Scopes as Types

Scan DOI

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

Problem

Declarative, executable specifications of type systems are complicated by name binding.

- Different representations for different binding patterns are bad for reuse of concepts, code, and tools.
- Executability introduces algorithmic concerns such as premise ordering and rule splitting.

Binding Pattern	Representation	Execution
Lexical	Ordered environment Name-type list	Top-down environment construction
Structural Records	Unordered fields Label-type map	Interleaving of name resolution and type checking
Modules	Module table Name-interface map	Staged module table construction and module body checking

Goal

- A generic model for name binding and name resolution.
- A way to write declarative, executable type system specifications using that model.

Approach

- Represent binding with *scope graphs*, consisting of scopes and declarations. Resolve names with queries over the graph.
- Write type systems in a constraint language, *Statix*, that supports mixing scope graph assertions and queries in rules.

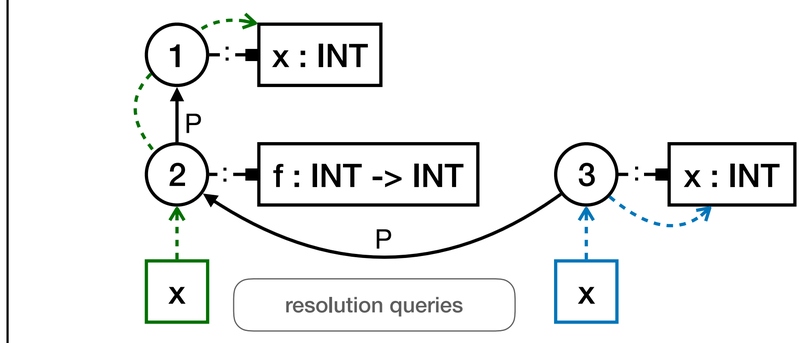
Scope Graph

- Scope graphs represent binding patterns as a graph of scopes and declarations, connected by labeled edges.
- Names are resolved by querying the graph. Query parameters (a regular expression and an order on edge labels) determine visibility and shadowing.

Statix

- Statix is a constraint language to specify type systems with syntax-directed rules.
- Rules specify assertions and queries on the (implicit) scope graph.
- Statix specifications have a declarative semantics, and are executable.

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

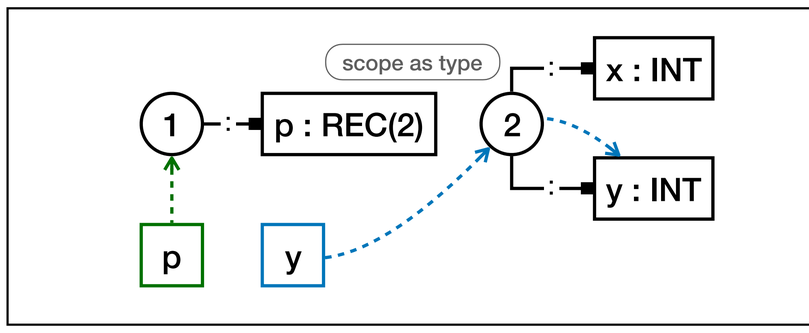


$$\text{(Let)}\ \frac{s \vdash \mathtt{e}_1 : T_1 \quad \nabla s_b \quad s_b \xrightarrow{\mathsf{P}} s \quad s_b \xrightarrow{:} \mathtt{x}_i : T_1 \quad s_b \vdash \mathtt{e}_2 : T_2}{s \vdash \mathtt{let}\ \mathtt{x}_i\ \mathtt{=}\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : T_2}$$ scope graph assertions

$$\text{(Var)}\ \frac{DECL(\mathtt{x}_i), \mathsf{P}^*, \leq_\top, <_l \vdash p : s \mapsto \mathtt{x}_j : T}{s \vdash \mathtt{x}_i : T}$$ resolution query

$$\frac{\mathtt{x}_i \simeq \mathtt{x}_j}{(\mathtt{x}_j : \mathtt{t}) \in DECL(\mathtt{x}_i)} \quad t_1 \leq_\top t_2 \quad \$ <_l \mathsf{P}$$ visibility and shadowing parameters

