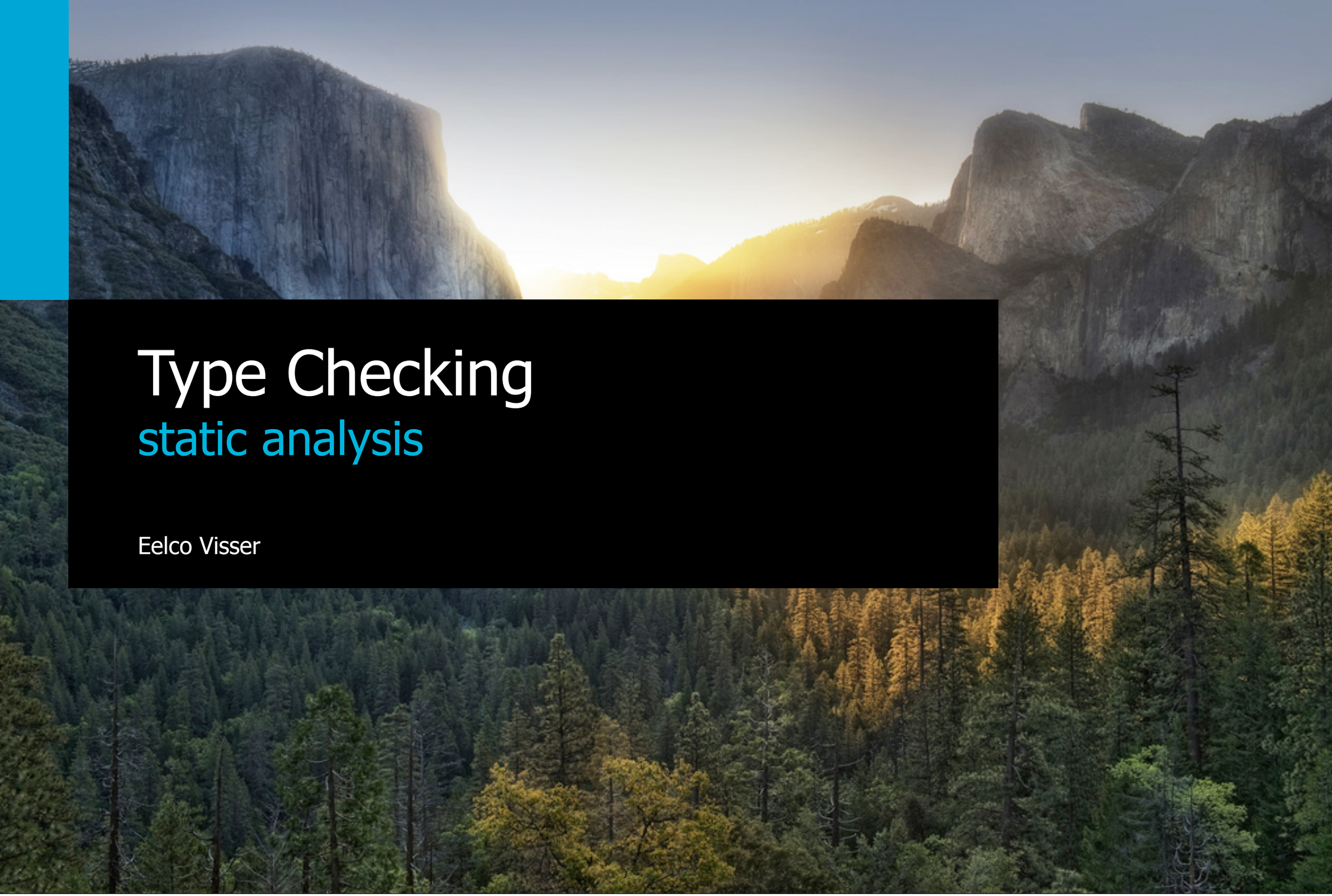
Type Checking
static analysis

Eelco Visser



Name Resolution

A Theory of Name Resolution

Pierre Neron[1], Andrew Tolmach[2], Eelco Visser[1], and Guido Wachsmuth[1]

[1)] Delft University of Technology, The Netherlands,
{p.j.m.neron, e.visser, g.wachsmuth}@tudelft.nl,
[2)] Portland State University, Portland, OR, USA
tolmach@pdx.edu

Abstract. We describe a language-independent theory for name binding and resolution, suitable for programming languages with complex scoping rules including both lexical scoping and modules. We formulate name resolution as a two-stage problem. First a language-independent scope graph is constructed using language-specific rules from an abstract syntax tree. Then references in the scope graph are resolved to corresponding declarations using a language-independent resolution process. We introduce a resolution calculus as a concise, declarative, and language-independent specification of name resolution. We develop a resolution algorithm that is sound and complete with respect to the calculus. Based on the resolution calculus we develop language-independent definitions of α-equivalence and rename refactoring. We illustrate the approach using a small example language with modules. In addition, we show how our approach provides a model for a range of name binding patterns in existing languages.

1 Introduction

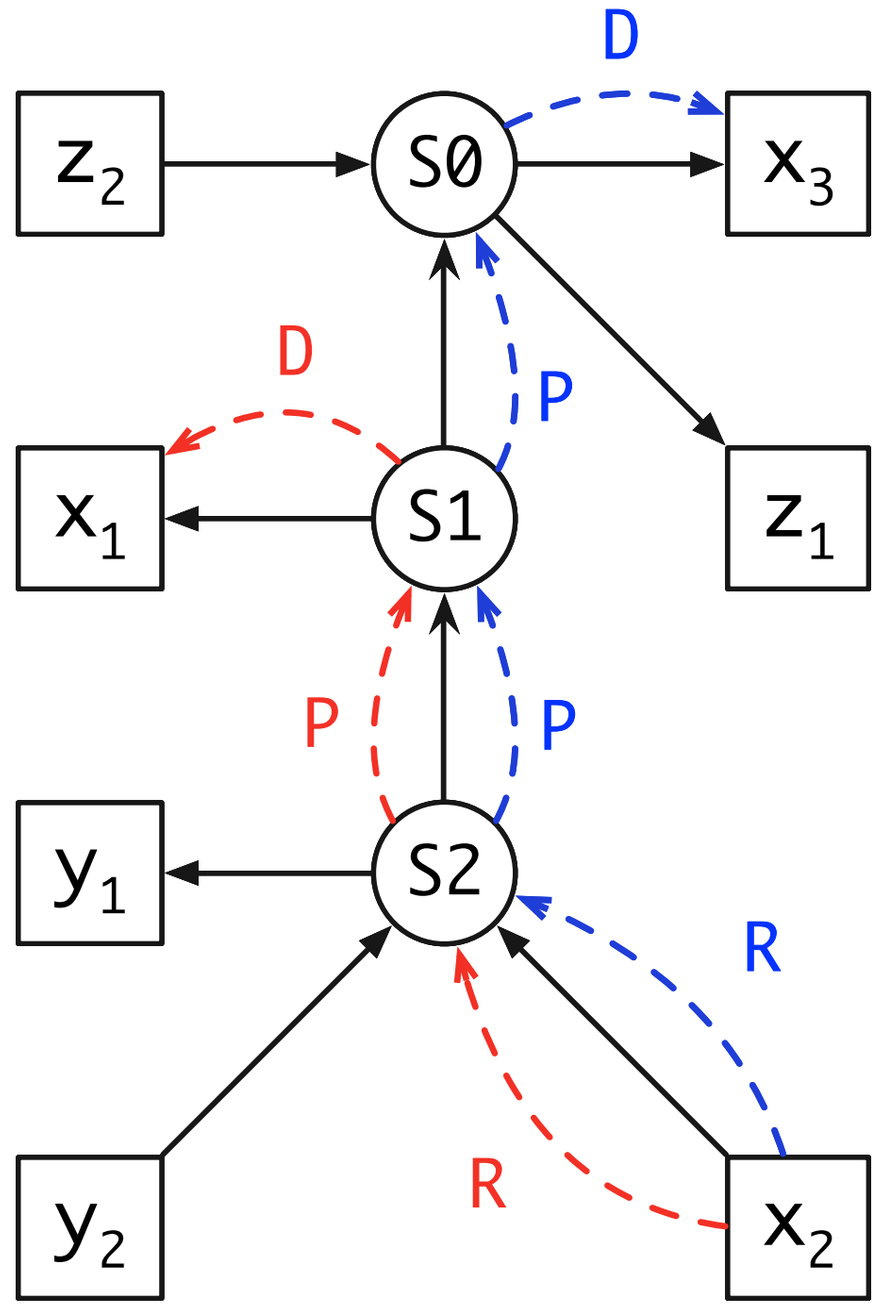
Naming is a pervasive concern in the design and implementation of programming languages. Names identify *declarations* of program entities (variables, functions, types, modules, etc.) and allow these entities to be *referenced* from other parts of the program. Name *resolution* associates each reference to its intended declaration(s), according to the semantics of the language. Name resolution underlies most operations on languages and programs, including static checking, translation, mechanized description of semantics, and provision of editor services in IDEs. Resolution is often complicated, because it cuts across the local inductive structure of programs (as described by an abstract syntax tree). For example, the name introduced by a `let` node in an ML AST may be referenced by an arbitrarily distant child node. Languages with explicit name spaces lead to further complexity; for example, resolving a qualified reference in Java requires first resolving the class or package name to a context, and then resolving the member name within that context. But despite this diversity, it is intuitively clear that the basic concepts of resolution reappear in similar form across a broad range of lexically-scoped languages.

In practice, the name resolution rules of real programming languages are usually described using *ad hoc* and informal mechanisms. Even when a language *is* formalized, its resolution rules are typically encoded as part of static

[ESOP 2015]

Shadowing

```
def x3 = z2 5 7                    S0

def z1 =
  fun x1 {                         S1
    fun y1 {                       S2
      x2 + y2
    }
  }
```



$$\frac{}{D < P.p}$$

$$\frac{p < p'}{s.p < s.p'}$$

$$R.P.D < R.P.P.D$$

Blocks in Java

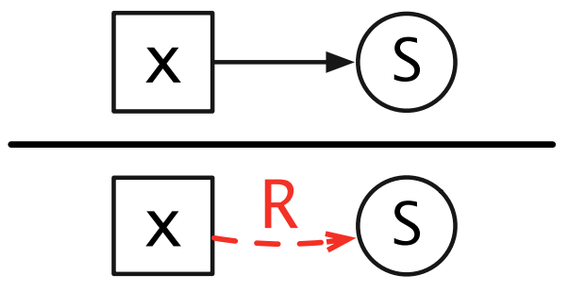
```
class Foo {
  void foo() {
    int x = 1;
    {
      int y = 2;
    }
    x = y;
  }
}
```

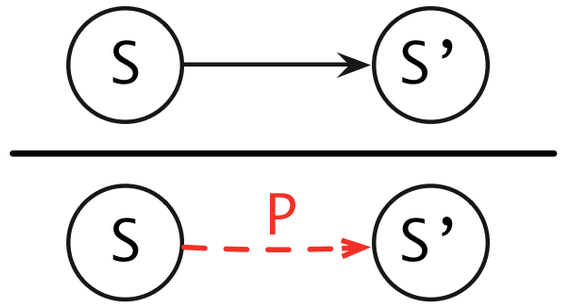
What is the scope graph for this program?

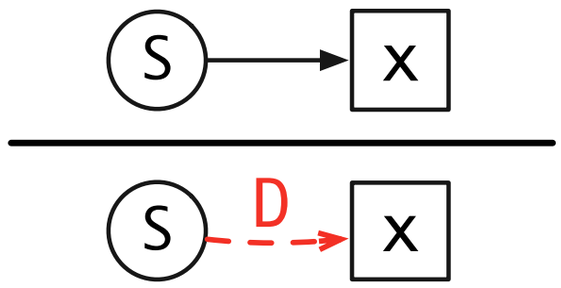
Is the y declaration visible to the y reference?

