Declare Your Language

Chapter 5: Name Resolution

Eelco Visser

IN4303 Compiler Construction
TU Delft
September 2017

TU Delft

Check that names are used correctly and that expressions are well-typed

Reading Material

Type checkers are algorithms that check names and types in programs.

This paper introduces the NaBL Name Binding Language, which supports the declarative definition of the name binding and scope rules of programming languages through the definition of scopes, declarations, references, and imports associated.

Background

SLE 2012

http://dx.doi.org/10.1007/978-3-642-36089-3_18

Declarative Name Binding and Scope Rules

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser

Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl,
{l.c.l.kats,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. In textual software languages, names are used to reference elements like variables, methods, classes, etc. Name resolution analyses these names in order to establish references between definition and use sites of elements. In this paper, we identify recurring patterns for name bindings in programming languages and introduce a declarative meta-language for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes. Based on such declarative name binding specifications, we provide a language-parametric algorithm for static name resolution during compile-time. We discuss the integration of the algorithm into the Spoofax Language Workbench and show how its results can be employed in semantic editor services like reference resolution, constraint checking, and content completion.

1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity. One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*. Syntax definitions are a prominent example. With declarative formalisms such as EBNF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [15].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 311–331, 2013.

This paper introduces scope graphs as a language-independent representation for the binding information in programs.

ESOP 2015

http://dx.doi.org/10.1007/978-3-662-46669-8_9

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

Separating type checking into constraint generation and constraint solving provides more declarative definition of type checkers. This paper introduces a constraint language integrating name resolution into constraint resolution through scope graph constraints.

This is the basis for the design of the NaBL2 static semantics specification language.

PEPM 2016

https://doi.org/10.1145/2847538.2847543

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.

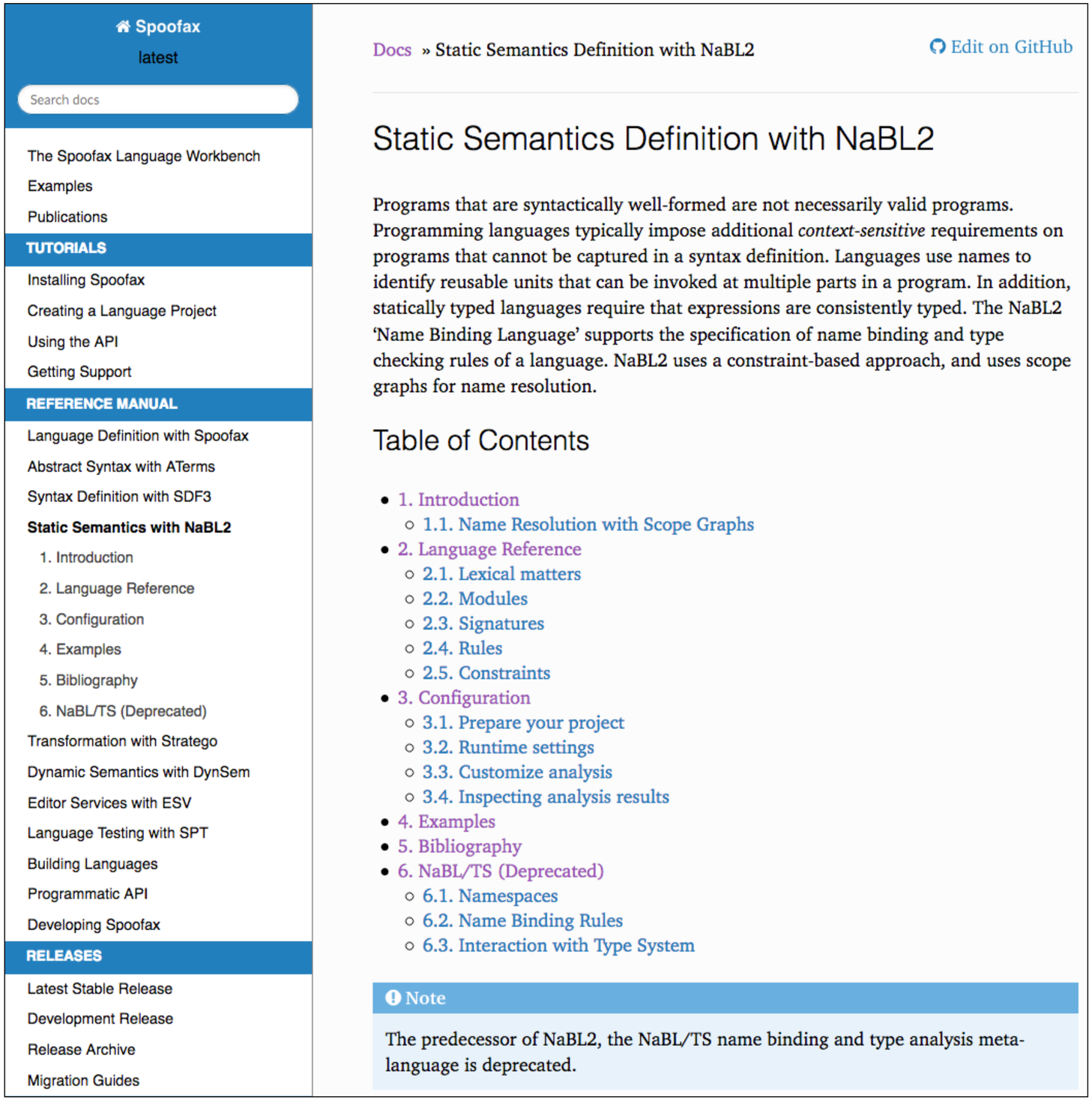
In recent work we show how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting 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,

Documentation for NaBL2 at the metaborg.org website.



http://www.metaborg.org/en/latest/source/langdev/meta/lang/nabl2/index.html

Name Resolution

Examples in Tiger

Name Resolution

```
let
  var x : int := 0 + z
  var y : int := x + 1
  var z : int := x + y + 1
 in
   x + y + z
end
```

Name Resolution: Scope

```
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
```

Name Resolution

```
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
```

Mutually Recursive Functions should be Adjacent

```
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
```

Name Resolution

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
 in foo(56) + foo
end
```

Name Resolution: Name Spaces

```
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 and variables are in the same namespace

Name Resolution

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

Type Dependent Name Resolution

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

Resolving origin.x requires the type of origin

Name Resolution

```
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
```

Name Resolution: Name Set Correspondence

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
```

Field “y” not initialized

Reference “z” not resolved

Name Resolution

```
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
```

Name Resolution: 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

Testing Static Analysis

TU Delft

Testing Name Resolution

```
test outer name [[
   let type t = u
       type [[u]] = int
       var x: [[u]] := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := 42
      end
end
]] resolve #2 to #1
```

