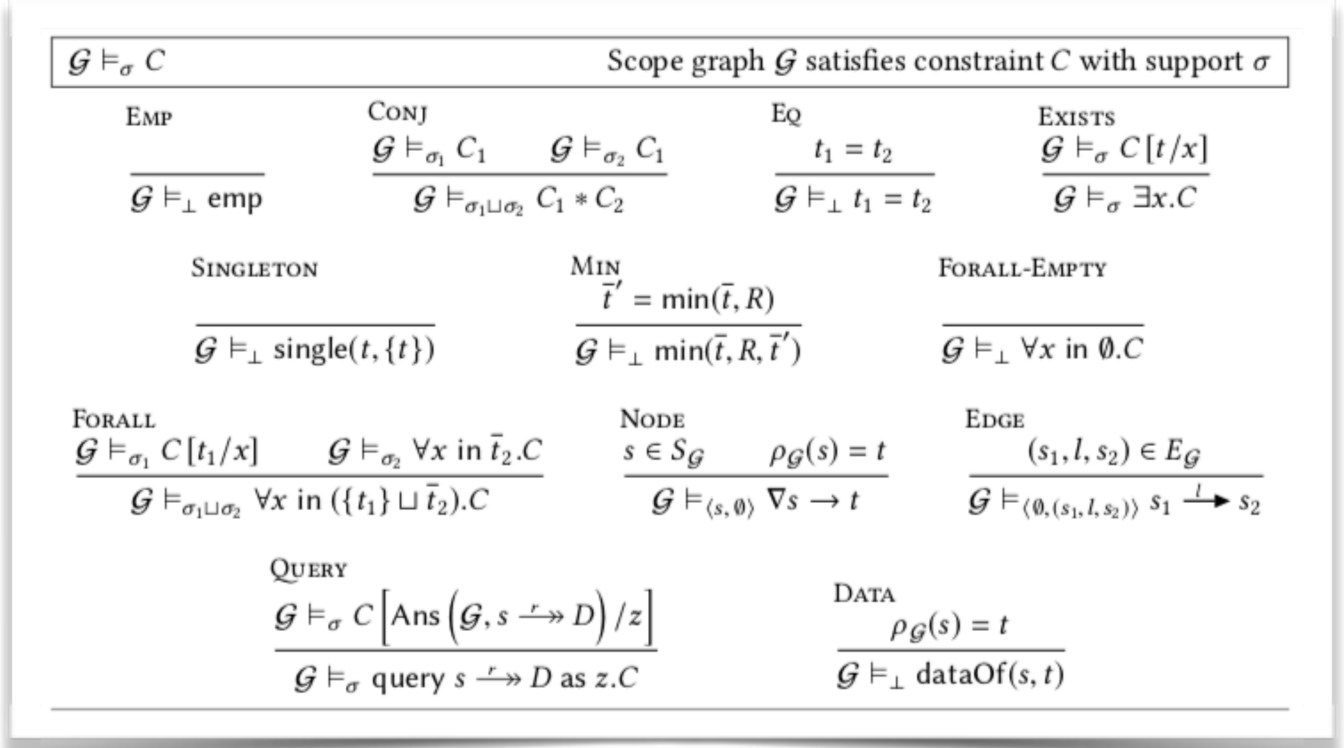
Statix

Declarative specifications

Abstracts over execution order

Constraint solver

The Meaning of Statix



$\mathcal{G} \vDash_\sigma C$ — Scope graph $\mathcal{G}$ satisfies constraint C with support σ

Emp
$$\frac{}{\mathcal{G} \vDash_\bot \mathsf{emp}}$$

Conj
$$\frac{\mathcal{G} \vDash_{\sigma_1} C_1 \qquad \mathcal{G} \vDash_{\sigma_2} C_1}{\mathcal{G} \vDash_{\sigma_1 \sqcup \sigma_2} C_1 * C_2}$$

Eq
$$\frac{t_1 = t_2}{\mathcal{G} \vDash_\bot t_1 = t_2}$$

Exists
$$\frac{\mathcal{G} \vDash_\sigma C[t/x]}{\mathcal{G} \vDash_\sigma \exists x.C}$$

Singleton
$$\frac{}{\mathcal{G} \vDash_\bot \mathsf{single}(t, \{t\})}$$

Min
$$\frac{\bar{t}' = \mathsf{min}(\bar{t}, R)}{\mathcal{G} \vDash_\bot \mathsf{min}(\bar{t}, R, \bar{t}')}$$

Forall-Empty
$$\frac{}{\mathcal{G} \vDash_\bot \forall x \text{ in } \emptyset.C}$$

