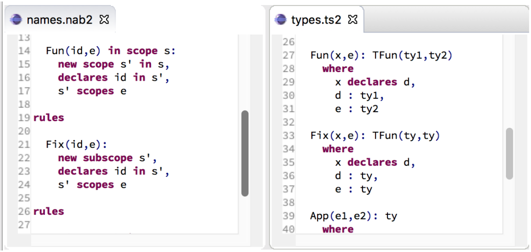
A Constraint Language for Static Semantic Analysis Based on Scope Graphs

Hendrik van Antwerpen, Pierre Neron, Andrew Tolmach, Eelco Visser, Guido Wachsmuth

PEPM, January 19, 2016

Motivation



```

  Fun(id,e) in scope s:
    new scope s' in s,
    declares id in s',
    s' scopes e

rules

  Fix(id,e):
    new subscope s',
    declares id in s',
    s' scopes e

rules

```

```

  Fun(x,e): TFun(ty1,ty2)
    where
      x declares d,
      d : ty1,
      e : ty2

  Fix(x,e): TFun(ty,ty)
    where
      x declares d,
      d : ty,
      e : ty

  App(e1,e2): ty
    where
```

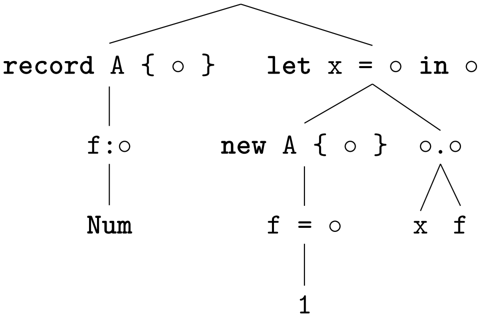
- Language engineering with language workbench
- Language-dependent specification, using language-independent framework
- Separate language aspects
- Today: framework for static semantic analysis, based on
  - scope graphs
  - constraints

Example

Program

```
record A {
  f:Num
}

let
  x = new A {
    f = 1
  }
in x.f
```

Abstract Syntax Tree



`record A { ○ }`

`let x = ○ in ○`

`f:○`

`Num`

`new A { ○ }`

`○.○`

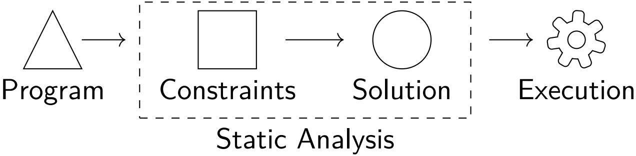
`f = ○`

`1`

`x`

`f`

Constraint-based Type Analysis



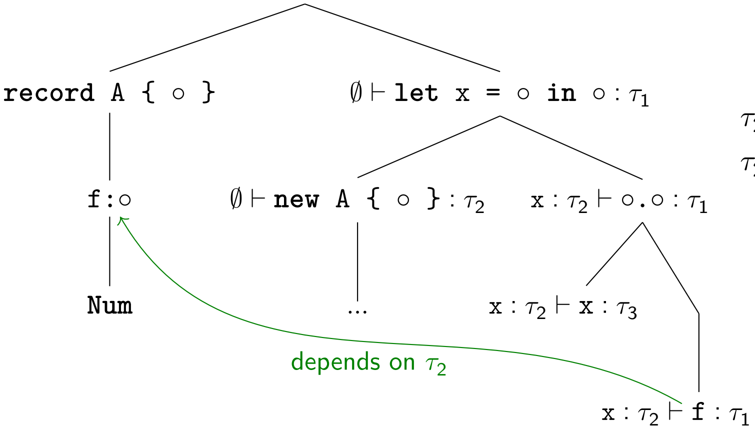
- Language-dependent constraint generation
- Language-independent constraint solving
- Freedom in order of solving
- Potential for inference