```
test inner name [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type [[u]] = t
          var y: [[u]] := 0
      in
         y := 42
      end
end
]] resolve #2 to #1
```

Testing Type Checking

```
test integer constant [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[42]]
      end
end
]] run get-type to INT()
```

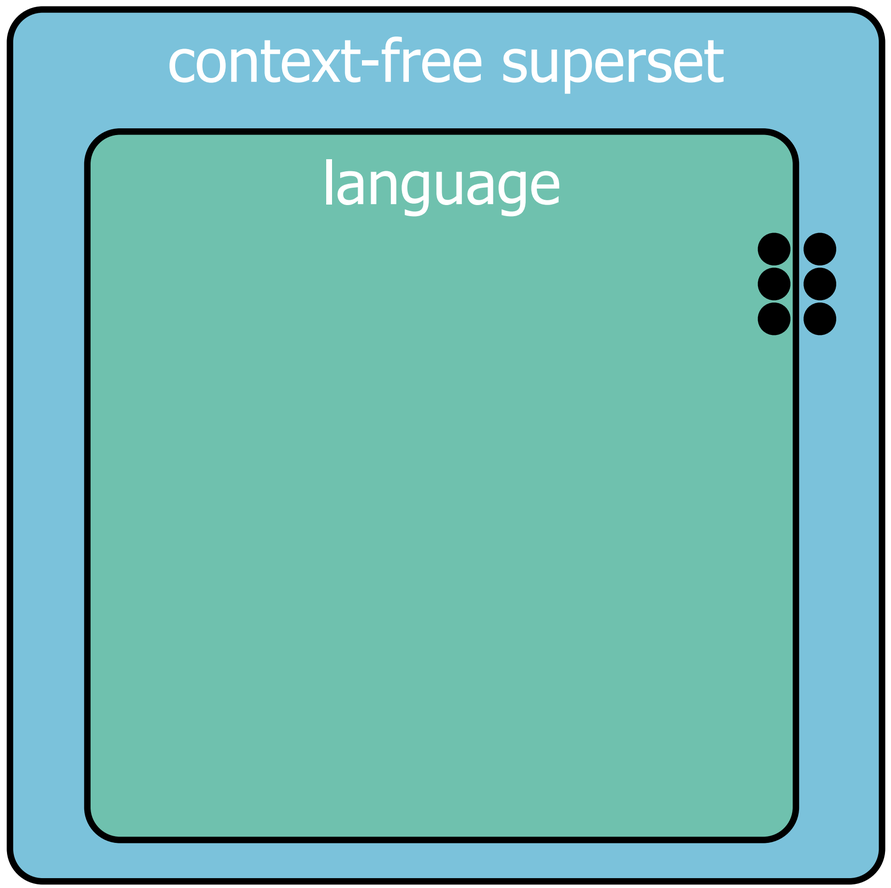
```
test variable reference [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[x]]
      end
end
]] run get-type to INT()
```

Testing Errors

```
test undefined variable [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[z]]
      end
end
]] 1 error
```

```
test type error [[
   let type t = u
       type u = string
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[x]]
      end
end
]] 1 error
```

Test Corner Cases

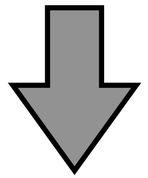


Implementing a Type Checker

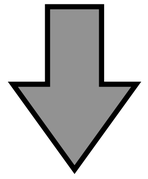
Compute Type of Expression

```
rules

  type-check :
    Mod(e) -> t
    where <type-check> e => t

  type-check :
    String(_) -> STRING()

  type-check :
    Int(_) -> INT()

  type-check :
    Plus(e1, e2) -> INT()
    where
      <type-check> e1 => INT();
      <type-check> e2 => INT()

  type-check :
    Times(e1, e2) -> INT()
    where
      <type-check> e1 => INT();
      <type-check> e2 => INT()
```

```
1 + 2 * 3
```



```
Mod(
  Plus(Int("1"),
       Times(Int("2"),
             Int("3")))
)
```

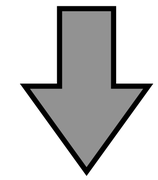


```
INT()
```

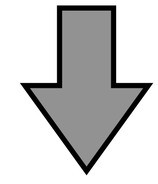
Compute Type of Variable?

```
rules

  type-check :
    Let([VarDec(x, e)], e_body) -> t
    where
      <type-check> e => t_e;
      <type-check> e_body => t

  type-check :
    Var(x) -> t // ???
```

```
let
  var x := 1
 in
   x + 1
end
```



```
Mod(
  Let(
    [VarDec("x", Int("1"))]
  , [Plus(Var("x"), Int("1"))]
  )
)
```

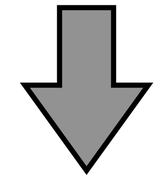


?

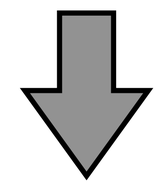
Type Checking Variable Bindings

```
rules

  type-check(|env) :
    Let([VarDec(x, e)], e_body) -> t
    where
      <type-check(|env)> e => t_e;
      <type-check(|[(x, t_e) | env])> e_body => t

  type-check(|env) :
    Var(x) -> t
    where
      <fetch(?(x, t))> env
```

```
let
  var x := 1
 in
   x + 1
end
```



```
Mod(
  Let(
    [VarDec("x", Int("1"))]
  , [Plus(Var("x"), Int("1"))]
  )
)
```



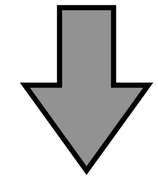
```
INT()
```

Store association between variable and type in type environment

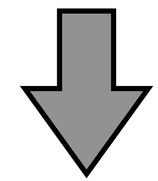
Pass Environment to Sub-Expressions

```
rules

  type-check :
    Mod(e) -> t
    where <type-check(|[])> e => t

  type-check(|env) :
    String(_) -> STRING()

  type-check(|env) :
    Int(_) -> INT()

  type-check(|env) :
    Plus(e1, e2) -> INT()
    where
      <type-check(|env)> e1 => INT();
      <type-check(|env)> e2 => INT()

  type-check(|env) :
    Times(e1, e2) -> INT()
    where
      <type-check(|env)> e1 => INT();
      <type-check(|env)> e2 => INT()
```