Forall
$$\frac{\mathcal{G} \vDash_{\sigma_1} C[t_1/x] \qquad \mathcal{G} \vDash_{\sigma_2} \forall x \text{ in } \bar{t}_2.C}{\mathcal{G} \vDash_{\sigma_1 \sqcup \sigma_2} \forall x \text{ in } (\{t_1\} \sqcup \bar{t}_2).C}$$

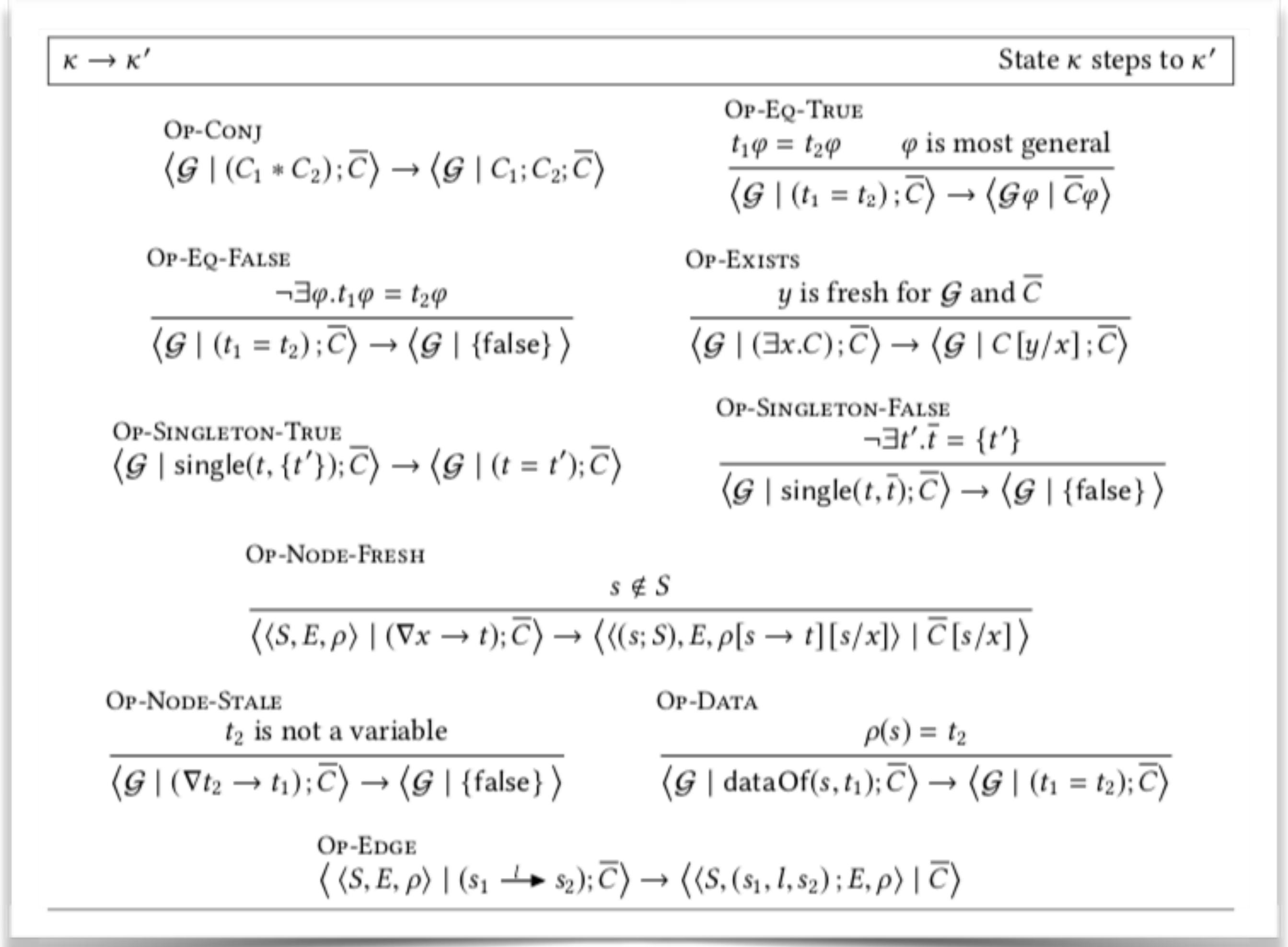
Node
$$\frac{s \in S_{\mathcal{G}} \qquad \rho_{\mathcal{G}}(s) = t}{\mathcal{G} \vDash_{\langle s, \emptyset \rangle} \nabla s \to t}$$