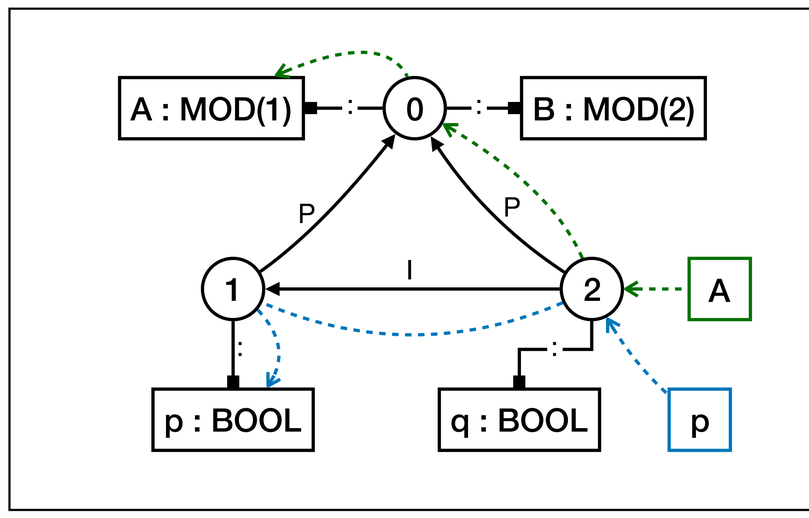
```
let p : { x:int, y:int } =
  { x = -1, y = 2 } in
p.y
```



$$\text{(Rec)}\ \frac{s \vdash \bar{\mathtt{e}} : \bar{T} \quad \nabla s_r \quad s_r \xrightarrow{:} \bar{\mathtt{x}}_i : \bar{T}}{s \vdash\ \{\ \bar{\mathtt{x}}_i\ \mathtt{=}\ \bar{\mathtt{e}}\ \} : \mathrm{REC}(s_r)}$$ scope as type

$$\text{(Fld)}\ \frac{s \vdash \mathtt{e} : \mathrm{REC}(s_r) \quad DECL(\mathtt{x}_i) \vdash p : s_r \mapsto \mathtt{x}_j : T}{s \vdash \mathtt{e.x}_i : T}$$ type-dependent query

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



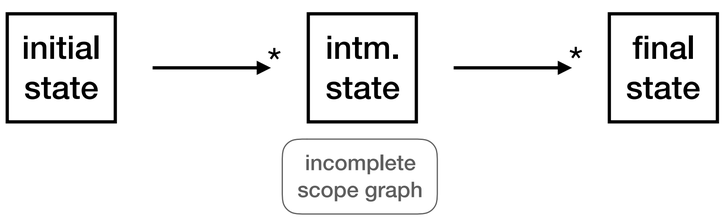
$$\text{(Mod)}\ \frac{\nabla s_m \quad s \xrightarrow{:} \mathtt{x}_i : \mathrm{MOD}(s_m) \quad s_m \xrightarrow{\mathsf{P}} s \quad s_m \vdash \bar{\mathtt{stm}}\ \mathrm{OK}}{s \vdash \mathtt{mod}\ \mathtt{x}_i\ \{\ \bar{\mathtt{stm}}\ \}\ \mathrm{OK}}$$ scope graph assertions, checking subterms, and queries are mixed in rules

$$\text{(Imp)}\ \frac{DECL(\mathtt{x}_i), \mathsf{P}^*, \leq_\top, <_l \vdash p : s \mapsto \mathtt{x}_j : \mathrm{MOD}(s_m) \quad s \xrightarrow{\mathsf{I}} s_m}{s \vdash \mathtt{import}\ \mathtt{x}_i\ \mathrm{OK}}$$

$$\text{(Var')}\ \frac{DECL(\mathtt{x}_i), \mathsf{P}^*\mathsf{I}^*, \leq_\top, <'_l \vdash p : s \mapsto \mathtt{x}_j : T}{s \vdash \mathtt{x}_i : T} \quad \$ <'_l \mathsf{P} \quad \$ <'_l \mathsf{I} \quad \mathsf{I} <'_l \mathsf{P}$$ variable resolution with import edges

Executing Statix Specifications

Specifications are executed by rewriting a constraint set and a solution:



- Constraints are solved by unification, building the scope graph, querying the scope graph, and rule-based simplification.
- Intermediate scope graphs may be *incomplete*, because of remaining scope graph assertions in the constraint set.
- Resolving queries in an incomplete scope graph is essential to support type-dependent name resolution or binding in types.

Can we soundly resolve queries in an incomplete scope graph?

- A query result on an intermediate graph is sound if it also holds in the final graph.
- This is true if remaining constraints do not add data that shadows the query result.
- We (over)approximate which labeled edges may be added to the scopes in the intermediate graph.
- This approximation uses *static* rule information and *dynamic* information on the remaining constraints.
- A dynamic check ensures that scope graph queries are *delayed* if an invalidating graph extension may occur.