```
let
  var x := 1
 in
   x + 1
end
```



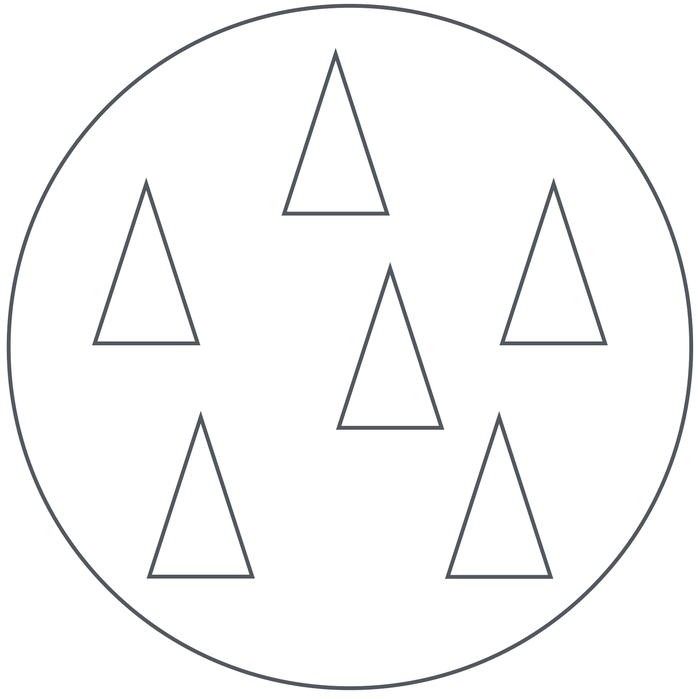
```
Mod(
  Let(
    [VarDec("x", Int("1"))]
  , [Plus(Var("x"), Int("1"))]
  )
)
```

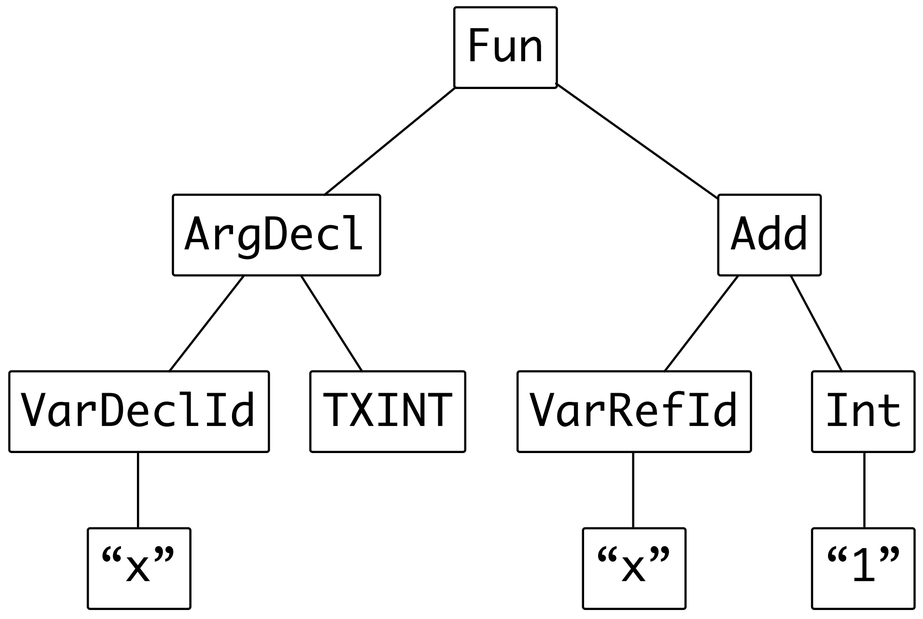