Edge
$$\frac{(s_1, l, s_2) \in E_{\mathcal{G}}}{\mathcal{G} \vDash_{\langle \emptyset, (s_1, l, s_2) \rangle} s_1 \xrightarrow{l} s_2}$$

Query
$$\frac{\mathcal{G} \vDash_\sigma C\left[\mathsf{Ans}\left(\mathcal{G}, s \xrightarrow{r} D\right) / z\right]}{\mathcal{G} \vDash_\sigma \mathsf{query}\ s \xrightarrow{r} D \text{ as } z.C}$$

Data
$$\frac{\rho_{\mathcal{G}}(s) = t}{\mathcal{G} \vDash_\bot \mathsf{dataOf}(s, t)}$$

Declarative Semantics



$\kappa \to \kappa'$ — State κ steps to κ'

Op-Conj
$$\langle \mathcal{G} \mid (C_1 * C_2); \overline{C} \rangle \to \langle \mathcal{G} \mid C_1; C_2; \overline{C} \rangle$$

Op-Eq-True
$$\frac{t_1\varphi = t_2\varphi \qquad \varphi \text{ is most general}}{\langle \mathcal{G} \mid (t_1 = t_2); \overline{C} \rangle \to \langle \mathcal{G}\varphi \mid \overline{C}\varphi \rangle}$$

Op-Eq-False
$$\frac{\neg\exists\varphi.t_1\varphi = t_2\varphi}{\langle \mathcal{G} \mid (t_1 = t_2); \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

Op-Exists
$$\frac{y \text{ is fresh for } \mathcal{G} \text{ and } \overline{C}}{\langle \mathcal{G} \mid (\exists x.C); \overline{C} \rangle \to \langle \mathcal{G} \mid C[y/x]; \overline{C} \rangle}$$

Op-Singleton-True
$$\langle \mathcal{G} \mid \mathsf{single}(t, \{t'\}); \overline{C} \rangle \to \langle \mathcal{G} \mid (t = t'); \overline{C} \rangle$$

Op-Singleton-False
$$\frac{\neg\exists t'.\bar{t} = \{t'\}}{\langle \mathcal{G} \mid \mathsf{single}(t, \bar{t}); \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

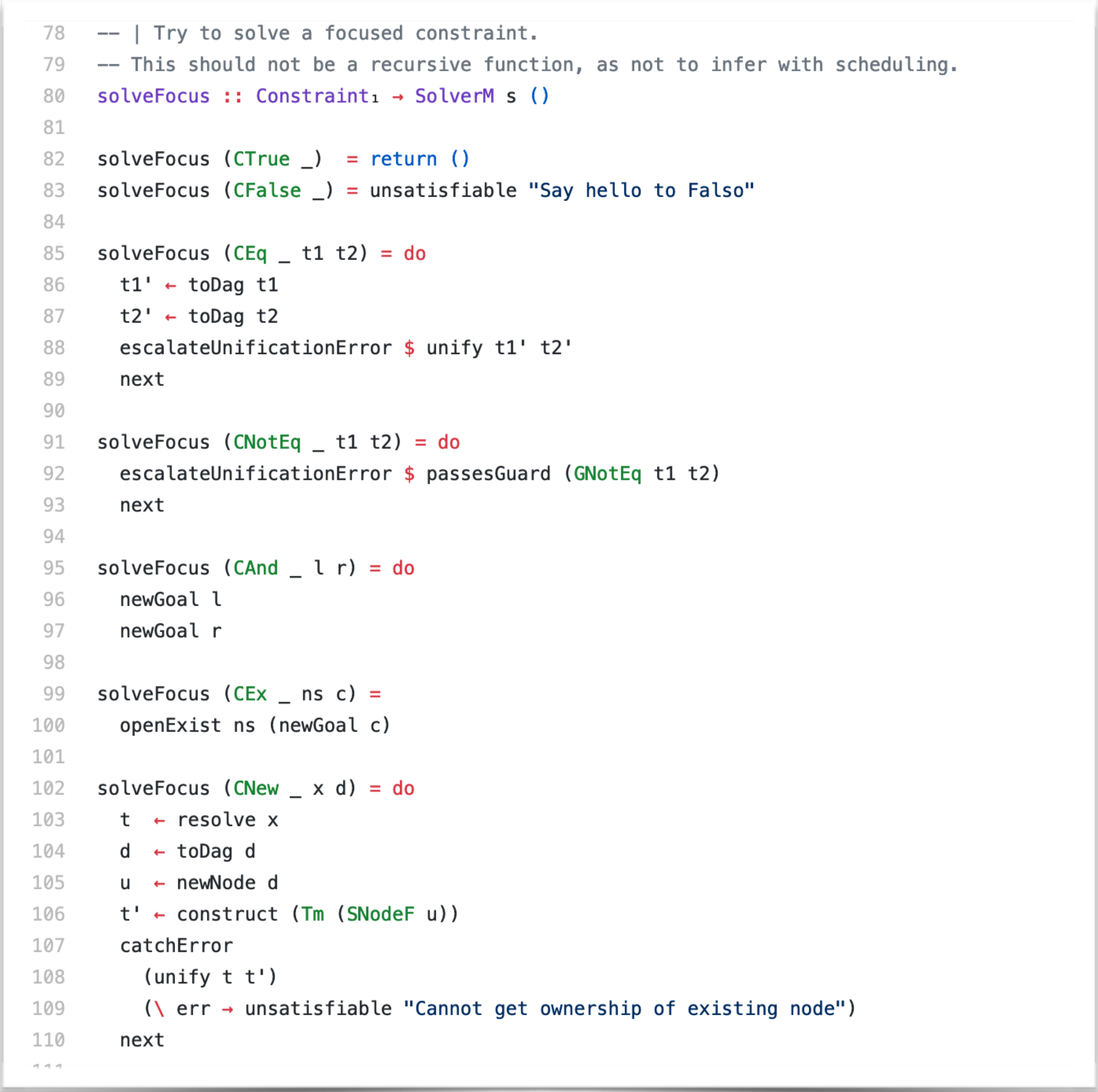
Op-Node-Fresh
$$\frac{s \notin S}{\langle \langle S, E, \rho \rangle \mid (\nabla x \to t); \overline{C} \rangle \to \langle \langle (s; S), E, \rho[s \to t][s/x] \rangle \mid \overline{C}[s/x] \rangle}$$