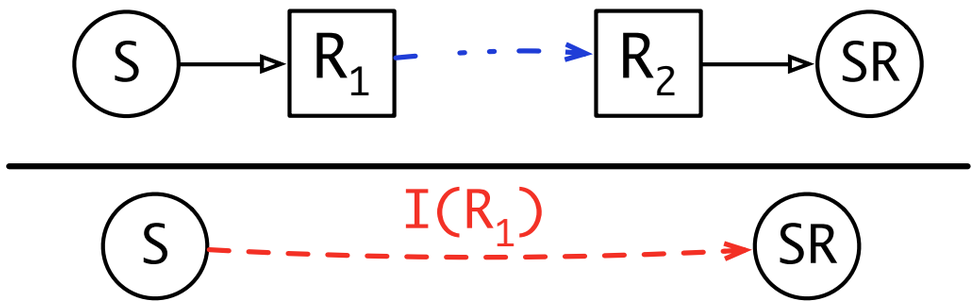
A Calculus for Name Resolution

Reachability









Well formed path: `R.P*.I(_)*.D`

Visibility

$$\frac{}{D < P.p} \qquad \frac{}{I(\_).p' < P.p}$$

$$\frac{p < p'}{s.p < s.p'} \qquad \frac{}{D < I(\_).p'}$$

Alpha Equivalence

Language-independent α-equivalence

Program similarity

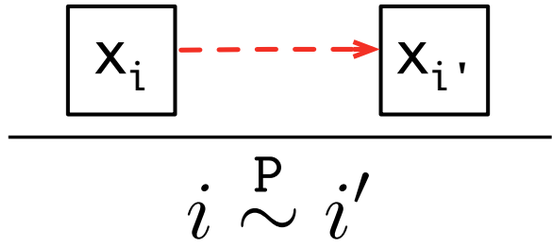
`P` $\simeq$ `P'` if have same AST ignoring identifier names

Language-independent α-equivalence

Program similarity

$\texttt{P} \simeq \texttt{P'}$ if have same AST ignoring identifier names

Position equivalence



$$\frac{i' \overset{P}{\sim} i}{i \overset{P}{\sim} i'}$$

$$\frac{i \overset{P}{\sim} i' \quad i' \overset{P}{\sim} i''}{i \overset{P}{\sim} i''}$$

$$\frac{}{i \overset{P}{\sim} i}$$

Language-independent α-equivalence

Program similarity

$\texttt{P} \simeq \texttt{P'}$ if have same AST ignoring identifier names

Position equivalence

$$\frac{}{i \overset{\texttt{P}}{\simeq} i'} \qquad \frac{i' \overset{\texttt{P}}{\simeq} i}{i \overset{\texttt{P}}{\simeq} i'} \qquad \frac{i \overset{\texttt{P}}{\simeq} i' \quad i' \overset{\texttt{P}}{\simeq} i''}{i \overset{\texttt{P}}{\simeq} i''} \qquad \frac{}{i \overset{\texttt{P}}{\simeq} i}$$

Alpha equivalence

$$\texttt{P1} \overset{\alpha}{\approx} \texttt{P2} \triangleq \texttt{P1} \simeq \texttt{P2} \wedge \forall\ e\ e',\ e \overset{\texttt{P1}}{\simeq} e' \Leftrightarrow e \overset{\texttt{P2}}{\simeq} e'$$

(with some further details about free variables)

Preserving ambiguity

```
module A1 {
  def x2 := 1
}
module B3 {
  def x4 := 2
}
module C5 {
  import A6 B7;
  def y8 := x9
}
module D10 {
  import A11;
  def y12 := x13
}
module E14 {
  import B15;
  def y16 := x17
}
```

P1

```
module AA1 {
  def z2 := 1
}
module BB3 {
  def z4 := 2
}
module C5 {
  import AA6 BB7;
  def s8 := z9
}
module D10 {
  import AA11;
  def u12 := z13
}
module E14 {
  import BB15;
  def v16 := z17
}
```

P2

```
module A1 {
  def z2 := 1
}
module B3 {
  def x4 := 2
}
module C5 {
  import A6 B7;
  def y8 := z9
}
module D10 {
  import A11;
  def y12 := z13
}
module E14 {
  import B15;
  def y16 := x17
}
```

P3

$P1 \overset{\alpha}{\approx} P2$ $\quad P2 \not\overset{\alpha}{\approx} P3$

Names and Types

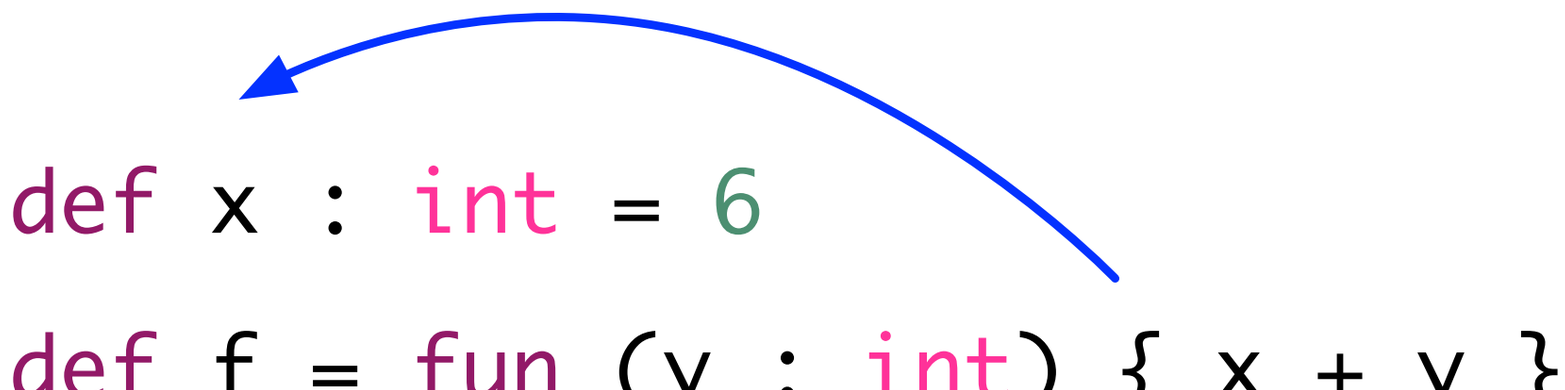
Types from Declaration

```
def x : int = 6

def f = fun (y : int) { x + y }
```

Static type-checking (or inference) is one obvious client for name resolution

In many cases, we can perform resolution **before** doing type analysis

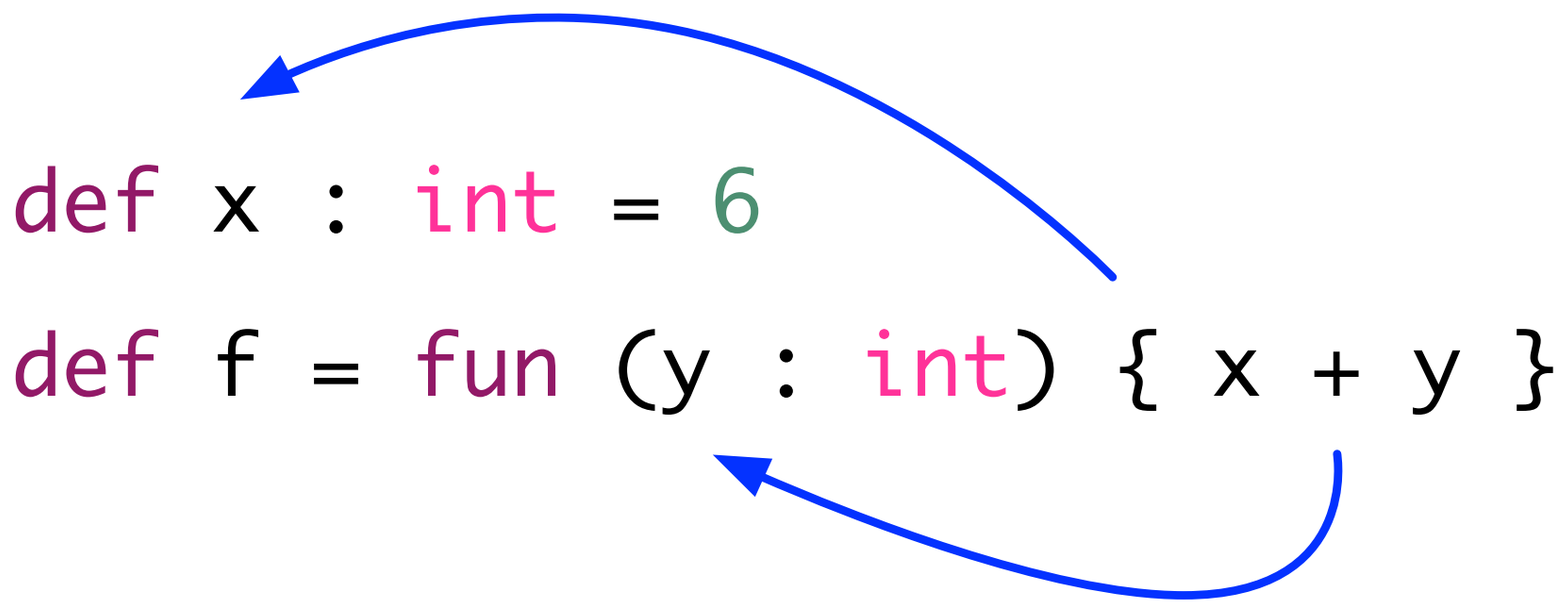
Types from Declaration



Static type-checking (or inference) is one obvious client for name resolution

In many cases, we can perform resolution **before** doing type analysis

Types from Declaration



Static type-checking (or inference) is one obvious client for name resolution

In many cases, we can perform resolution **before** doing type analysis

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A₁ { x₁ : int }
record B₁ { a₁ : A₂ ; x₂ : bool}

def z₁ : B₂ = ...

def y₁ = z₂.x₃

def y₂ = z₃.a₂.x₄
```

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Type-Dependent Name Resolution

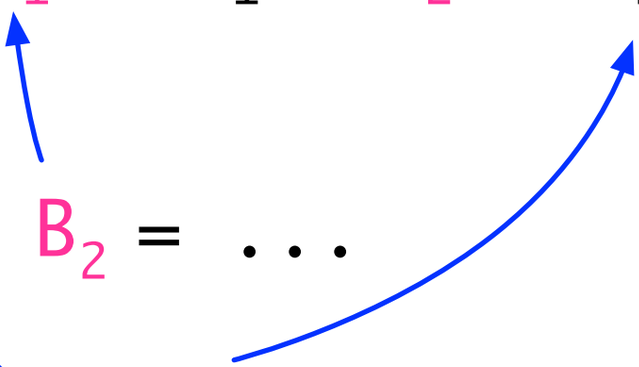
But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



Type-Dependent Name Resolution

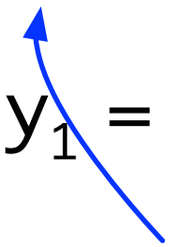
But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

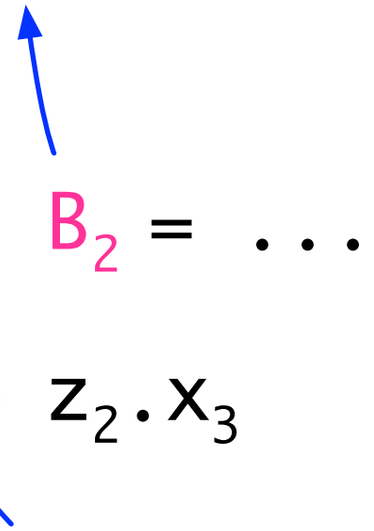
```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