```
INT()
```

Name Resolution

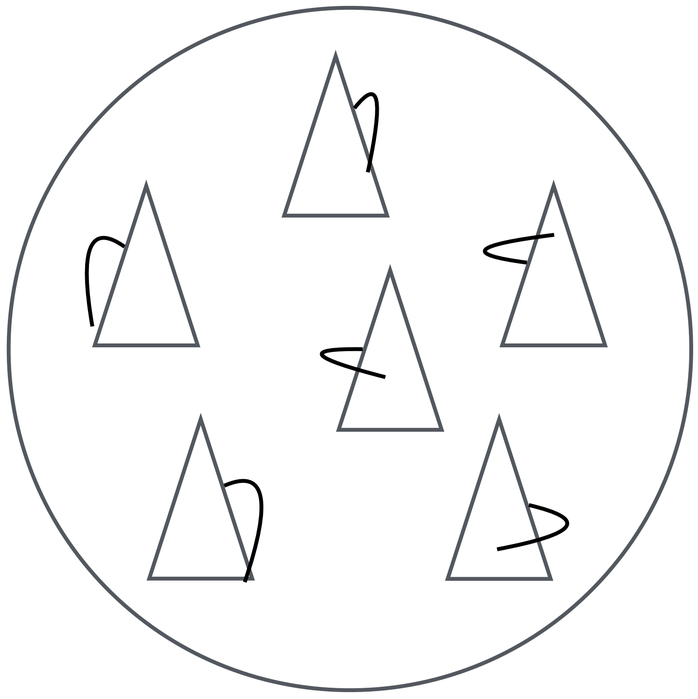
Language = Set of Trees

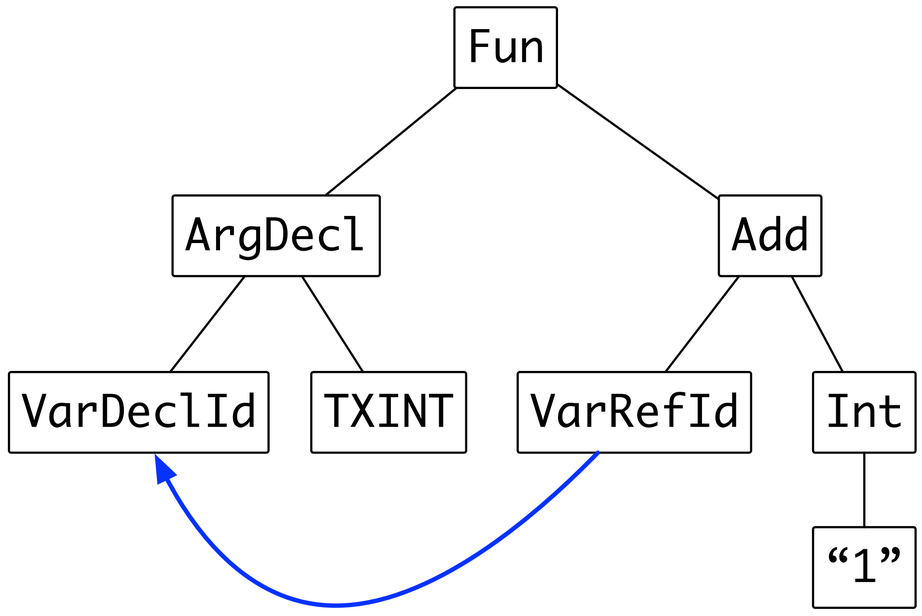




Tree is a convenient interface for transforming programs

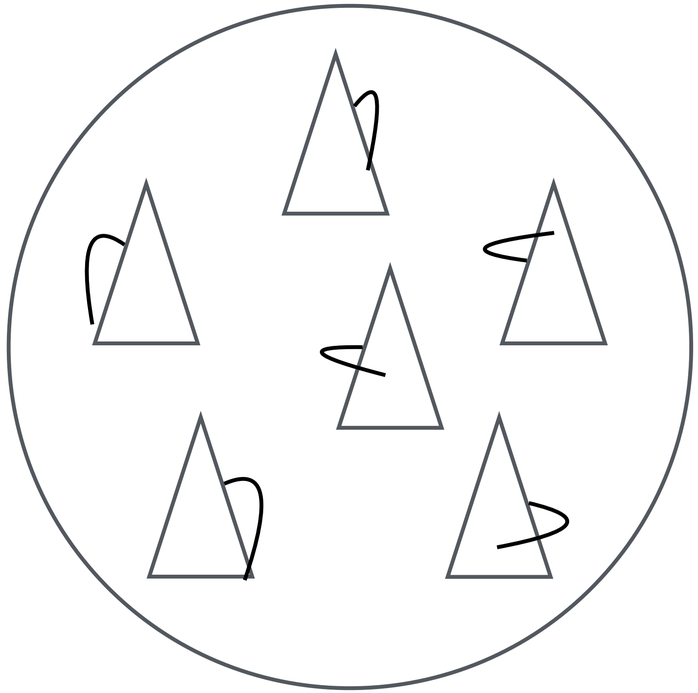
Language = Set of *Graphs*

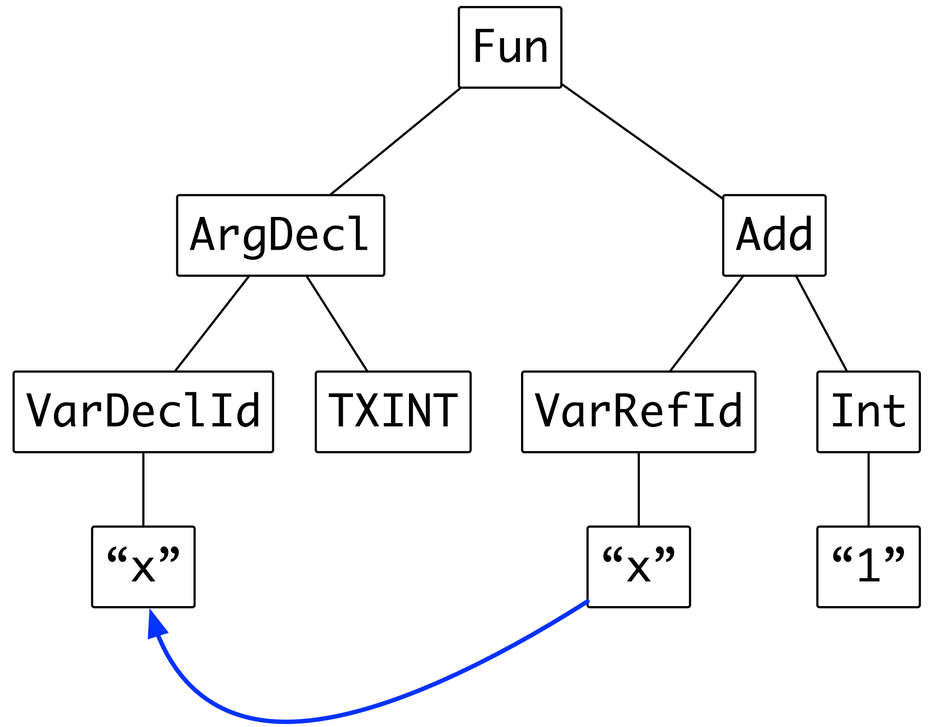




Edges from references to declarations

Language = Set of *Graphs*



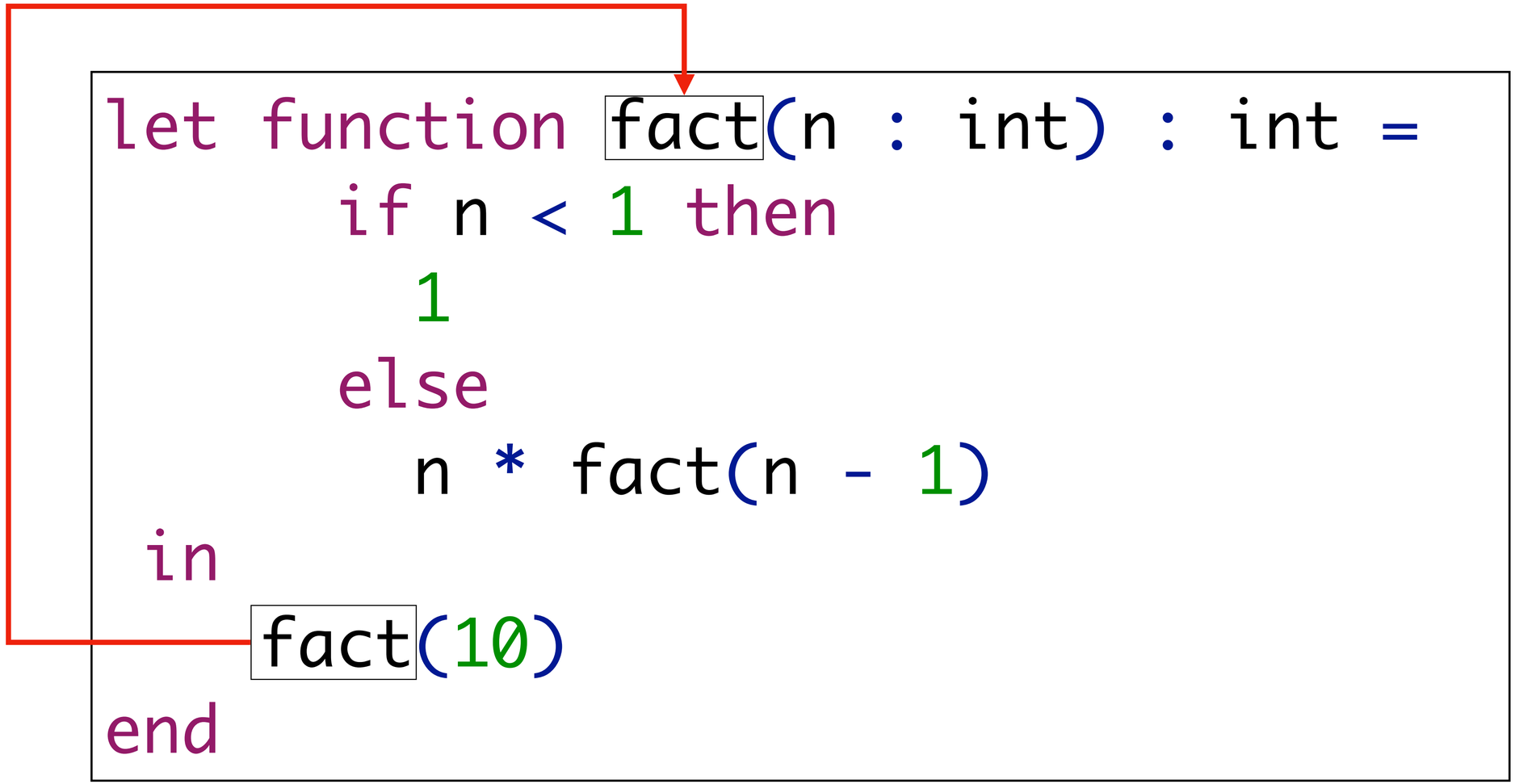


Names are placeholders for edges in linear / tree representation

Variables

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

Function Calls



Nested Scopes

(Shadowing)