Op-Node-Stale
$$\frac{t_2 \text{ is not a variable}}{\langle \mathcal{G} \mid (\nabla t_2 \to t_1); \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

Op-Data
$$\frac{\rho(s) = t_2}{\langle \mathcal{G} \mid \mathsf{dataOf}(s, t_1); \overline{C} \rangle \to \langle \mathcal{G} \mid (t_1 = t_2); \overline{C} \rangle}$$

Op-Edge
$$\langle \langle S, E, \rho \rangle \mid (s_1 \xrightarrow{l} s_2); \overline{C} \rangle \to \langle \langle S, (s_1, l, s_2); E, \rho \rangle \mid \overline{C} \rangle$$

Small-step Operational Semantics

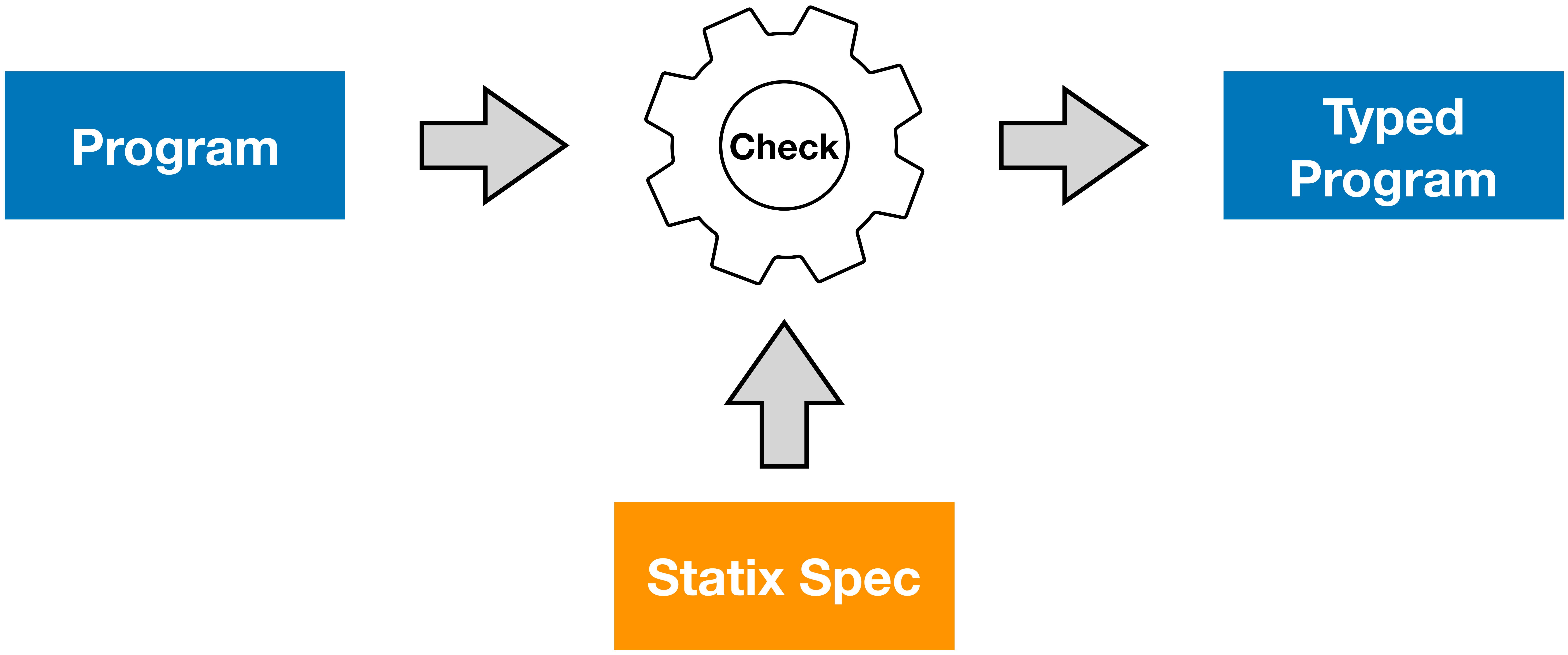


```haskell
-- | Try to solve a focused constraint.
-- This should not be a recursive function, as not to infer with scheduling.
solveFocus :: Constraint₁ → SolverM s ()

solveFocus (CTrue _)  = return ()
solveFocus (CFalse _) = unsatisfiable "Say hello to Falso"

solveFocus (CEq _ t1 t2) = do
  t1' ← toDag t1
  t2' ← toDag t2
  escalateUnificationError $ unify t1' t2'
  next

solveFocus (CNotEq _ t1 t2) = do
  escalateUnificationError $ passesGuard (GNotEq t1 t2)
  next

solveFocus (CAnd _ l r) = do
  newGoal l
  newGoal r

solveFocus (CEx _ ns c) =
  openExist ns (newGoal c)

solveFocus (CNew _ x d) = do
  t  ← resolve x
  d  ← toDag d
  u  ← newNode d
  t' ← construct (Tm (SNodeF u))
  catchError
    (unify t t')
    (\ err → unsatisfiable "Cannot get ownership of existing node")
  next
```