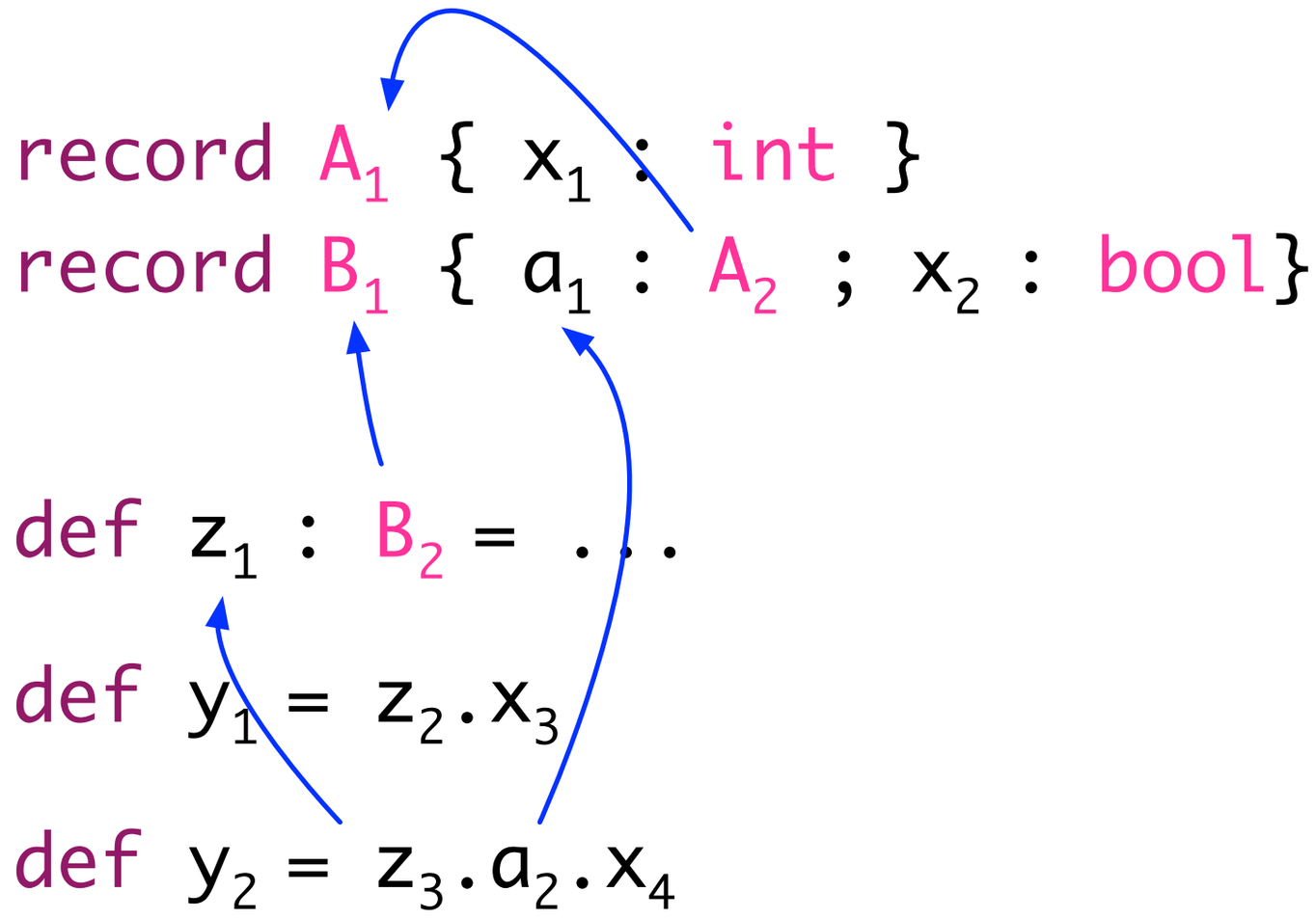
Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution

```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

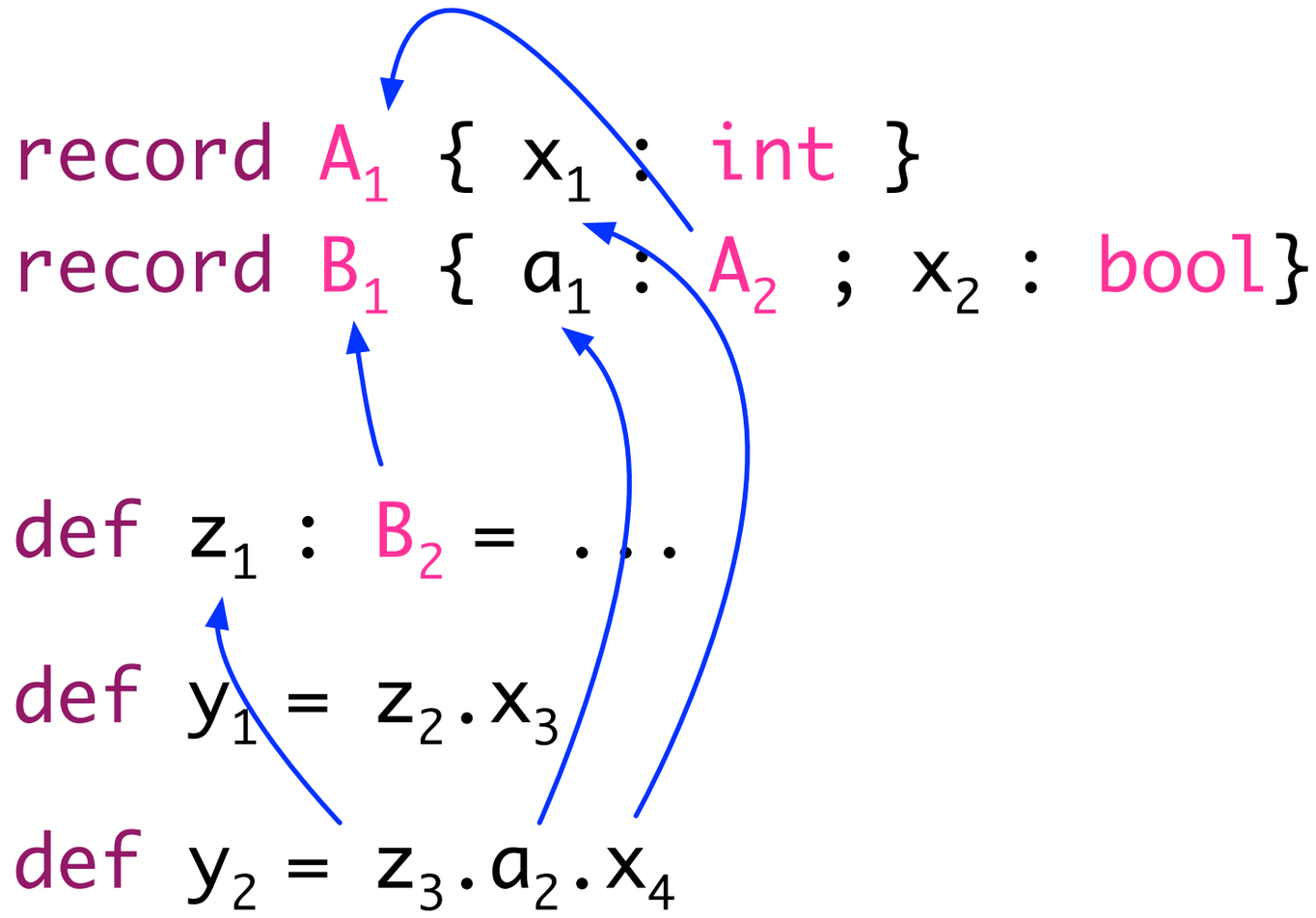
Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution



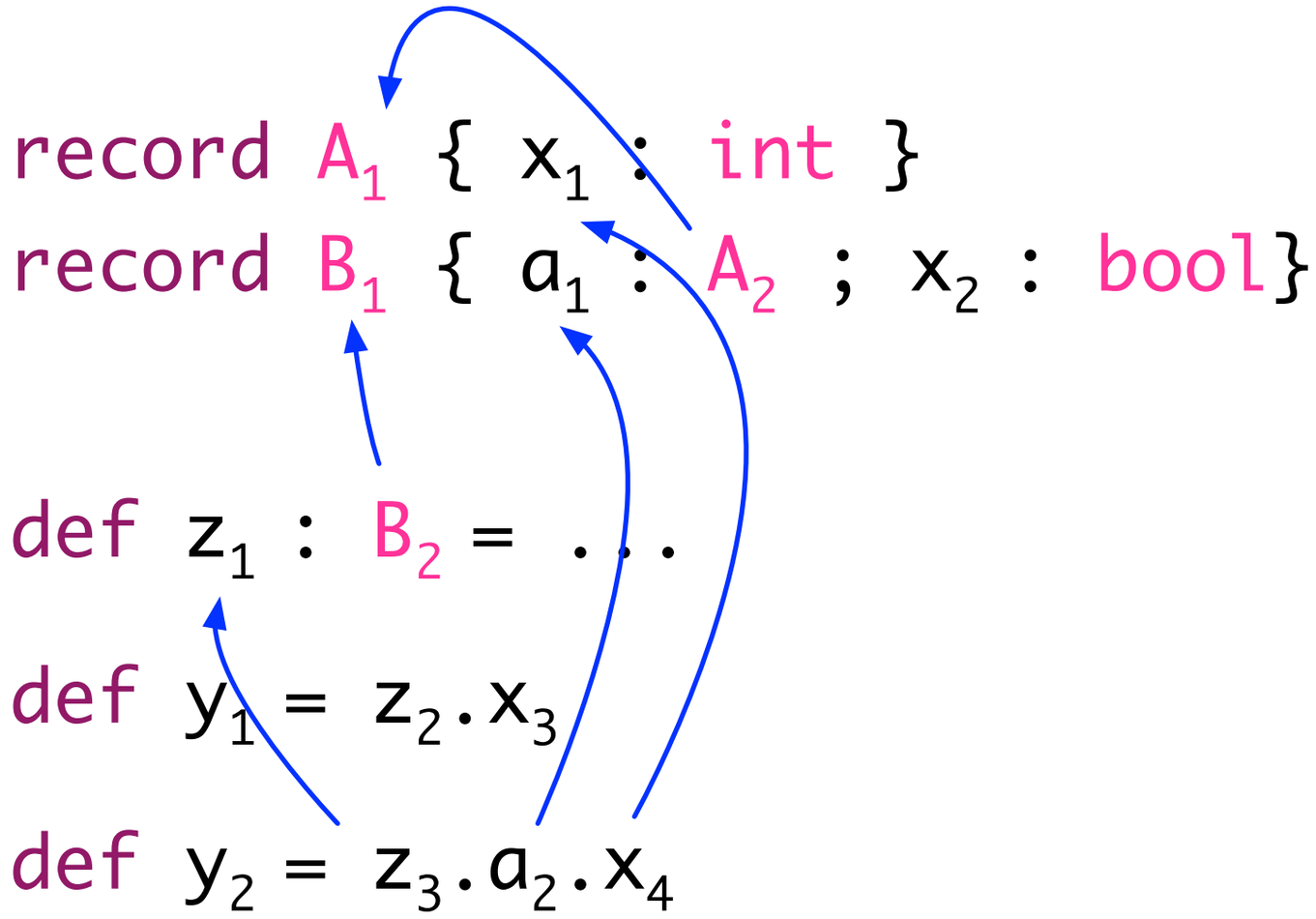
Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution



Type-Dependent Name Resolution

But sometimes we need types **before** we can do name resolution



Our approach: interleave **partial** name resolution with type resolution (also using constraints)

See PEPM 2016 paper / talk

Constraint Language

A Constraint Language for Static Semantic Analysis based on Scope Graphs

Hendrik van Antwerpen
Delft University of Technology
h.vanantwerpen@student.tudelft.nl

Pierre Néron
Delft University of Technology
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University
tolmach@pdx.edu

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Abstract

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

In recent work we show how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one, none, or many declarations. A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

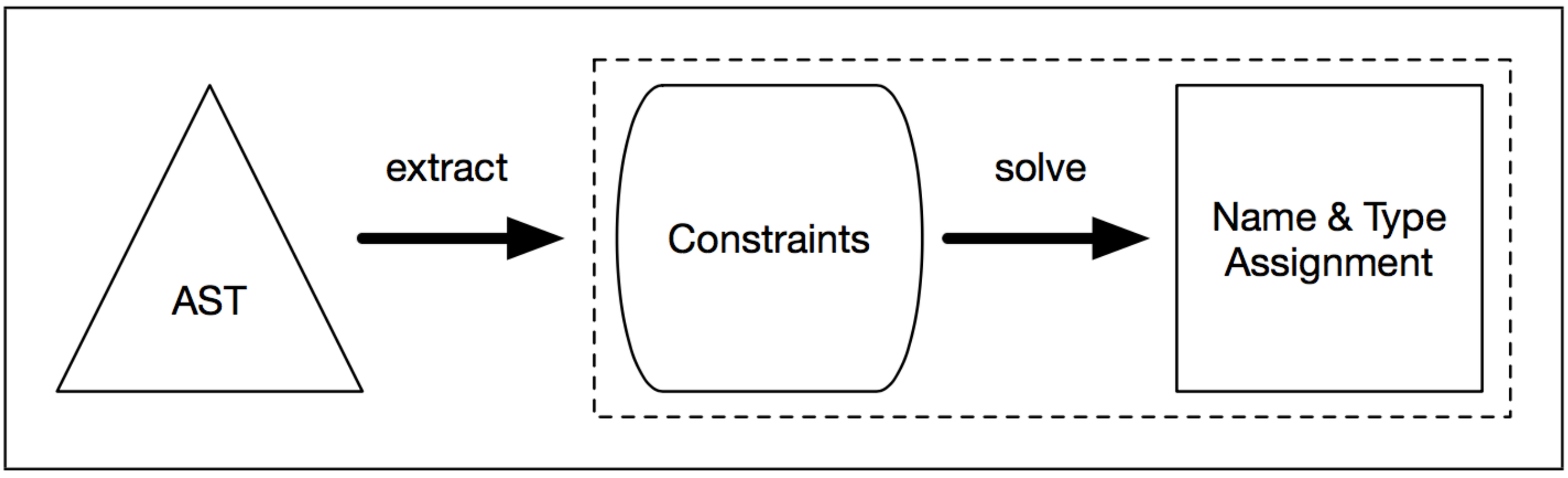
In this paper, we refine and extend the scope graph framework of [14] to construct a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checking step–would work. But the full story is more complicated, because sometimes name resolution also depends on type resolution. For example, in a language that uses dot notation for object field projection, determining the resolution of `x` in the expression `r.x` requires first determining the object type of `r`, which in turn requires name resolution again. Thus, our framework requires a unified mechanism for expressing and solving arbitrarily interdependent naming and typing resolution problems.

To address this challenge, we base our framework on a language of *constraints*. Term equality constraints are a standard choice for describing type inference problems while abstracting away from the details of an AST in a particular language. Adopting constraints to describe both typing and scoping requirements has the advantage of uniform notation, and, more importantly, provides a clean way to combine naming and typing problems. In particular, we extend our previous work to support *incomplete* scope graphs, which correspond to constraint sets with (as yet) unresolved variables.

Our new framework continues to satisfy the criteria outlined above. (i) The resolution calculus and standard term equality constraint theory provide a solid language-independent theory for name and type resolution. (ii) Our framework supports type checking and inference for statically typed, monomorphic languages with user-defined types, and can also express uniqueness and completeness requirements on declarations and initializers. The framework inherits from scope graphs the ability to model a broad range of binding patterns, including many variants of lexical scoping, records, and modules. (iii) The constraint language has a declarative semantics given by a constraint satisfaction relation, which employs the resolution calculus to define the semantics of name resolution relative to a scope graph. We define a constraint resolution algorithm based on our previous name resolution algorithm,

[PEPM 2016]

Architecture



Syntax of Constraints

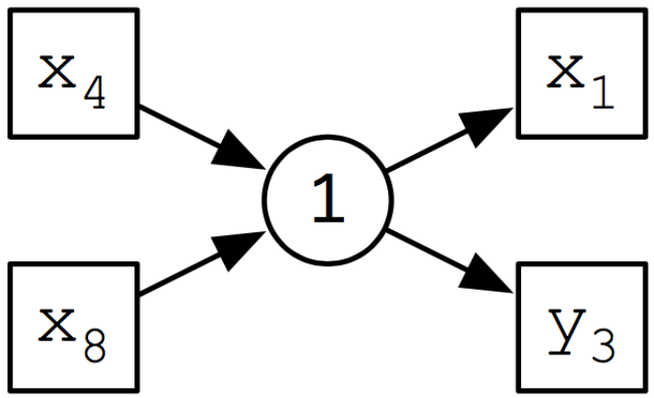
$$
\begin{array}{rcl}
\mathrm{C} & := & \mathrm{C}^{G} \mid \mathrm{C}^{Ty} \mid \mathrm{C}^{Res} \mid \mathrm{C} \wedge \mathrm{C} \mid \mathsf{True} \\
\mathrm{C}^{G} & := & \mathrm{R} \longrightarrow \mathrm{S} \mid \mathrm{S} \longrightarrow \mathrm{D} \mid \mathrm{S} \xrightarrow{l} \mathrm{S} \mid \mathrm{D} \longrightarrow\!\!\triangleright \mathrm{S} \mid \mathrm{S} \xrightarrow{l}\!\!\triangleright \mathrm{R} \\
\mathrm{C}^{Res} & := & \mathrm{R} \mapsto \mathrm{D} \mid \mathrm{D} \rightsquigarrow \mathrm{S} \mid !\mathrm{N} \mid \mathrm{N} \subseteq\!\!\!\sim\ \mathrm{N} \\
\mathrm{C}^{Ty} & := & \mathrm{T} \equiv \mathrm{T} \mid \mathrm{D} : \mathrm{T} \\
\mathrm{D} & := & \delta \mid x_i^{\mathbf{D}} \\
\mathrm{R} & := & x_i^{\mathbf{R}} \\
\mathrm{S} & := & \varsigma \mid n \\
\mathrm{T} & := & \tau \mid c(\mathrm{T}, ..., \mathrm{T}) \text{ with } c \in C_{\mathcal{T}} \\
\mathrm{N} & := & \overline{\mathcal{D}}(\mathrm{S}) \mid \overline{\mathcal{R}}(\mathrm{S}) \mid \overline{\mathcal{V}}(\mathrm{S})
\end{array}
$$

LMR: Language with Modules and Records

$$
\begin{array}{rcl}
prog & = & decl^* \\
decl & = & \texttt{module}\ id\ \{\, decl^*\,\} \\
 & | & \texttt{import}\ id \\
 & | & \texttt{def}\ bind \\
 & | & \texttt{record}\ id\ \{\, fdecl^*\,\} \\
fdecl & = & id\ \texttt{:}\ ty \\
ty & = & \texttt{Int} \\
 & | & \texttt{Bool} \\
 & | & id \\
 & | & ty \to ty \\
exp & = & int \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & | & id \\
 & | & exp \oplus exp \\
 & | & \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp \\
 & | & \texttt{fun}\ (\ id\ \texttt{:}\ ty\ )\ \{\, exp\,\} \\
 & | & exp\ exp \\
 & | & \texttt{letrec}\ tbind\ \texttt{in}\ exp \\
 & | & \texttt{new}\ id\ \{\, fbind^*\,\} \\
 & | & \texttt{with}\ exp\ \texttt{do}\ exp \\
 & | & exp\ \texttt{.}\ id \\
bind & = & id = exp \\
 & | & tbind \\
tbind & = & id\ \texttt{:}\ ty = exp \\
fbind & = & id = exp
\end{array}
$$