```
function prettyprint(tree: tree) : string =
let
  var output := ""

  function write(s: string) =
    output := concat(output, s)

  function show(n: int, t: tree) =
  let function indent(s: string) =
        (write("\n");
         for i := 1 to n
          do write(" ");
         output := concat(output, s))
   in if t = nil then indent(".")
      else (indent(t.key);
            show(n+1, t.left);
            show(n+1, t.right))
  end

 in show(0, tree);
    output
end
```

```
let
  type point = {
     x : int,
     y : int
  }
  var origin := point {
     x = 1,
     y = 2
   }
 in
  origin.x := 10;
  origin := nil
end
```

Type References

```
let
  type point = {
      x : int,
      y : int
  }
  var origin := point {
      x = 1,
      y = 2
  }
 in
  origin.x := 10;
  origin := nil
end
```

Record Fields

```
let
  type point = {
      x : int,
      y : int
  }
  var origin := point {
     x = 1,
     y = 2
  }
in
  origin.x := 10;
  origin := nil
end
```

Type Dependent Name Resolution

Name Resolution is Pervasive

Used in many different language artifacts

- compiler
- interpreter
- semantics
- IDE
- refactoring

Binding rules encoded in many different and ad-hoc ways

- symbol tables
- environments
- substitutions

No standard approach to formalization

- there is no BNF for name binding

No reuse of binding rules between artifacts

- how do we know substitution respects binding rules?

Representing Bound Programs

Approaches to representing binding within an (extended) AST

- Numeric indexing [deBruijn72]
- Higher-order abstract syntax [PfenningElliott88]
- Nominal logic approaches [GabbayPitts02]

Support binding-sensitive AST manipulation

But

- Do not say how to resolve identifiers in the first place!
- Can not represent ill-bound programs
- Tend to be biased towards lambda-like bindings

Name Binding Languages

TU Delft

How to define the name binding rules of a language

?

What is the BNF of name binding

?

Name Binding Languages

DSLs for specifying binding structure of a (target) language

- Ott [Sewell+10]
- Romeo [StansiferWand14]
- Unbound [Weirich+11]
- Caml [Pottier06]
- NaBL [Konat+12]

Generate code to do resolution and record results

What is the semantics of such a name binding language?

NaBL Name Binding Language

```
binding rules // variables

  Param(t, x) :
    defines Variable x of type t

  Let(bs, e) :
    scopes Variable

Bind(t, x, e) :
  defines Variable x of type t

Var(x) :
  refers to Variable x
```

Declarative specification

Abstracts from implementation

Incremental name resolution

But:

How to explain it to Coq?

What is the semantics of NaBL?

Declarative Name Binding and Scope Rules

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser

SLE 2012

NaBL Name Binding Language

```
binding rules // classes

  Class(c, _, _, _) :
   defines Class c of type ClassT(c)
   scopes Field, Method, Variable