Implementation
(Java & Haskell)

https://github.com/ajrouvoet/ministatix.hs

Type Checking with Statix



Literature

Scope Graphs and Static Semantics

- Néron Pierre, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. *A Theory of Name Resolution.* ESOP 2015.
- Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. *A Constraint Language for Static Semantic Analysis Based on Scope Graphs.* PEPM 2016.
- Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. *Scopes As Types.* OOPSLA 2018.

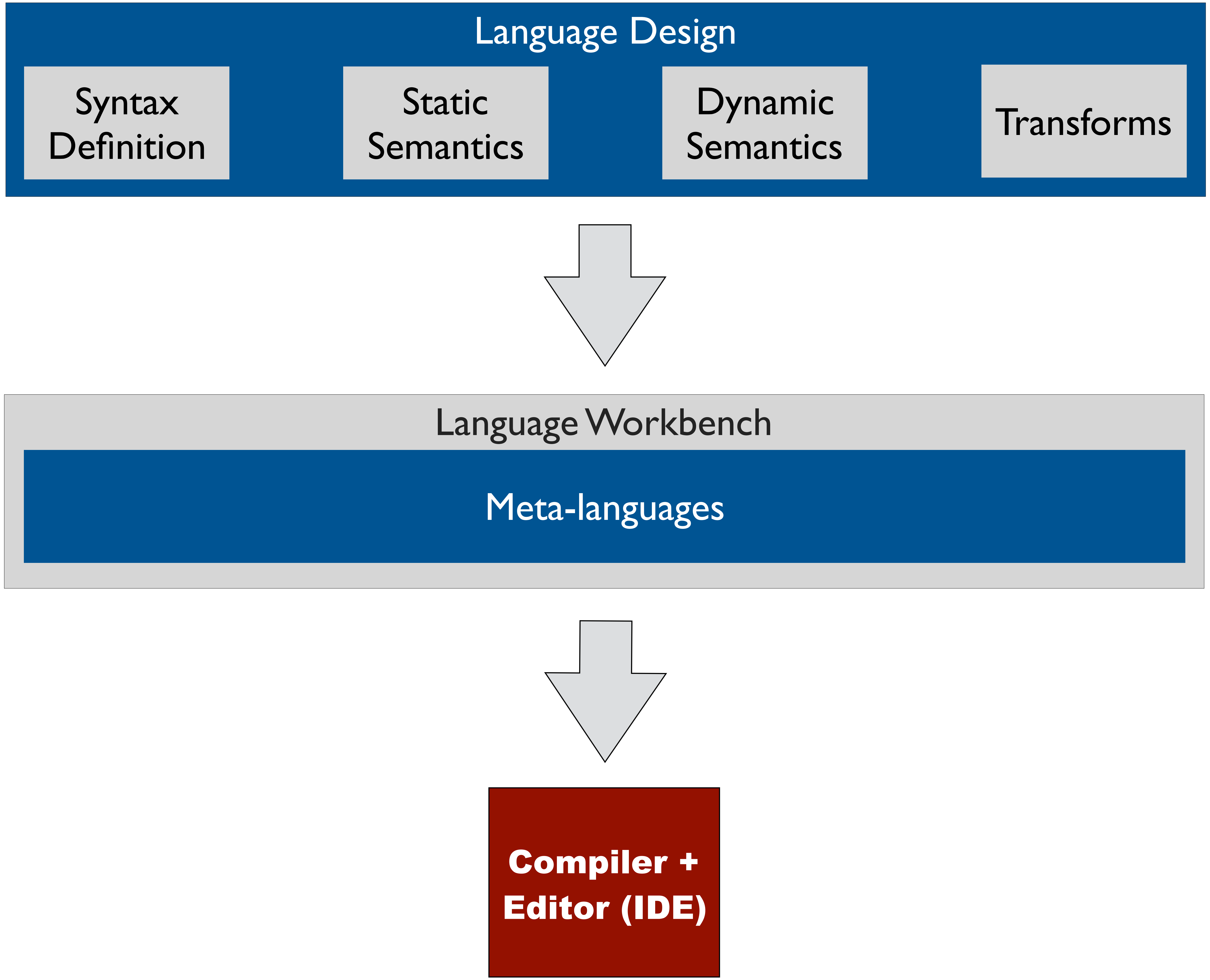
Scope Graphs and Dynamic Semantics

- Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser. *Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics.* ECOOP 2016.
- Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. *Intrinsically-Typed Definitional Interpreters for Imperative Languages.* POPL 2018.