```
def x1 = 12
def y3 = (if (x4 == 05)6 then 37 else x8)9
```

Scope graph constraints



Resolution constraints

$$\mathrm{x}_4^{\mathbf{R}} \mapsto \delta_4$$
$$\mathrm{x}_8^{\mathbf{R}} \mapsto \delta_8$$
$$!\overline{\mathcal{D}}(1)$$

Typing constraints

$$\mathrm{x}_1^{\mathbf{D}} : \tau_2 \qquad \tau_2 \equiv \mathit{Int}$$
$$\mathrm{y}_3^{\mathbf{D}} : \tau_9 \qquad \tau_9 \equiv \tau_7$$
$$\tau_7 \equiv \tau_8 \qquad \delta_4 : \tau_4$$
$$\delta_8 : \tau_8 \qquad \tau_6 \equiv \mathit{Bool}$$
$$\tau_4 \equiv \mathit{Int} \qquad \tau_5 \equiv \mathit{Int}$$
$$\tau_7 \equiv \mathit{Int}$$

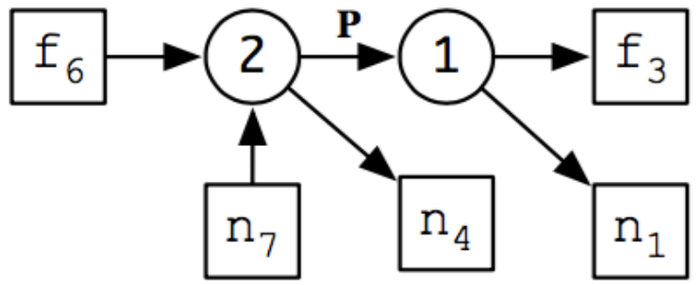
Solution

$$\delta_4 = \delta_8 = \mathrm{x}_1^{\mathbf{D}}$$
$$\tau_2 = \tau_4 = \tau_5 = \mathit{Int}$$
$$\tau_7 = \tau_8 = \tau_9 = \mathit{Int}$$
$$\tau_6 = \mathit{Bool}$$

Lexical Scope

```
def n1 = true2
def f3 = (
    fun (n4 : Int5) {
        f6 (n7)
    }8
)9
```

Scope graph constraints



Resolution Constraints

$$\mathtt{f}_6^{\mathbf{R}} \mapsto \delta_6 \qquad \mathtt{n}_7^{\mathbf{R}} \mapsto \delta_7$$
$$!\overline{\mathcal{D}}(1) \qquad !\overline{\mathcal{D}}(2)$$

Typing constraints

$$\mathtt{n}_1^{\mathbf{D}} : \tau_2 \qquad \tau_2 \equiv \mathit{Bool}$$
$$\mathtt{f}_3^{\mathbf{D}} : \tau_9 \qquad \tau_9 \equiv \mathit{Fun}[\tau_5,\tau_8]$$
$$\mathtt{n}_4^{\mathbf{D}} : \tau_5 \qquad \tau_5 \equiv \mathit{Int}$$
$$\delta_6 : \tau_6 \qquad \tau_6 \equiv \mathit{Fun}[\tau_7,\tau_8]$$
$$\delta_7 : \tau_7$$

Solution

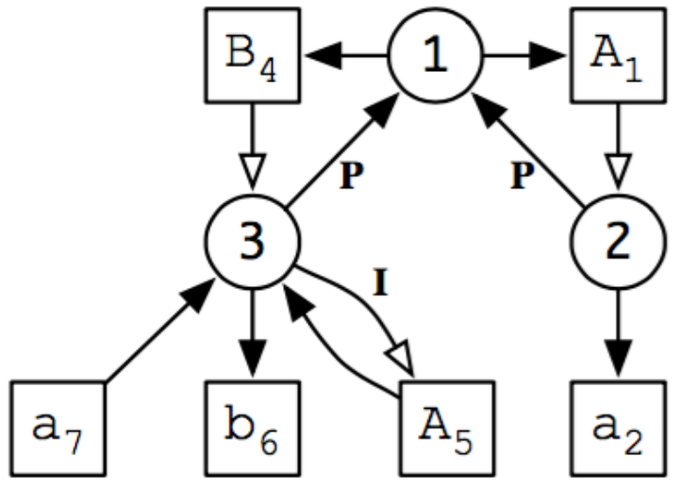
$$\delta_6 = \mathtt{f}_3^{\mathbf{D}} \qquad \delta_7 = \mathtt{n}_4^{\mathbf{D}}$$
$$\tau_2 = \mathit{Bool} \qquad \tau_8 = t_0$$
$$\tau_5 = \tau_7 = \mathit{Int}$$
$$\tau_6 = \tau_9 = \mathit{Fun}[\mathit{Int},t_0]$$

where t_0 is any fixed arbitrary type

Imports