  Extends(c) :
   imports Field, Method from Class c

  ClassT(c) :
   refers to Class c

  New(c) :
   refers to Class c
```

Especially:

What is the semantics of imports?

Declarative Name Binding and Scope Rules

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser

SLE 2012

Approach

What is semantics of binding language?

- the meaning of a binding specification for language L should be given by
- a function from L programs to their “resolution structures”

So we need

- a (uniform, language-independent) method for describing such resolution structures ...
- … that can be used to compute the resolution of each program identifier
- (or to verify that a claimed resolution is valid)

That supports

- Handle broad range of language binding features ...
- … using minimal number of constructs
- Make resolution structure language-independent
- Handle named collections of names (e.g. modules, classes, etc.) within the theory
- Allow description of programs with resolution errors

Separation of Concerns in Name Binding

Representation

- To conduct and represent the results of name resolution

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Separation of Concerns in Name Binding

Representation

- **Scope Graphs**

Declarative Rules

- To define name binding rules of a language

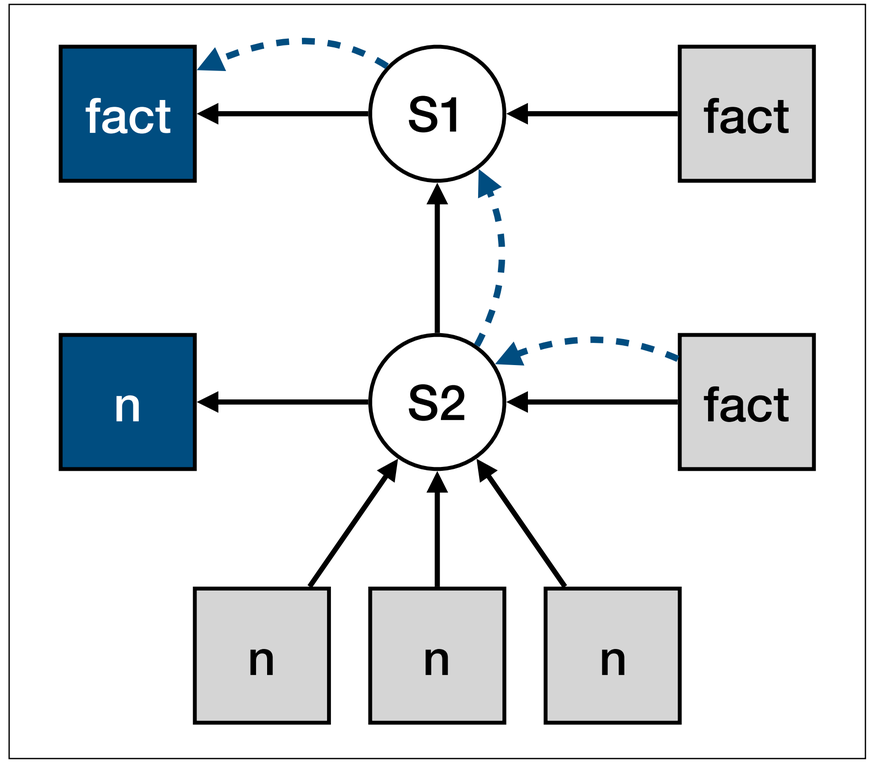
Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Program

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

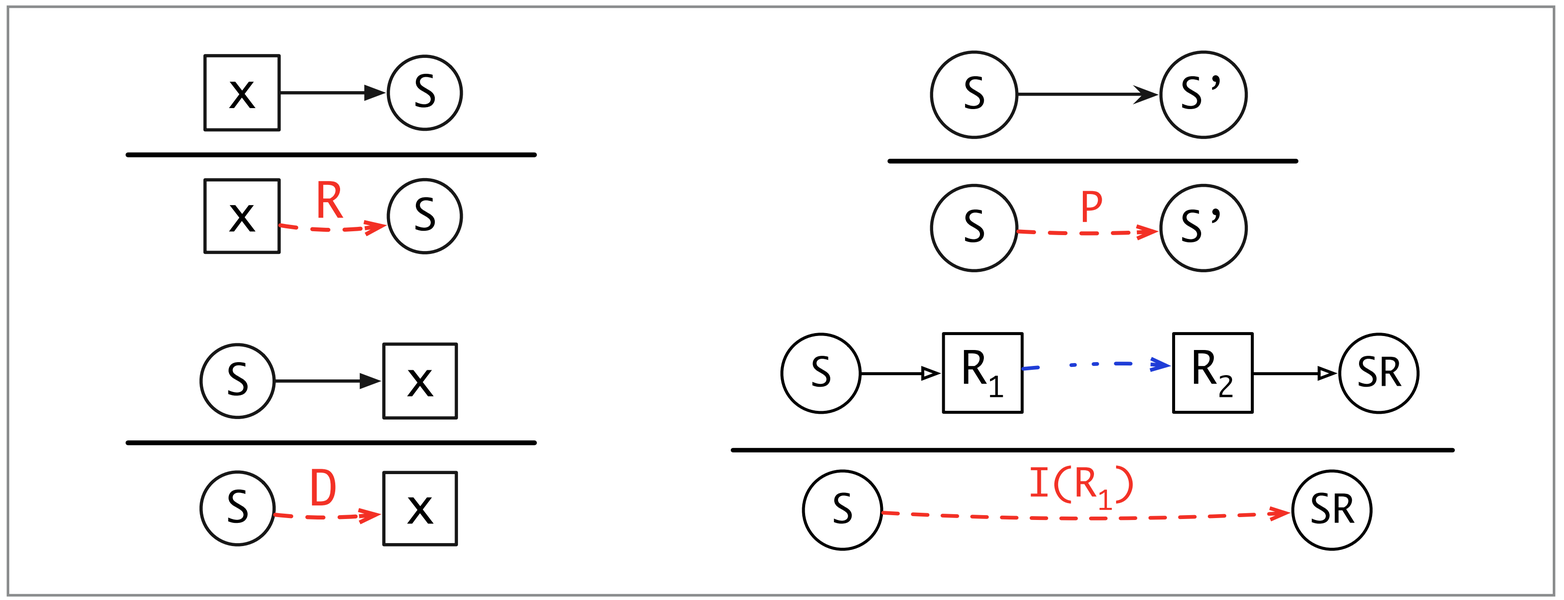
Scope Graph



Name Resolution

A Calculus for Name Resolution

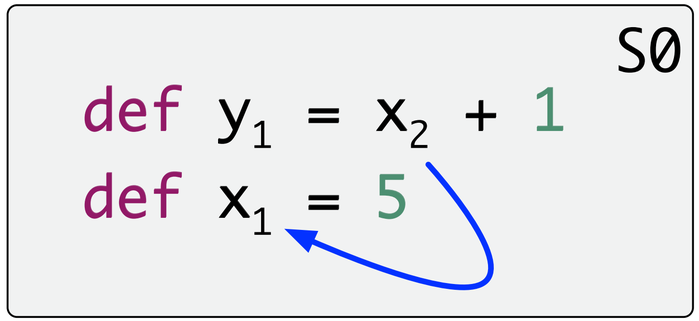
Scopes, References, Declarations, Parents, Imports

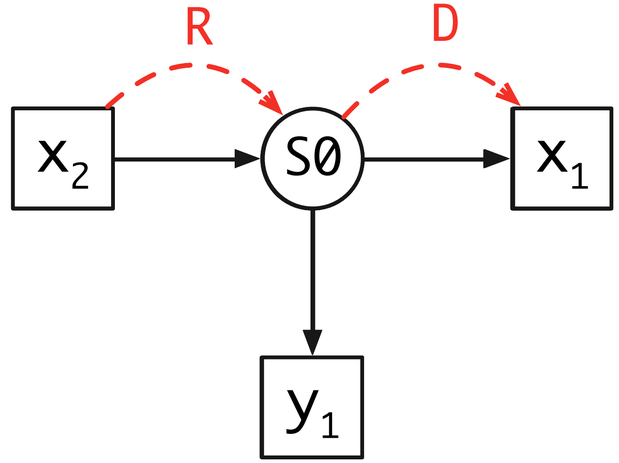


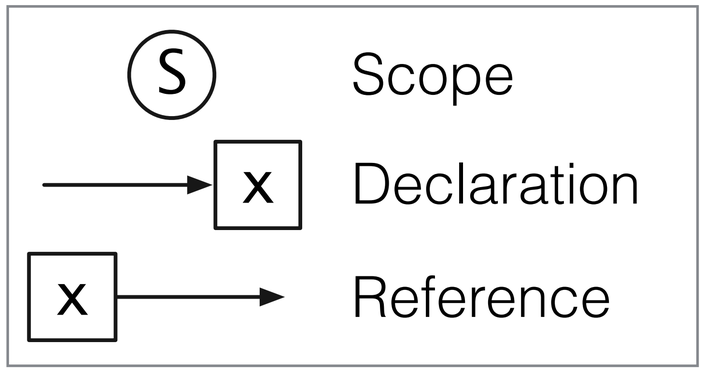