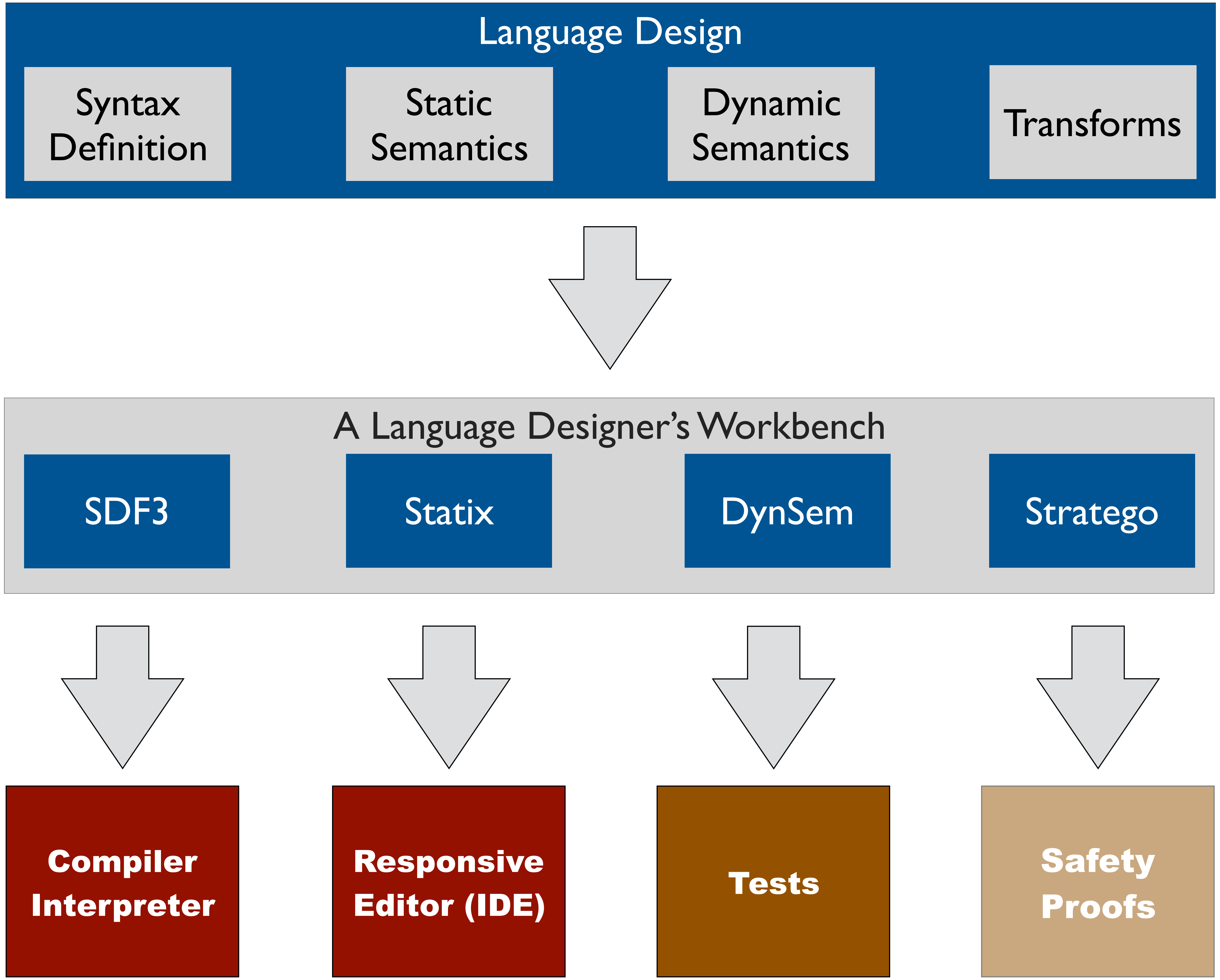
Scope Graphs and IDEs

- Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. *Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper).* ECOOP 2019

Spoofax Language Workbench



Spoofax Language Workbench



Language Testing

Questions about your language definition

- Is type system sound w.r.t. the interpreter?
- Does optimizing compiler preserve program behavior?
- Is student compiler equivalent to reference?

Answer by proving

- Finding proof can be hard, time-consuming
- Few people have the required knowledge