```
module A1 {
    def a2 = 43
}
module B4 {
   import A5
   def b6 = a7
}
```

Scope graph constraints



Resolution constraints

$$\mathtt{a}_7^{\mathbf{R}} \mapsto \delta_7 \qquad !\overline{\mathcal{D}}(1)$$
$$!\overline{\mathcal{D}}(2) \qquad !\overline{\mathcal{D}}(3)$$

Typing constraints

$$\mathtt{a}_2^{\mathbf{D}} : \tau_3 \qquad \tau_3 \equiv \mathit{Int}$$
$$\mathtt{b}_6^{\mathbf{D}} : \tau_7 \qquad \delta_7 : \tau_7$$

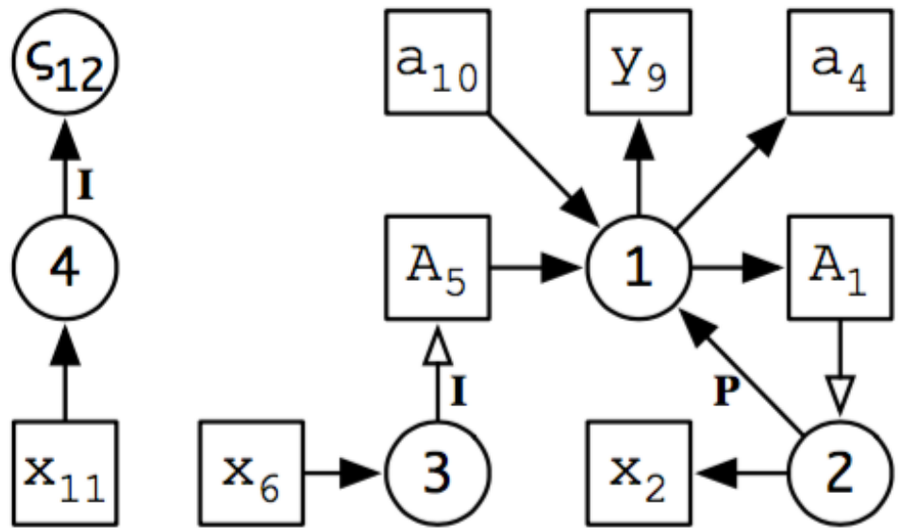
Solution

$$\delta_7 = \mathtt{a}_2^{\mathbf{D}}$$
$$\tau_3 = \mathit{Int} \qquad \tau_7 = \mathit{Int}$$

Type Dependent Name Resolution

```
record A1 { x2 : Int3 }
def a4 = ( new A5{x6=17} )8
def y9 = ( a10.x11 )12
```

Scope graph constraints



Resolution constraints

$$\mathrm{A}_5^{\mathbf{R}} \mapsto \delta_8 \qquad \mathrm{x}_6^{\mathbf{R}} \mapsto \delta_6$$
$$\mathrm{a}_{10}^{\mathbf{R}} \mapsto \delta_{10} \qquad \mathrm{x}_{11}^{\mathbf{R}} \mapsto \delta_{11}$$
$$!\overline{\mathcal{D}}(1) \qquad !\overline{\mathcal{D}}(2)$$
$$\overline{\mathcal{V}}(3) \approx \overline{\mathcal{R}}(3) \qquad \delta_{12} \rightsquigarrow \varsigma_{12}$$

Typing constraints

$$\mathrm{x}_2^{\mathbf{D}} : \tau_3 \qquad \tau_3 \equiv Int$$
$$\mathrm{a}_4^{\mathbf{D}} : \tau_8 \qquad \tau_8 \equiv Rec(\delta_8)$$
$$\mathrm{y}_9^{\mathbf{D}} : \tau_{12} \qquad \delta_6 : \tau_6$$
$$\delta_{10} : \tau_{10} \qquad \delta_{11} : \tau_{12}$$
$$\tau_7 \equiv \tau_6 \qquad \tau_7 \equiv Int$$
$$\tau_{10} \equiv Rec(\delta_{12})$$

Solution

$$\delta_6 = \delta_{11} = \mathrm{x}_2^{\mathbf{D}}$$
$$\delta_8 = \delta_{12} = \mathrm{A}_1^{\mathbf{D}}$$
$$\delta_{10} = \mathrm{a}_4^{\mathbf{D}} \qquad \varsigma_{12} = 2$$
$$\tau_3 = \tau_6 = \tau_7 = \tau_{12} = Int$$
$$\tau_8 = \tau_{10} = Rec(\mathrm{A}_1^{\mathbf{D}})$$

Constraints for Declarations

$$
\begin{array}{rcl}
[\![ds]\!]^{prog} & := & !\overline{\mathcal{D}}(s) \;\wedge\; [\![ds]\!]_s^{decl^*} \\[2ex]
[\![\textbf{module}\; x_i\; \{\, ds \,\}]\!]_s^{decl} & := & s \longrightarrow x_i^{\mathsf{D}} \;\wedge\; x_i^{\mathsf{D}} \longrightarrow\!\!\triangleright\, s' \;\wedge\; s' \xrightarrow{\mathbf{P}} s \;\wedge\; !\overline{\mathcal{D}}(s') \;\wedge\; [\![ds]\!]_{s'}^{decl^*} \\[2ex]
[\![\textbf{import}\; x_i]\!]_s^{decl} & := & x_i^{\mathsf{R}} \longrightarrow s \;\wedge\; s \xrightarrow{\mathbf{I}}\!\!\triangleright\, x_i^{\mathsf{R}} \\[2ex]
[\![\textbf{def}\; b]\!]_s^{decl} & := & [\![b]\!]_s^{bind} \\[2ex]
[\![\textbf{record}\; x_i\; \{\, fs \,\}]\!]_s^{decl} & := & s \longrightarrow x_i^{\mathsf{D}} \;\wedge\; x_i^{\mathsf{D}} \longrightarrow\!\!\triangleright\, s' \;\wedge\; s' \xrightarrow{\mathbf{P}} s \;\wedge\; !\overline{\mathcal{D}}(s') \;\wedge\; [\![fs]\!]_{s,s'}^{fdecl^*} \\[2ex]
[\![x_i = e]\!]_s^{bind} & := & s \longrightarrow x_i^{\mathsf{D}} \;\wedge\; x_i^{\mathsf{D}} : \tau \;\wedge\; [\![e]\!]_{s,\tau}^{exp} \\[2ex]
[\![x_i \,:\, t = e]\!]_s^{bind} & := & s \longrightarrow x_i^{\mathsf{D}} \;\wedge\; x_i^{\mathsf{D}} : \tau \;\wedge\; [\![t]\!]_{s,\tau}^{ty} \;\wedge\; [\![e]\!]_{s,\tau}^{exp} \\[2ex]
[\![x_i : t]\!]_{s_r,s_d}^{fdecl} & := & s_d \longrightarrow x_i^{\mathsf{D}} \;\wedge\; x_i^{\mathsf{D}} : \tau \;\wedge\; [\![t]\!]_{s_r,\tau}^{ty}
\end{array}
$$

Constraints for Expressions

$$
\begin{array}{rcl}
[\![\texttt{fun}\ (x_i : t_1)\ \{e\}]\!]^{exp}_{s,t} & := & t \equiv \mathit{Fun}[\tau_1,\tau_2] \wedge s' \xrightarrow{\mathbf{P}} s \wedge !\overline{\mathcal{D}}(s') \wedge s' \longrightarrow x_i^{\mathbf{D}} \\
 & & \wedge\ x_i^{\mathbf{D}} : \tau_1 \wedge [\![t_1]\!]^{ty}_{s,\tau_1} \wedge [\![e]\!]^{exp}_{s',\tau_2} \\
[\![\texttt{letrec}\ bs\ \texttt{in}\ e]\!]^{exp}_{s,t} & := & s' \xrightarrow{\mathbf{P}} s \wedge !\overline{\mathcal{D}}(s') \wedge [\![bs]\!]^{bind}_{s'} \wedge [\![e]\!]^{exp}_{s',t} \\
[\![n]\!]^{exp}_{s,t} & := & t \equiv \mathit{Int} \\
[\![\texttt{true}]\!]^{exp}_{s,t} & := & t \equiv \mathit{Bool} \\
[\![\texttt{false}]\!]^{exp}_{s,t} & := & t \equiv \mathit{Bool} \\
[\![e_1 \oplus e_2]\!]^{exp}_{s,t} & := & t \equiv t_3 \wedge \tau_1 \equiv t_1 \wedge \tau_2 \equiv t_2 \wedge [\![e_1]\!]^{exp}_{s,\tau_1} \wedge [\![e_2]\!]^{exp}_{s,\tau_2} \\
 & & \textit{(where } \oplus \textit{ has type } t_1 \times t_2 \to t_3\textit{)} \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]^{exp}_{s,t} & := & \tau_1 \equiv \mathit{Bool} \wedge [\![e_1]\!]^{exp}_{s,\tau_1} \wedge [\![e_2]\!]^{exp}_{s,t} \wedge [\![e_3]\!]^{exp}_{s,t} \\
[\![x_i]\!]^{exp}_{s,t} & := & x_i^{\mathbf{R}} \longrightarrow s \wedge x_i^{\mathbf{R}} \mapsto \delta \wedge \delta : t \\
[\![e_1\ e_2]\!]^{exp}_{s,t} & := & \tau \equiv \mathit{Fun}[\tau_1,t] \wedge [\![e_1]\!]^{exp}_{s,\tau} \wedge [\![e_2]\!]^{exp}_{s,\tau_1} \\
[\![e\,.\,x_i]\!]^{exp}_{s,t} & := & [\![e]\!]^{exp}_{s,\tau} \wedge \tau \equiv \mathit{Rec}(\delta) \wedge \delta \rightsquigarrow \varsigma \wedge s' \xrightarrow{\mathbf{I}} \varsigma \wedge [\![x_i]\!]^{exp}_{s',t} \\
[\![\texttt{with}\ e_1\ \texttt{do}\ e_2]\!]^{exp}_{s,t} & := & [\![e_1]\!]^{exp}_{s,\tau} \wedge \tau \equiv \mathit{Rec}(\delta) \wedge \delta \rightsquigarrow \varsigma \\
 & & \wedge\ s' \xrightarrow{\mathbf{P}} s \wedge s' \xrightarrow{\mathbf{I}} \varsigma \wedge [\![e_2]\!]^{exp}_{s',t} \\
[\![\texttt{new}\ x_i\ \{\,bs\,\}]\!]^{exp}_{s,t} & := & x_i^{\mathbf{R}} \longrightarrow s \wedge x_i^{\mathbf{R}} \mapsto \delta \wedge s' \xrightarrow{\mathbf{I}} x_i^{\mathbf{R}} \\
 & & \wedge\ [\![bs]\!]^{fbind^*}_{s,s'} \wedge \overline{\mathcal{V}}(s') \approx \overline{\mathcal{R}}(s') \wedge t \equiv \mathit{Rec}(\delta) \\
[\![x_i = e]\!]^{fbind}_{s,s'} & := & x_i^{\mathbf{R}} \longrightarrow s' \wedge x_i^{\mathbf{R}} \mapsto \delta \wedge \delta : \tau \wedge [\![e]\!]^{exp}_{s,\tau}
\end{array}
$$

Constraint Generation with NaBL2

Constraint Generation

```
module constraint-gen

imports signatures/-

rules

  [[ Constr(t1, ..., tn) ^ (s1, …, sn) : ty ]] :=
     C1,
     ...,
     Cn.
```

one rule per abstract syntax constructor

Generic Constraints

```
false                  // always fails

true                   // always succeeds

C1, C2                 // conjunction of constraints

[[ e ^ (s) : ty ]] // generate constraints for sub-term


C | error $[something is wrong]    // custom error message

C | warning $[does not look right] // custom warning
```

Visibility Policy and Scope Graph Constraints

```
name resolution
  namespaces
    NS1 NS2
  labels
    P I
  well-formedness
    P* . I*
  order
    D < P,
    D < I,
    I < P
```

```
new s            // generate a new scope

NS{x} <- s       // declaration of name x in namespace NS in scope s

NS{x} -> s       // reference of name x in namespace NS in scope s

s ---> s'        // unlabeled scope edge from s to s’

s -L-> s'        // scope edge from s to s' labeled L

NS{x} |-> d      // resolve reference x in namespace NS to declaration d
```

Name Set Constraints

```
distinct D(s)/NS, // declarations for NS in s should be distinct

D(s1) subseteq R(s2), // name set

D(s_rec)/Field subseteq R(s_use)/Field
                          | error $[Field [NAME] not initialized] @r
```

Note: incomplete

Type Signature and Type Constraints

```
signature
  types
    TC1()
    TC2(type)
    TC3(scope)
    TC4(type, scope, type)
```

```
[[ e ^ (s) : ty ]] // subterm e has type ty under scope s

o : ty             // occurrence o has type ty

o : ty !           // with priority

ty1 == ty2         // ty1 and ty2 should unify

ty <! ty2          // declare ty1 a subtype of ty2

ty1 <? ty2         // is ty1 a subtype of ty1?
```

Tiger in NaBL2

Visibility Policy and Type Signature

```
signature
  name resolution
    namespaces
      Type Var Field Loop
    labels
      P I
  well-formedness
    P* . I*
  order
    D < P,
    D < I,
    I < P
```

```
signature
  types
    UNIT()
    INT()
    STRING()
    NIL()
    RECORD(scope)
    ARRAY(type, scope)
    FUN(List(type), type)
```

Declaration of Built-in Types and Functions

```
init ^ (s) : ty_init :=
  new s,                    // the root scope

  Type{"int"} <- s,     // declare primitive type int
  Type{"int"} : INT() !!,

  Type{"string"} <- s, // declare primitive type string
  Type{"string"} : STRING() !!,

  // standard library

  Var{"print"} <- s,
  Var{"print"} : FUN([STRING()], UNIT()) !!,

  …

  Var{"exit"} <- s,
  Var{"exit"} : FUN([INT()], UNIT()) !!.
```

```
module nabl-lib
rules

  [[ None() ^ (s) ]] := true.
  [[ Some(e) ^ (s) ]] := [[ e ^ (s) ]].