$$\Gamma \vdash e : t \longrightarrow C$$

Constraint-based Type Analysis

Generation

Constraints



`record A { ∘ }`

$\emptyset \vdash$ `let x = ∘ in ∘` $: \tau_1$

`f:∘`

$\emptyset \vdash$ `new A { ∘ }` $: \tau_2$

`x` $: \tau_2 \vdash \circ$`.`$\circ : \tau_1$

`Num`

…

`x` $: \tau_2 \vdash$ `x` $: \tau_3$

depends on τ_2

`x` $: \tau_2 \vdash$ `f` $: \tau_1$

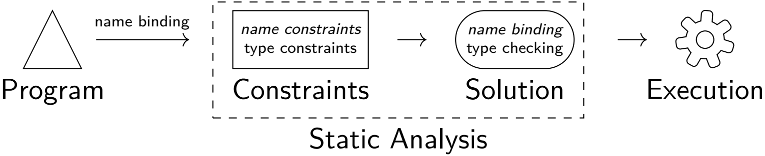
$\vdots$

$\tau_2 \overset{?}{=} \texttt{Rec(A)}$

$\tau_2 \overset{?}{=} \tau_3$

Requires additional, ad hoc data structures (e.g. class table)

Constraint-based Name and Type Analysis



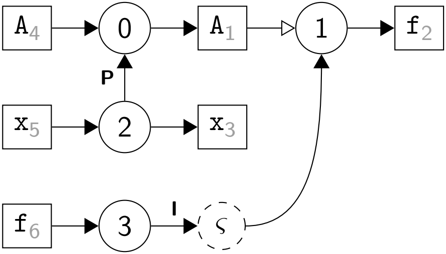
- Existing approaches: name binding during constraint generation
- Goal: name binding during constraint solving
- Use scope graphs for language independent name resolution

Intermezzo: Scope Graphs

Program

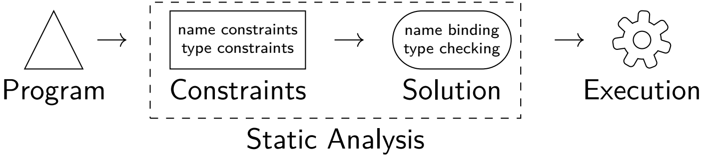
```
record A1 {
  f2:Num
}

let
  x3 = new A4 {
    ...
  }
in x5.f6
```

Scope Graph



Introduced by Neron e.a., *A Theory of Name Resolution*, ESOP, 2015

Constraint-based Name and Type Analysis

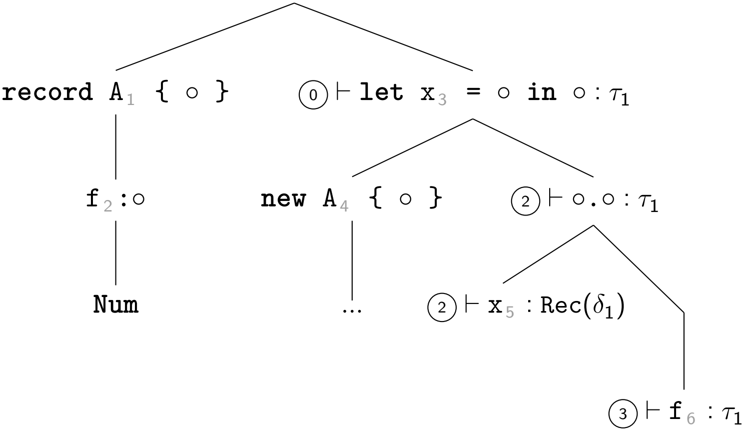


$$\textcircled{s} \vdash e : t \longrightarrow C$$

Constraint-based Name and Type Analysis

Generation

Constraints



record $\mathtt{A}_{1}$ { ○ }

⓪ $\vdash$ **let** $\mathtt{x}_{3}$ = ○ **in** ○ : τ_1

$\mathtt{f}_{2}$: ○

new $\mathtt{A}_{4}$ { ○ }

② $\vdash$ ○.○ : τ_1

Num

…

② $\vdash$ $\mathtt{x}_{5}$: Rec(δ_1)

③ $\vdash$ $\mathtt{f}_{6}$: τ_1

⋮

$\delta_1 \rightsquigarrow \varsigma$

③ $\xrightarrow{\shortmid} \varsigma$

$\mathtt{f}_{6} \longrightarrow$ ③

$\mathtt{f}_{6} \mapsto \delta_2$

$\delta_2 : \tau_1$

Constraint-based Name and Type Analysis

- Constraints for
  - Scope graph construction
  - Name resolution
  - Type checking
- Formal semantics for solution $\mathcal{G}, \psi, \varphi \models C$
  - Scope graph $\mathcal{G}$
  - Type environment ψ
  - Substitution φ
- Resolution algorithm $\textsc{Solve}(C) = \langle \mathcal{G}, \psi, \varphi \rangle$