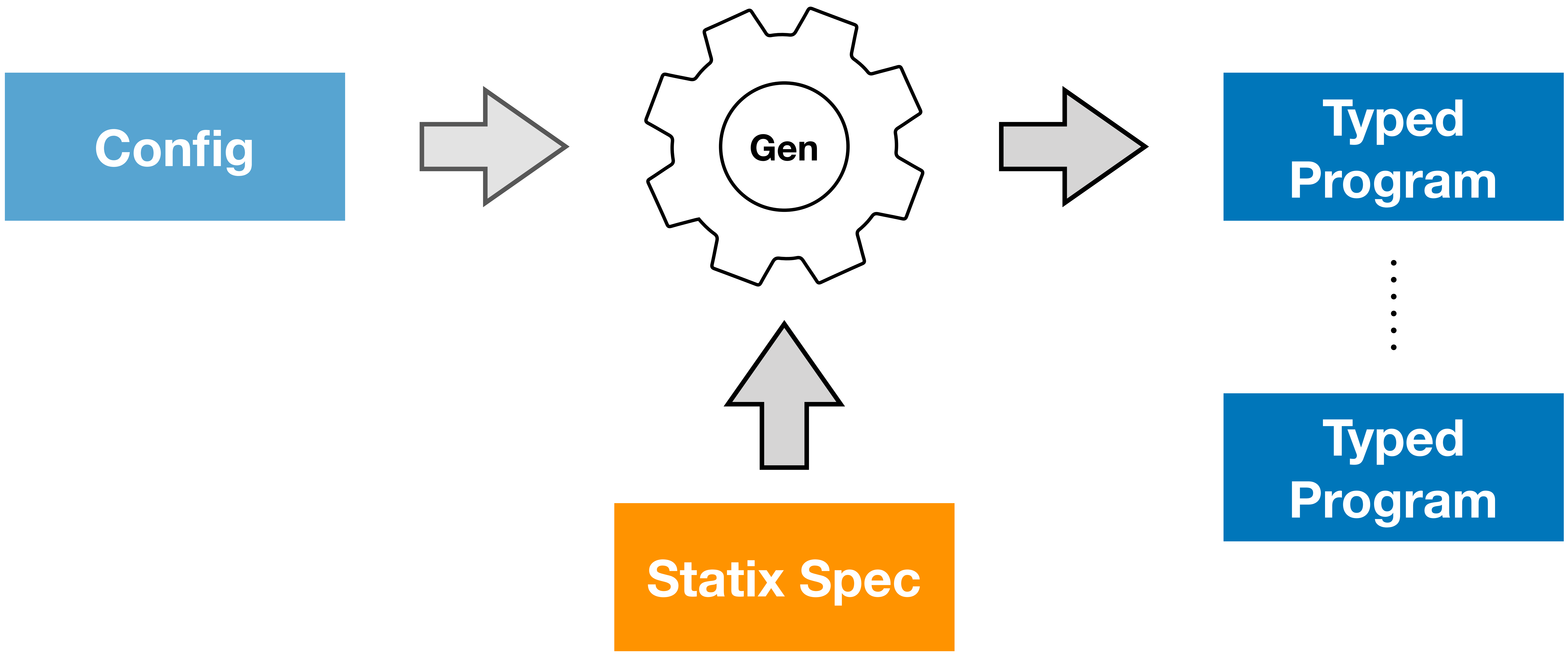
Answer by testing

- Creating good tests can be hard, time-consuming
- Feasible for more people

Answer by property-based testing

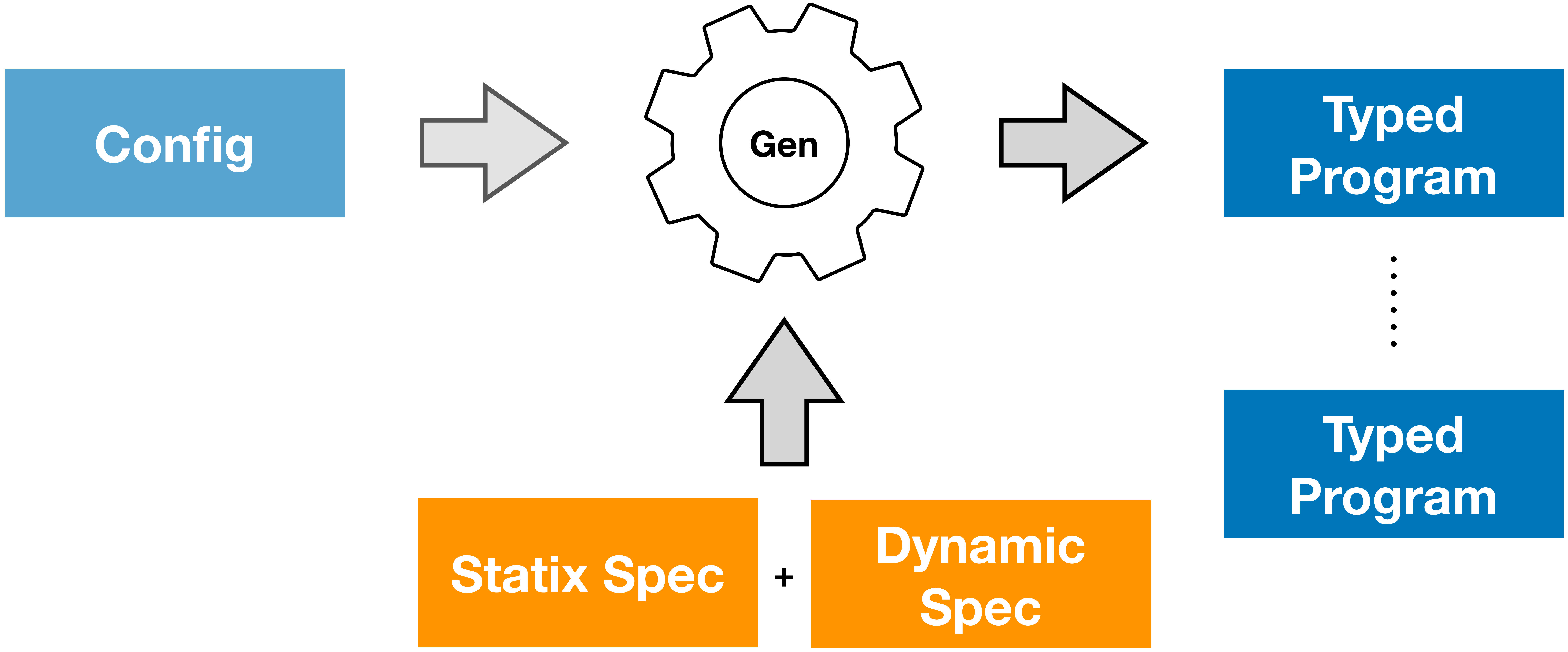
- Generate well-formed programs as test inputs
- (Semi-)automatic tool

Generating Well-Typed Programs with Statix



Do generated program have interesting execution behavior?
Do they not immediately terminate?

Generating Well-Typed Programs with Statix



Simulate execution of generated program during search
Prune uninteresting programs early

Generating Well-Typed Programs with Statix

Goal

- Random generation of well-typed programs from Statix specifications

Application

- Testing runtime behavior (compilers, interpreters)

Evaluation

- Use MiniJava language (small, but non-lexical scoping)
- Test student compilers (of which we have many) against a reference compiler

Technical Challenges

- Add search to the Statix solver (now only forward reasoning inference)
- Non-locality of names (declarations and references far apart in AST)
- Ensure interesting execution behavior in generated programs