  Map[[ [] ^ (s) ]] := true.
  Map[[ [ x | xs ] ^ (s) ]] :=
    [[ x ^ (s) ]], Map[[ xs ^ (s) ]].

  Map2[[ [] ^ (s, s') ]] := true.
  Map2[[ [ x | xs ] ^ (s, s') ]] :=
    [[ x ^ (s, s') ]], Map2[[ xs ^ (s, s') ]].

  MapT2[[ [] ^ (s, s') : [] ]] := true.
  MapT2[[ [ x | xs ] ^ (s, s') : [ty | tys] ]] :=
    [[ x ^ (s, s') : ty ]], MapT2[[ xs ^ (s, s') : tys ]].

  MapT[[ [] ^ (s) : ty ]] := true.
  MapT[[ [ x | xs ] ^ (s) : ty ]] :=
    [[ x ^ (s) : ty ]], MapT[[ xs ^ (s) : ty ]].

  MapTs[[ [] ^ (s) : [] ]] := true.
  MapTs[[ [ x | xs ] ^ (s) : [ty | tys] ]] :=
    [[ x ^ (s) : ty ]],
    MapTs[[ xs ^ (s) : tys ]].

  MapTs2[[ [] ^ (s1, s2) : [] ]] := true.
  MapTs2[[ [ x | xs ] ^ (s1, s2) : [ty | tys] ]] :=
    [[ x ^ (s1, s2) : ty ]], MapTs2[[ xs ^ (s1, s2) : tys ]].
```

Constraint Generation for Lists of Terms

Non-Binding Constructs

Literals and Operators

```
rules // literals

  [[ Int(i)      ^ (s) : INT() ]] := true.
  [[ String(str) ^ (s) : STRING() ]] := true.
  [[ NilExp()    ^ (s) : NIL() ]] := true.

rules // operators

  [[ Uminus(e) ^ (s) : INT() ]] :=
     [[ e ^ (s) : INT() ]].

  [[ Divide(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : INT() ]], [[ e2 ^ (s): INT() ]].

  [[ Times(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : INT() ]], [[ e2 ^ (s): INT() ]].

  // etc.
```

Assignment

```
rules

  [[ Assign(e1, e2) ^ (s) : UNIT() ]] :=
     [[ e1 ^ (s) : ty1 ]],
     [[ e2 ^ (s) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch] @ e2.
```

Sequences

```
rules

  [[ Seq(es) ^ (s) : ty ]] :=
     Seq[[ es ^ (s) : ty ]].

  Seq[[ [] ^ (s) : UNIT() ]] :=
    true.

  Seq[[ [e] ^ (s) : ty ]] :=
    [[ e ^ (s) : ty ]].

  Seq[[ [ e | es@[_|_] ] ^ (s) : ty ]] :=
    [[ e ^ (s) : ty' ]],
    Seq[[ es ^ (s) : ty ]].
```

Control-Flow

```
rules

  [[ If(e1, e2, e3) ^ (s) : ty2 ]] :=
     [[ e1 ^ (s) : INT() ]],
     [[ e2 ^ (s) : ty2 ]],
     [[ e3 ^ (s) : ty3 ]],
     ty2 == ty3 | error $[branches should have same type].

  [[ IfThen(e1, e2) ^ (s) : UNIT() ]] :=
     [[ e1 ^ (s) : INT() ]],
     [[ e2 ^ (s) : ty ]].

  [[ While(e1, e2) ^ (s) : UNIT() ]] :=
     new s', s' -P-> s,
     [[ e1 ^ (s) : INT() ]],
     [[ e2 ^ (s') : ty ]].
```

Binding Constructs

Lets Bind Sequentially

```
rules // let

  [[ Let(blocks, exps) ^ (s) : ty ]] :=
     new s_body,
     distinct D(s_body),
     Decs[[ blocks ^ (s, s_body) ]],
     Seq[[ exps ^ (s_body) : ty ]].

  Decs[[ [] ^ (s_outer, s_body) ]] :=
    s_body -P-> s_outer.

  Decs[[ [block] ^ (s_outer, s_body) ]] :=
    s_body -P-> s_outer,
    Dec[[ block ^ (s_body, s_outer) ]].

  Decs[[ [block | blocks@[_|_]] ^ (s_outer, s_body) ]] :=
    new s_dec,
    s_dec -P-> s_outer,
    Dec[[ block ^ (s_dec, s_outer) ]],
    Decs[[ blocks ^ (s_dec, s_body) ]].
```

```
let
  var x : int := 0 + z  // z not in scope
  var y : int := x + 1
  var z : int := x + y + 1
 in
   x + y + z
end
```

Variable Declarations and References

```
rules // variable declarations

  Dec[[ VarDec(x, t, e) ^ (s, s_outer) ]] :=
     Var{x} <- s, Var{x} : ty1 !,
     [[ t ^ (s_outer) : ty1 ]],
     [[ e ^ (s_outer) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch] @ e.

  Dec[[ VarDecNoInit(x, t) ^ (s, s_outer) ]] :=
     Var{x} <- s, Var{x} : ty !,
     [[ t ^ (s_outer) : ty ]].

rules // variable references

  [[ Var(x) ^ (s) : ty ]] :=
     Var{x} -> s,
     Var{x} |-> d,
     d : ty.
```

```
let
  var x : int := 5
  var f : int := 1
in
  for y := 1 to x do (
    f := f * y
  )
end
```

Type Declarations

```
rules

  Dec[[ TypeDecs(tdecs) ^ (s, s_outer) ]] :=
     Map[[ tdecs ^ (s) ]].

  [[ TypeDec(x, t) ^ (s) ]] :=
     Type{x} <- s, Type{x} : ty !,
     [[ t ^ (s) : ty ]].

rules // types

  [[ Tid(x) ^ (s) : ty ]] :=
     Type{x} -> s,
     Type{x} |-> d | error $[Type [x] not declared],
     d : ty.
```

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
 in foo(56) + foo // both refer to the variable foo
end
```

Functions

Adjacent Functions are Mutually Recursive

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

Function Definitions

```
rules

  Dec[[ FunDecs(fdecs) ^ (s, s_outer) ]] :=
     Map2[[ fdecs ^ (s, s_outer) ]].

  [[ FunDec(f, args, t, e) ^ (s, s_outer) ]] :=
     Var{f} <- s,
     Var{f} : FUN(tys, ty) !,
     new s_fun,
     s_fun -P-> s,
     distinct D(s_fun) | error $[duplicate argument] @ NAMES,
     MapTs2[[ args ^ (s_fun, s_outer) : tys ]],
     [[ t ^ (s_outer) : ty ]],
     [[ e ^ (s_fun) : ty_body ]],
     ty == ty_body| error $[return type does not match body] @ t.

  [[ FArg(x, t) ^ (s_fun, s_outer) : ty ]] :=
     Var{x} <- s_fun,
     Var{x} : ty !,
     [[ t ^ (s_outer) : ty ]].
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in fact(10)
end
```

Function Calls

```
rules

  [[ Call(Var(f), exps) ^ (s) : ty ]] :=
     Var{f} -> s,
     Var{f} |-> d | error $[Function [f] not declared],
     d : FUN(tys, ty) | error $[Function expected] ,
     MapTs[[ exps ^ (s) : tys ]].
```

```
let function fact(n : int) : int =
      if n < 1 then 1 else (n * fact(n - 1))
 in fact(10)
end
```

Records

Type Dependent Name Resolution

```
let
  type point = {x : int, y : int}
  var origin : point := point { x = 1, y = 2 }
 in origin.x
end
```

Errors in Record Declaration and Creation

Duplicate Declaration of Field “x”

```
let
  type point = {x : int, y : int}
  type errpoint = {x : int, x : int}
  var p : point
  var e : errpoint
 in
   p := point{ x = 3, y = 3, z = "a" }
   p := point{ x = 3 }
end
```

Reference “z” not resolved

Field “y” not initialized

Recursive Types

```
let
  type intlist = {hd : int, tl : intlist}
  type tree = {key : int, children : treelist}
  type treelist = {hd : tree, tl : treelist}
  var l : intlist
  var t : tree
  var tl : treelist
 in
  l := intlist { hd = 3, tl = l };
  t := tree {
    key = 2,
    children = treelist {
      hd = tree{ key = 3, children = 3 },
      tl = treelist{ }
    }
  };
  t.children.hd.children := t.children
end
```

type mismatch

Field "tl" not initialized
Field "hd" not initialized

NIL is a Subtype of RECORD

```
let
  type intlist = {hd : int, tl : intlist}
  var l : intlist := nil
 in
   l := intlist{ hd = 1, tl = l };
   l := intlist{ hd = 2, tl = l };
   l := intlist{ hd = 3, tl = l }
end
```

Record Types

```
rules

  [[ RecordTy(fields) ^ (s) : ty ]] :=
     new s_rec,
     ty == RECORD(s_rec),
     NIL() <! ty,
     distinct D(s_rec)/Field
                          | error $[Duplicate declaration of field [NAME]] @ NAMES,
     Map2[[ fields ^ (s_rec, s) ]].

  [[ Field(x, t) ^ (s_rec, s_outer) ]] :=
     Field{x} <- s_rec,
     Field{x} : ty !,
     [[ t ^ (s_outer) : ty ]].
```

```
let
  type point = {x : int, y : int}
  var origin : point := point { x = 1, y = 2 }
 in origin.x
end
```

Record Creation

```
rules

  [[ r@Record(t, inits) ^ (s) : ty ]] :=
     [[ t ^ (s) : ty ]],
     ty == RECORD(s_rec) | error $[record type expected],
     new s_use, s_use -I-> s_rec,
     D(s_rec)/Field subseteq R(s_use)/Field
                                    | error $[Field [NAME] not initialized] @r,
     Map2[[ inits ^ (s_use, s) ]].

  [[ InitField(x, e) ^ (s_use, s) ]] :=
     Field{x} -> s_use,
     Field{x} |-> d,
     d : ty1,
     [[ e ^ (s) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch].
```

```
let
  type point = {x : int, y : int}
  var origin : point := point { x = 1, y = 2 }
 in origin.x
end
```

Record Field Access

```
rules

  [[ FieldVar(e, f) ^ (s) : ty ]] :=
     [[ e ^ (s) : ty_e ]],
     ty_e == RECORD(s_rec),
     new s_use, s_use -I-> s_rec,
     Field{f} -> s_use,
     Field{f} |-> d,
     d : ty.
```

```
let
  type point = {x : int, y : int}
  var origin : point := point { x = 1, y = 2 }
 in origin.x
end
```

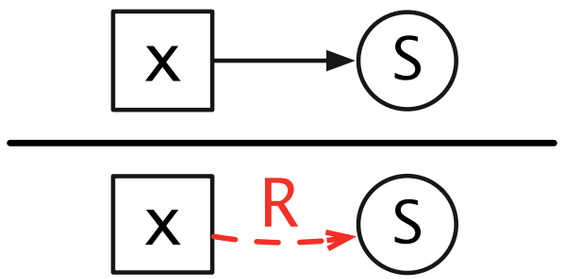
On GitHub

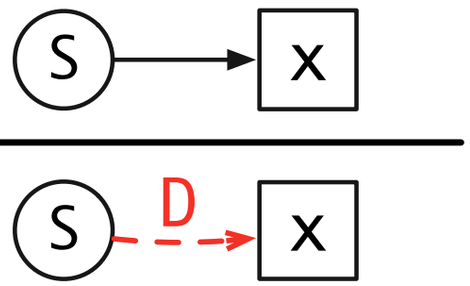
- Arrays
- For loops

Formal Framework
(with labeled scope edges)