Constraint-based Name and Type Analysis

Program

```
record A_1 {
  f_2 :Num
}

let
  x_3 = new A_4 {
    ...
  }
in x_5 .f_6
```

Constraints

$$\vdots$$

$\mathtt{A}_1 \rightsquigarrow \varsigma$

$\textcircled{3} \xrightarrow{I} \varsigma$

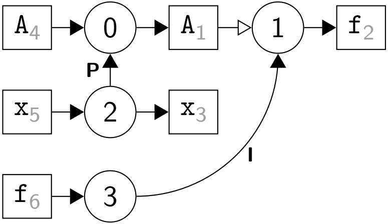
$\mathtt{f}_6 \longrightarrow \textcircled{3}$

$\mathtt{f}_6 \mapsto \delta_2$

$\mathtt{f}_2 : \tau_1$

Solution

Scope graph



Type env

$\mathtt{f}_2 : \mathtt{Num}$

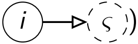
$\mathtt{x}_3 : \mathtt{Rec}(\mathtt{A}_1)$

Substitution

$\delta_1 \mapsto \mathtt{A}_1$ $\quad \delta_2 \mapsto \mathtt{f}_2$

$\varsigma \mapsto \textcircled{1}$ $\quad \tau_1 \mapsto \mathtt{Num}$

Constraint-based Name and Type Analysis

- Summary
  - Constraints to express name and type analysis
  - Language-specific constraint generation
  - Language-independent constraint solver
    - (modulo type vocabulary)
- Preliminary validation
  - Functional language: PCF
  - Object-oriented language: Featherweight Java
- Solve algorithm
  - Terminating and sound, $\textsc{Solve}(C) = \Delta \implies \Delta \models C$
  - Incomplete (conjecture: complete without )
  - Prototype implementation

More scope graph contributions in paper

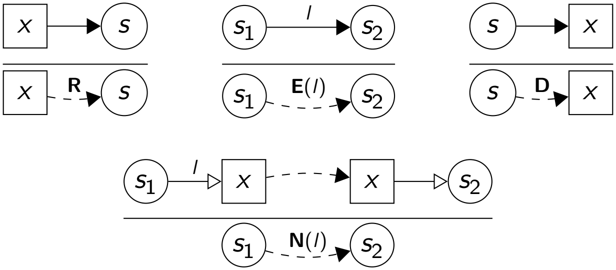
- Generalized parents and imports to arbitrary labels
- Parametrized name resolution algorithm
- Name disambiguation constraints
  - Uniqueness (e.g. prevent duplicate record definitions)
  - Inclusion (e.g. ensure all fields are initialized)

Future Work

- Scope graphs
    - High-level specification language based on scope graphs
    - Name resolution sensitive program transformations
    - Relate static scope graphs to dynamic semantics
- Constraint language
    - Support more advanced types, e.g. polymorphism
    - Factor out constraint domain X, cf. HM(X) and OutsideIn(X)
    - Constraint solver performance
- Mechanized language meta-theory based on this framework

Resolution Calculus

Reachability

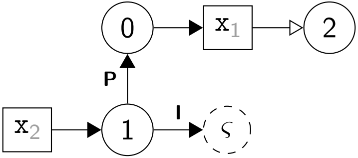


Visibility

Labels	$\mathcal{L}$	e.g.	$\{\mathbf{P}, \mathbf{I}\}$	
Ordering	$<$	e.g.	$\mathbf{I} < \mathbf{P}$	
Well-formedness	$\mathrm{WF}(p)$	e.g.	$\mathbf{P}^* \cdot \mathbf{I}^* \cdot \mathbf{D}$	(transitive)
		or	$\mathbf{P}^* \cdot \mathbf{I}^? \cdot \mathbf{D}$	(non-transitive)

Incompleteness Example

Scope graph



Visibility: **I** $<$ **P**

Constraints

$$\mathtt{x}_2{}^{\mathsf{R}} \mapsto \delta$$

$$\delta \rightsquigarrow \varsigma$$

Solution

$$\delta \mapsto \mathtt{x}_1{}^{\mathsf{D}}$$

$$\varsigma \mapsto ②$$

Type-dependent Path Ordering

```
class A {}
class B extends A {}

class X {
  void m(B b) {}
}
class Y extends X {
  void m(A a) {}
}

// new Y().m(new B())
```