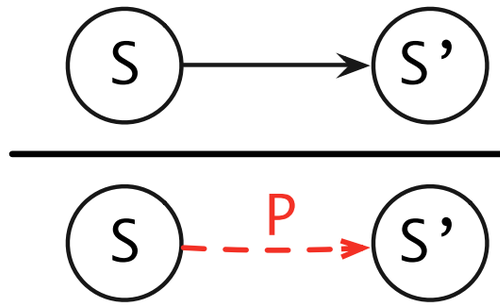
Issues with the Reachability Calculus

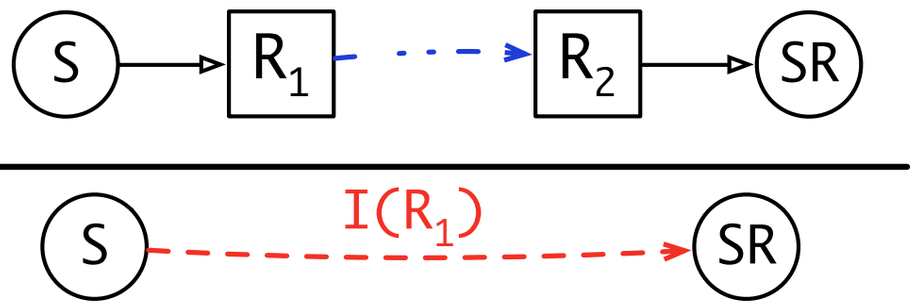
Disambiguating import paths





Cyclic Import Paths





Multi-import interpretation

Well formed path: `R.P*.I(_)*.D`

Fixed visibility policy

$$\frac{}{D < P.p} \qquad \frac{}{I(\_).p' < P.p}$$

$$\frac{p < p'}{s.p < s.p'} \qquad \frac{}{D < I(\_).p'}$$

Resolution Calculus with Edge Labels

$$\mathrm{R} \longrightarrow \mathrm{S} \mid \mathrm{S} \longrightarrow \mathrm{D} \mid \mathrm{S} \xrightarrow{l} \mathrm{S} \mid \mathrm{D} \longrightarrow\!\!\triangleright \mathrm{S} \mid \mathrm{S} \xrightarrow{l}\!\!\triangleright \mathrm{R}$$

Resolution paths

$$\begin{array}{rcl} s & := & \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{E}(l, S) \mid \mathbf{N}(l, x_i^{\mathsf{R}}, S) \\ p & := & [] \mid s \mid p \cdot p \qquad \text{(inductively generated)} \end{array}$$

$$[] \cdot p = p \cdot [] = p$$
$$(p_1 \cdot p_2) \cdot\ p_3 = p_1 \cdot (p_2 \cdot p_3)$$

Well-formed paths

$$WF(p) \Leftrightarrow labels(p) \in \mathcal{E}$$

Visibility ordering on paths

$$\frac{label(s_1) < label(s_2)}{s_1 \cdot p_1 < s_2 \cdot p_2} \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2}$$

Edges in scope graph

$$\frac{S_1 \xrightarrow{l} S_2}{\mathbb{I} \vdash \mathbf{E}(l, S_2) : S_1 \longrightarrow S_2} \quad (E)$$

$$\frac{S_1 \xrightarrow{l}\!\!\triangleright\ y_i^{\mathsf{R}} \quad y_i^{\mathsf{R}} \notin \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}} \quad y_j^{\mathsf{D}} \longrightarrow\!\!\triangleright S_2}{\mathbb{I} \vdash \mathbf{N}(l, y_i^{\mathsf{R}}, S_2) : S_1 \longrightarrow S_2} \quad (N)$$

Transitive closure

$$\frac{}{\mathbb{I}, \mathbb{S} \vdash [] : A \twoheadrightarrow A} \quad (I)$$

$$\frac{B \notin \mathbb{S} \quad \mathbb{I} \vdash s : A \longrightarrow B \quad \mathbb{I}, \{B\} \cup \mathbb{S} \vdash p : B \twoheadrightarrow C}{\mathbb{I}, \mathbb{S} \vdash s \cdot p : A \twoheadrightarrow C} \quad (T)$$

Reachable declarations

$$\frac{\mathbb{I}, \{S\} \vdash p : S \twoheadrightarrow S' \quad WF(p) \quad S' \longrightarrow x_i^{\mathsf{D}}}{\mathbb{I} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \quad (R)$$

Visible declarations

$$\frac{\begin{array}{c}\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \\ \forall j, p' (\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))\end{array}}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}} \quad (V)$$

Reference resolution

$$\frac{x_i^{\mathsf{R}} \longrightarrow S \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \quad (X)$$

Visibility Policies

Lexical scope

$$\mathcal{L} := \{\mathbf{P}\} \quad \mathcal{E} := \mathbf{P}^* \quad \mathbf{D} < \mathbf{P}$$

Non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{I}^? \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P}$$

Transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{TI}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{TI}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}$$

Transitive Includes

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{Inc}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}$$

Transitive includes and imports, and non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}, \mathbf{TI}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot (\mathbf{Inc} \mid \mathbf{TI})^* \cdot \mathbf{I}^?$$

$$\mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P},$$

Seen Imports

$$\dfrac{\dfrac{A_2^{\mathsf{D}}{:}S_{A_2} \in \mathcal{D}(S_{A_1}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{I}(S_{root}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{R}(S_{root}) \qquad A_1^{\mathsf{D}}{:}S_{A_1} \in \mathcal{D}(S_{root})}{A_4^{\mathsf{R}} \longmapsto A_1^{\mathsf{D}}{:}S_{A_1}}}{S_{root} \longrightarrow S_{A_1} \quad (*)}}{S_{root} \rightarrowtail A_2^{\mathsf{D}}{:}S_{A_2}} \qquad A_4^{\mathsf{R}} \in \mathcal{R}(S_{root}) \quad S_{root} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}}{A_4^{\mathsf{R}} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}}$$

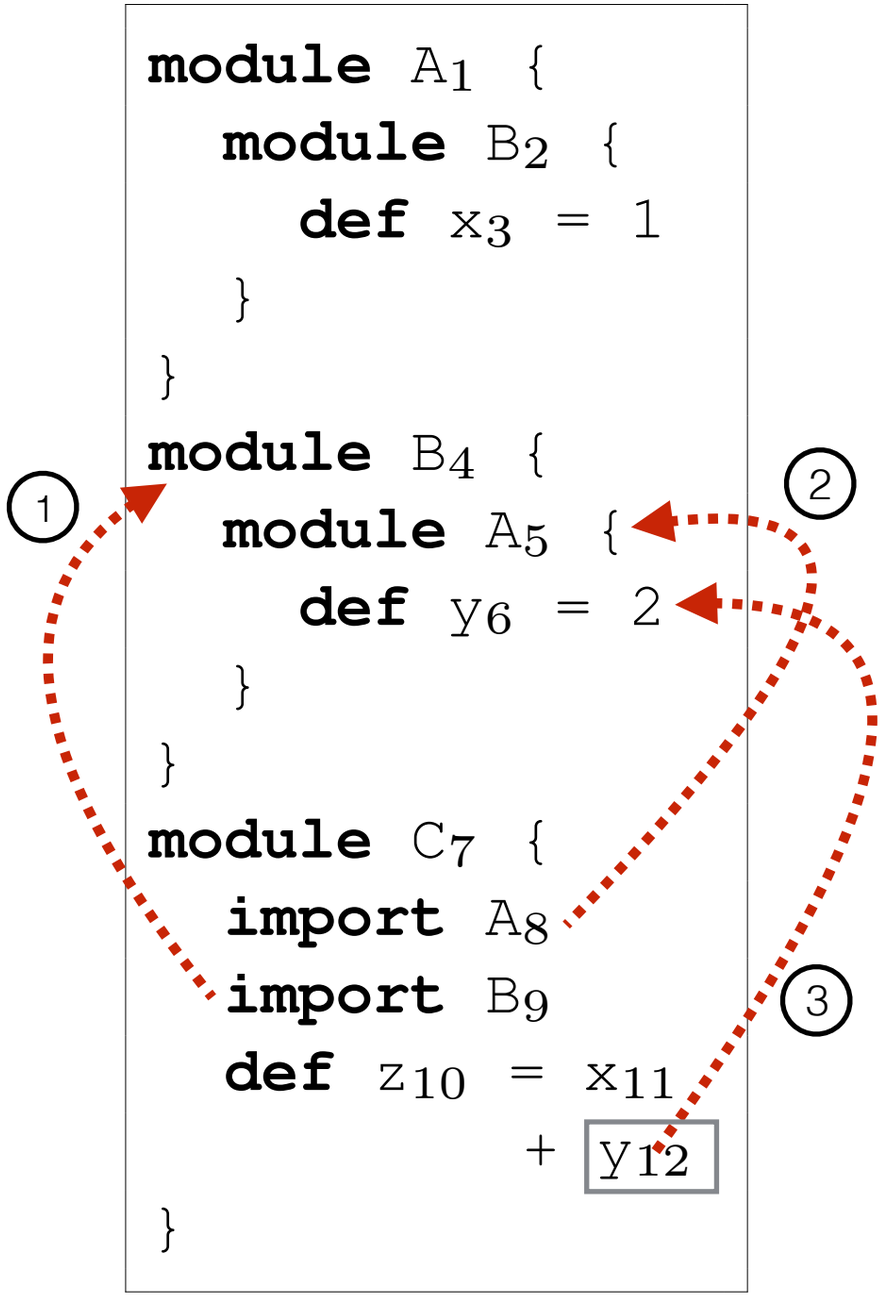
```
module A1 {
   module A2 {
      def a3 = ...
   }
}
import A4
def b5 = a6
```

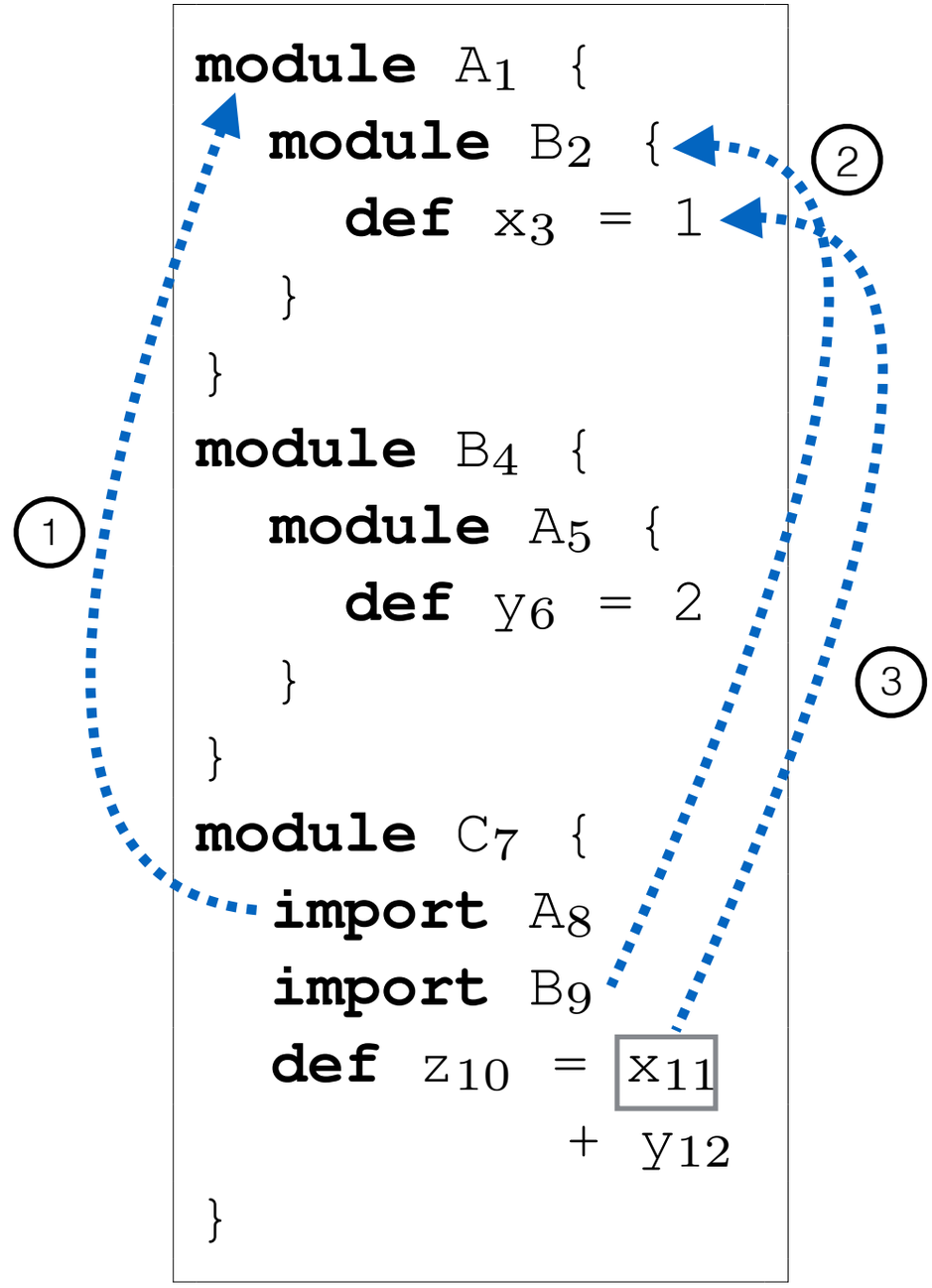
?

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}}$$

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \setminus \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbb{I} \vdash \mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2}$$

Anomaly


module A1 {
module B2 {
def x3 = 1
}
}
module B4 {
module A5 {
def y6 = 2
}
}
module C7 {
import A8
import B9
def z10 = x11
+ y12
}
1
2
3


module A1 {
module B2 {
def x3 = 1
}
}
module B4 {
module A5 {
def y6 = 2
}
}
module C7 {
import A8
import B9
def z10 = x11
+ y12
}
1
2
3

Resolution Algorithm

$$R[\mathbb{I}](x^{\mathbf{R}}) := \text{let } (r, s) = \mathit{Env}_{\mathcal{E}}[\{x^{\mathbf{R}}\} \cup \mathbb{I}, \emptyset](\mathcal{S}c(x^{\mathbf{R}}))\} \text{ in } \begin{cases} \mathsf{U} & \text{if } r = \mathsf{P} \text{ and } \{x^{\mathbf{D}} | x^{\mathbf{D}} \in s\} = \emptyset \\ \{x^{\mathbf{D}} | x^{\mathbf{D}} \in s\} \end{cases}$$

$$\mathit{Env}_{re}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{T},\ \emptyset) & \text{if } S \in \mathbb{S} \text{ or } re = \varnothing \\ \mathit{Env}_{re}^{\mathcal{L} \cup \{\mathbf{D}\}}[\mathbb{I}, \mathbb{S}](S) \end{cases}$$

$$\mathit{Env}_{re}^{L}[\mathbb{I}, \mathbb{S}](S) := \bigcup_{l \in Max(L)} \left(\mathit{Env}_{re}^{\{l' \in L | l' < l\}}[\mathbb{I}, \mathbb{S}](S) \lhd \mathit{Env}_{re}^{l}[\mathbb{I}, \mathbb{S}](S) \right)$$

$$\mathit{Env}_{re}^{\mathbf{D}}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{T},\ \emptyset) & \text{if } [] \notin re \\ (\mathsf{T},\ \mathcal{D}(S)) \end{cases}$$

$$\mathit{Env}_{re}^{l}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{P},\ \emptyset) \text{ if } S_l^{\blacktriangleright} \text{ contains a variable or } \mathit{IS}^l[\mathbb{I}](S) = \mathsf{U} \\ \bigcup_{S' \in \left(\mathit{IS}^l[\mathbb{I}](S) \cup S_l^{\blacktriangleright}\right)} \mathit{Env}_{(l^{-1}\, re)}[\mathbb{I}, \{S\} \cup \mathbb{S}](S') \end{cases}$$

$$\mathit{IS}^l[\mathbb{I}](S) := \begin{cases} \mathsf{U} & \text{if } \exists y^{\mathbf{R}} \in (S_l^{\triangleright} \setminus \mathbb{I}) \ s.t.\ R[\mathbb{I}](y^{\mathbf{R}}) = \mathsf{U} \\ \{S' \mid y^{\mathbf{R}} \in (S_l^{\triangleright} \setminus \mathbb{I}) \wedge y^{\mathbf{D}} \in R[\mathbb{I}](y^{\mathbf{R}}) \wedge y^{\mathbf{D}} \longrightarrow S'\} \end{cases}$$

Semantics of Constraints

$$\frac{}{\mathcal{G}, \leq, \psi \models \mathsf{True}} \quad \text{(C-True)}$$

$$\frac{\mathcal{G}, \leq, \psi \models C_1 \quad \mathcal{G}, \leq, \psi \models C_2}{\mathcal{G}, \leq, \psi \models C_1 \wedge C_2} \quad \text{(C-And)}$$

$$\frac{\psi(d) = T}{\mathcal{G}, \leq, \psi \models d : T} \quad \text{(C-TypeOf)}$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathbf{R}} \mapsto x_j^{\mathbf{D}}}{\mathcal{G}, \leq, \psi \models x_i^{\mathbf{R}} \mapsto x_j^{\mathbf{D}}} \quad \text{(C-Resolve)}$$

$$\frac{d \longrightarrow\!\triangleright_{\mathcal{G}} S}{\mathcal{G}, \leq, \psi \models d \rightsquigarrow S} \quad \text{(C-ScopeOf)}$$

$$\frac{\forall x, \mathbf{1}_{[\![N]\!]_{\mathcal{G}}}(x) \leq 1}{\mathcal{G}, \leq, \psi \models !N} \quad \text{(C-Unique)}$$

$$\frac{[\![N_1]\!]_{\mathcal{G}} \subseteq [\![N_2]\!]_{\mathcal{G}}}{\mathcal{G}, \leq, \psi \models N_1 \subsetneq N_2} \quad \text{(C-SubName)}$$

$$\frac{t_1 = t_2}{\mathcal{G}, \leq, \psi \models t_1 \equiv t_2} \quad \text{(C-Eq)}$$

Constraint Resolution

$$
\begin{array}{rcll}
(x^{\mathsf{R}} \mapsto \delta \wedge C, \mathcal{G}, \psi) & \longrightarrow & [\delta \mapsto x^{\mathsf{D}}](C, \mathcal{G}, \psi) & \textit{where } x^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}}) \\
(x^{\mathsf{D}} \rightsquigarrow \varsigma \wedge C, \mathcal{G}, \psi) & \longrightarrow & [\varsigma \mapsto S](C, \mathcal{G}, \psi) & \textit{where } x^{\mathsf{D}} \longrightarrow\!\!\triangleright S \\
(T_1 \equiv T_2 \wedge C, \mathcal{G}, \psi) & \longrightarrow & \sigma(C, \mathcal{G}, \psi) & \textit{where } \mathcal{U}(T_1, T_2) = \sigma \\
(!N \wedge C, \mathcal{G}, \psi) & \longrightarrow & (C, \mathcal{G}, \psi) & \textit{where } \forall x \in N_{\mathcal{G}}(N), \mathbf{1}_{N_{\mathcal{G}}(N)}(x) = 1 \\
(N_1 \subsetneq N_2 \wedge C, \mathcal{G}, \psi) & \longrightarrow & (C, \mathcal{G}, \psi) & \textit{where } N_{\mathcal{G}}(N_1) \subseteq N_{\mathcal{G}}(N_2) \\
(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, \psi) & \longrightarrow & \begin{cases} (C, \mathcal{G}, \{x^{\mathsf{D}} \mapsto T\} \cup \psi) & \textit{if } x^{\mathsf{D}} \notin \mathrm{dom}(\psi) \\ (\psi(x^{\mathsf{D}}) \equiv T \wedge C, \mathcal{G}, \psi) & \textit{otherwise} \end{cases} & \\
(\mathsf{True} \wedge C, \mathcal{G}, \psi) & \longrightarrow & (C, \mathcal{G}, \psi) &
\end{array}
$$

Summary

Q1: Compiler Front-End

- **Syntax Definition**
  - from formal grammars to syntax definition
  - derivation of parsers, abstract syntax trees, syntax-aware editors, …
- **Syntax Techniques (1)**
  - automata for lexical analysis
- **Term Rewriting**
  - simple syntactic transformations using term rewrite rules
  - desugaring, outline view
- **Imperative and OO Languages**
  - behavior and types
- **Static Semantics**
  - name resolution using scope graphs
  - type analysis using type constraints

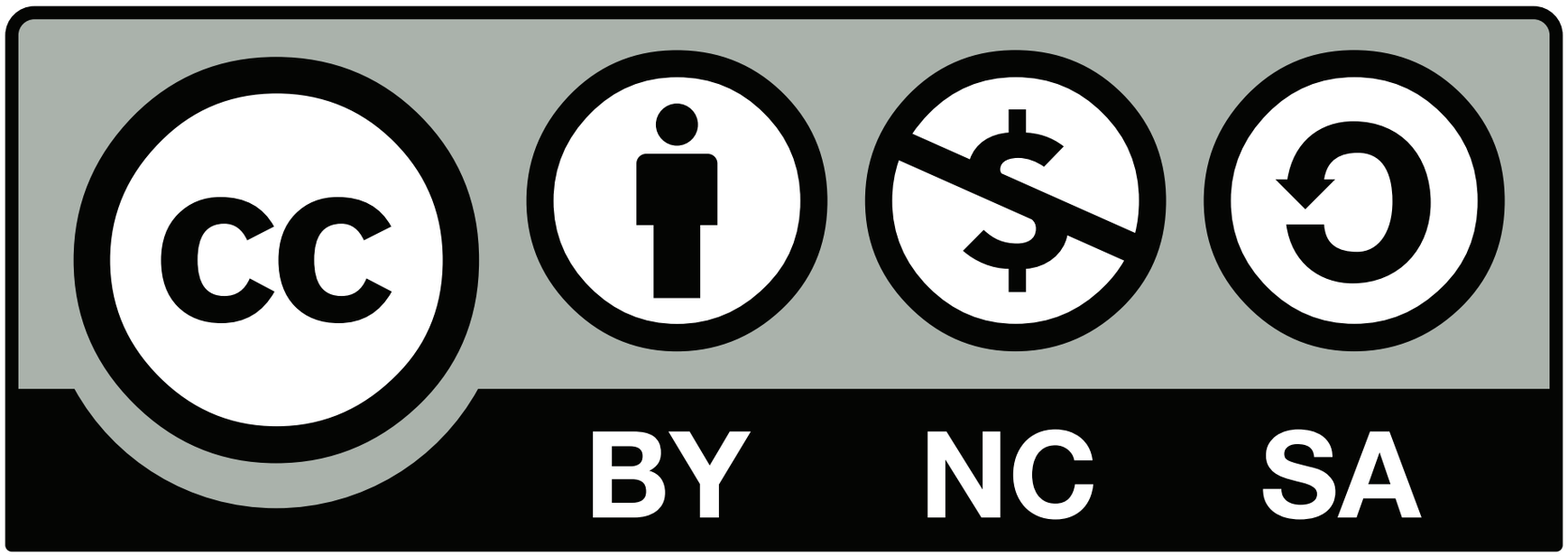
Q2: Compiler Back-End

- **Dynamic Semantics**
  - what is the meaning of programs in a language
- **Target Machine**
  - instruction set is API for programming (virtual) machine
- **Code Generation**
  - from (typed) abstract syntax trees to (virtual) machine code instructions
- **Garbage Collection**
  - techniques for safe automated memory management
- **Register Allocation**
  - mapping unlimited set of variables to limited set of registers
- **Dataflow Analysis**
  - basis for optimizations such as constant propagation, dead code elimination, …
- **Syntax Techniques (2)**
  - LL and LR parsing

First …

http://2016.splashcon.org

copyrights


BY
NC
SA

Pictures

copyrights

Slide 1:

How glorious a greeting the sun gives the mountains! by Justin Kern, some rights reserved