Path in scope graph connects reference to declaration

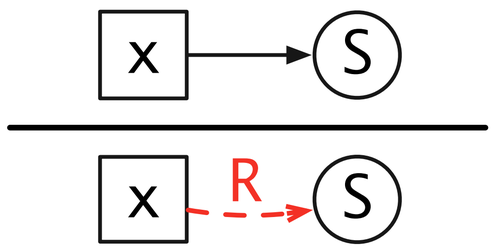
Neron, Tolmach, Visser, Wachsmuth
A Theory of Name Resolution
ESOP 2015

Simple Scopes

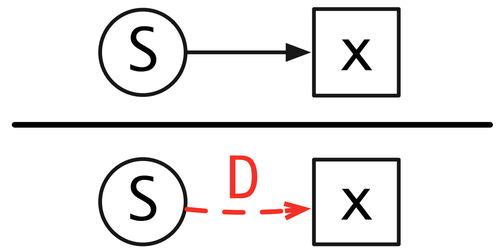






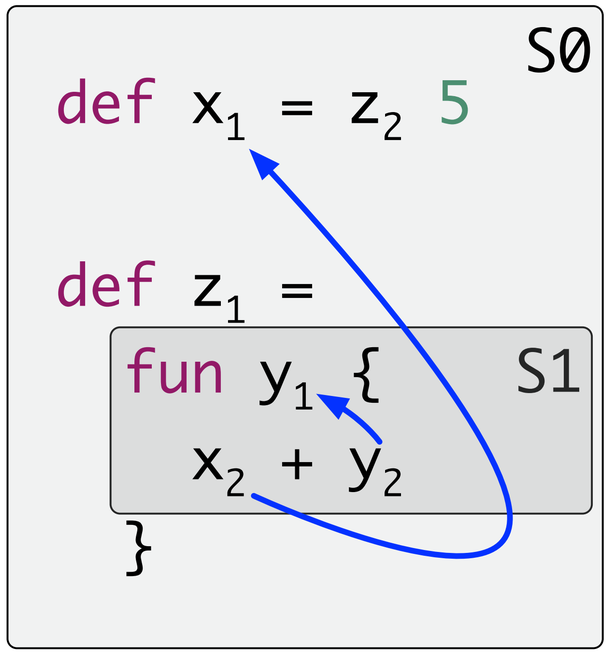


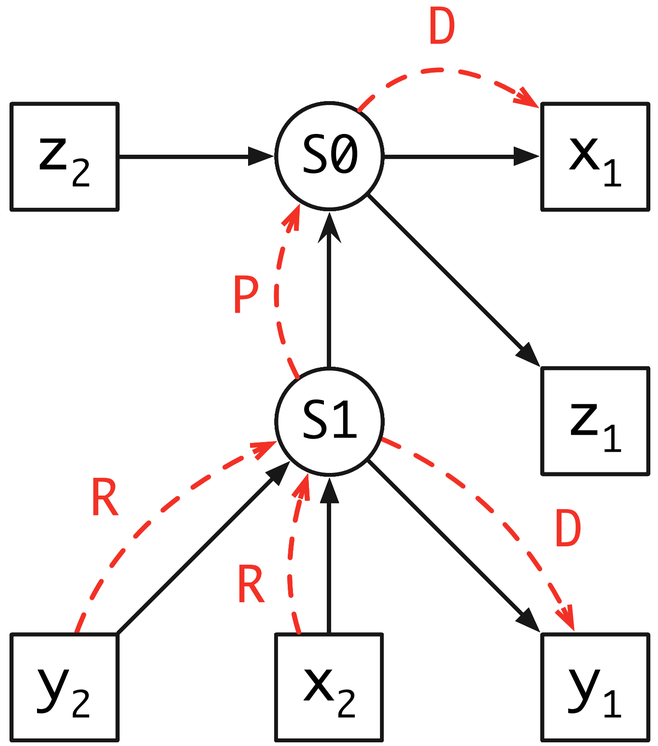
Reference Step

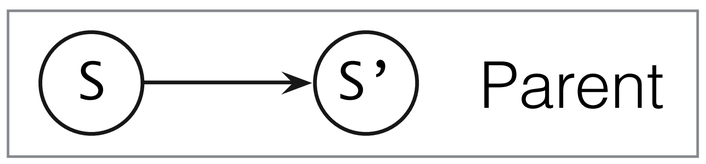


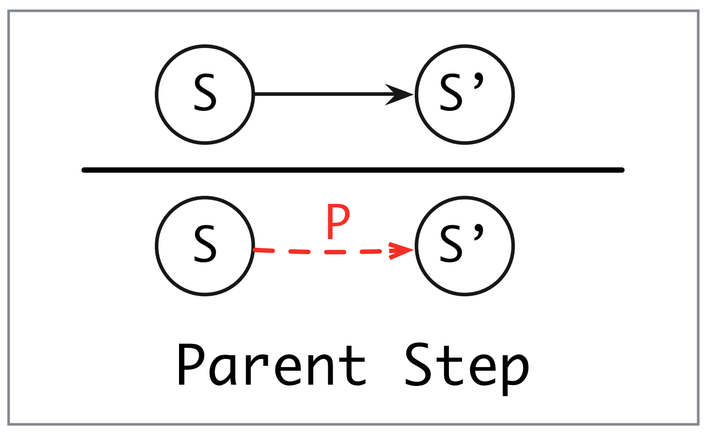
Declaration Step

Lexical Scoping

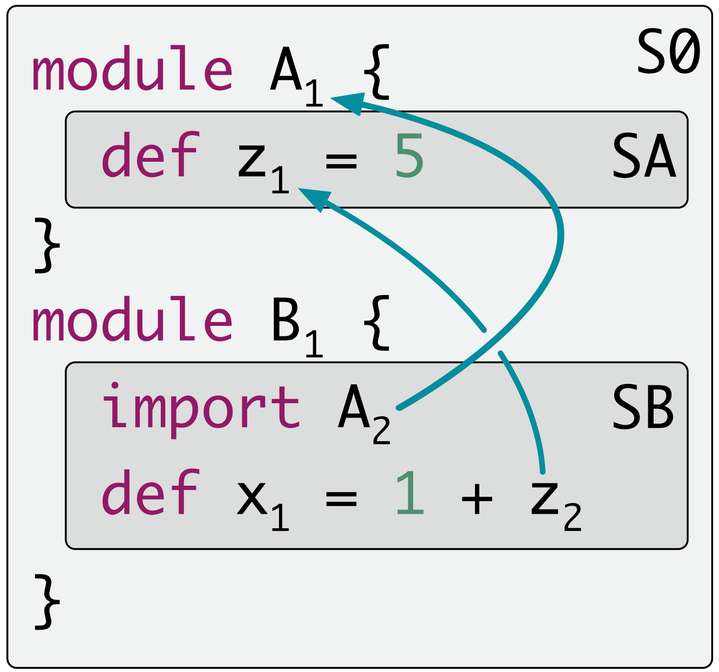


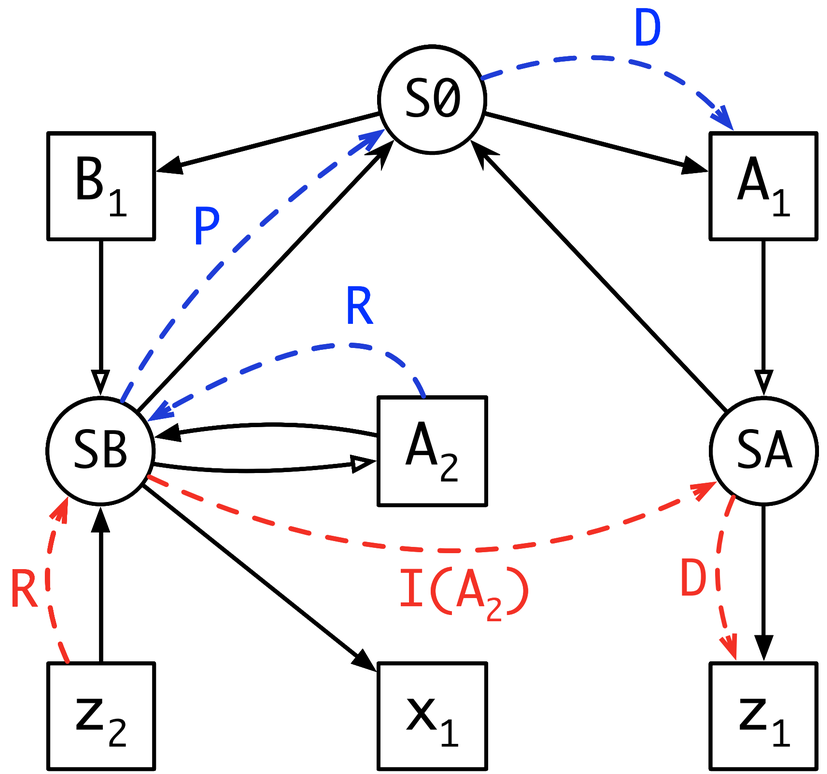


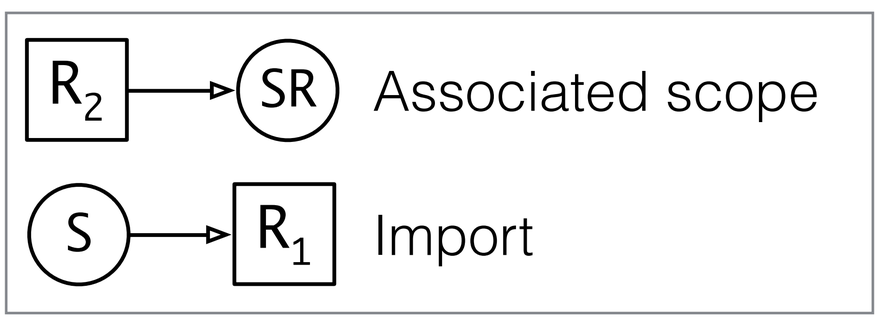


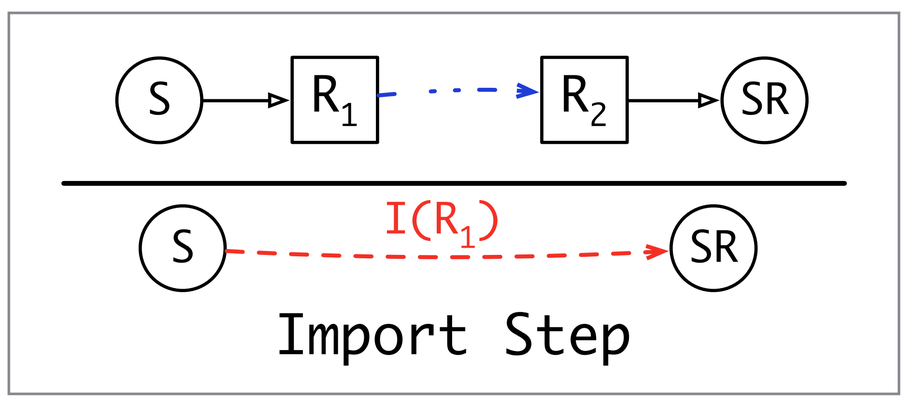


Imports


S0
module A_1 {
def z_1 = 5
SA
}
module B_1 {
import A_2
def x_1 = 1 + z_2
SB
}

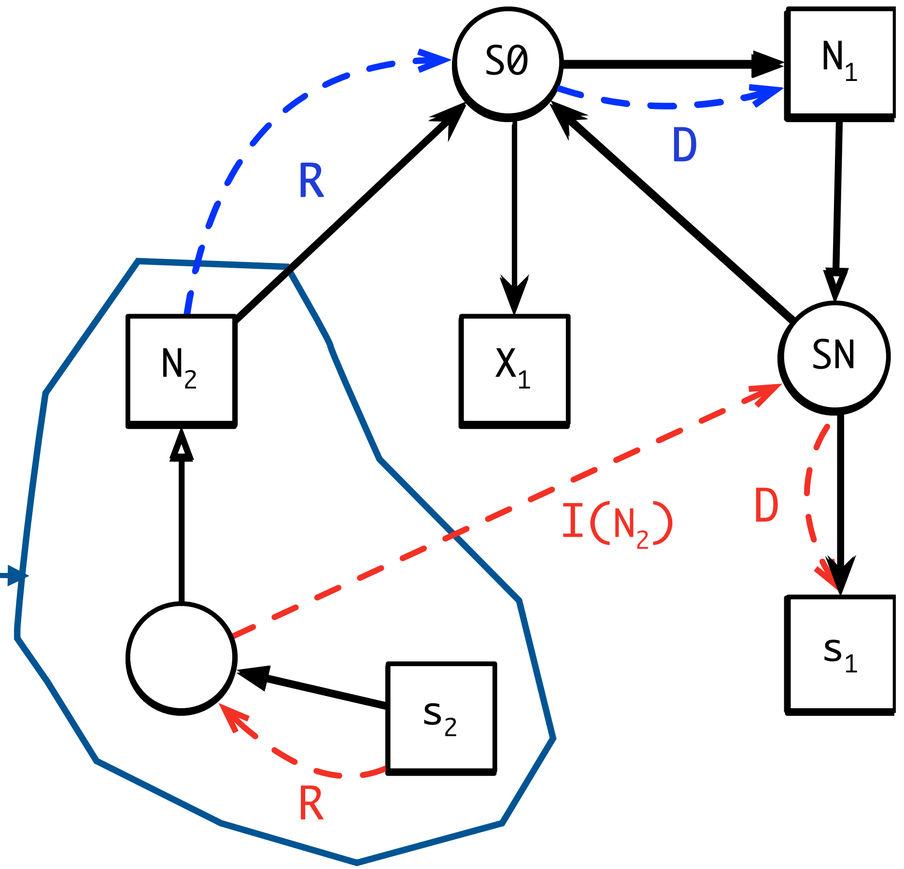

S0
B_1
A_1
P
D
R
SB
A_2
SA
R
$I(A_2)$
D
z_2
x_1
z_1


R_2
SR
Associated scope
S
R_1
Import


S
R_1
R_2
SR
$I(R_1)$
S
SR
Import Step

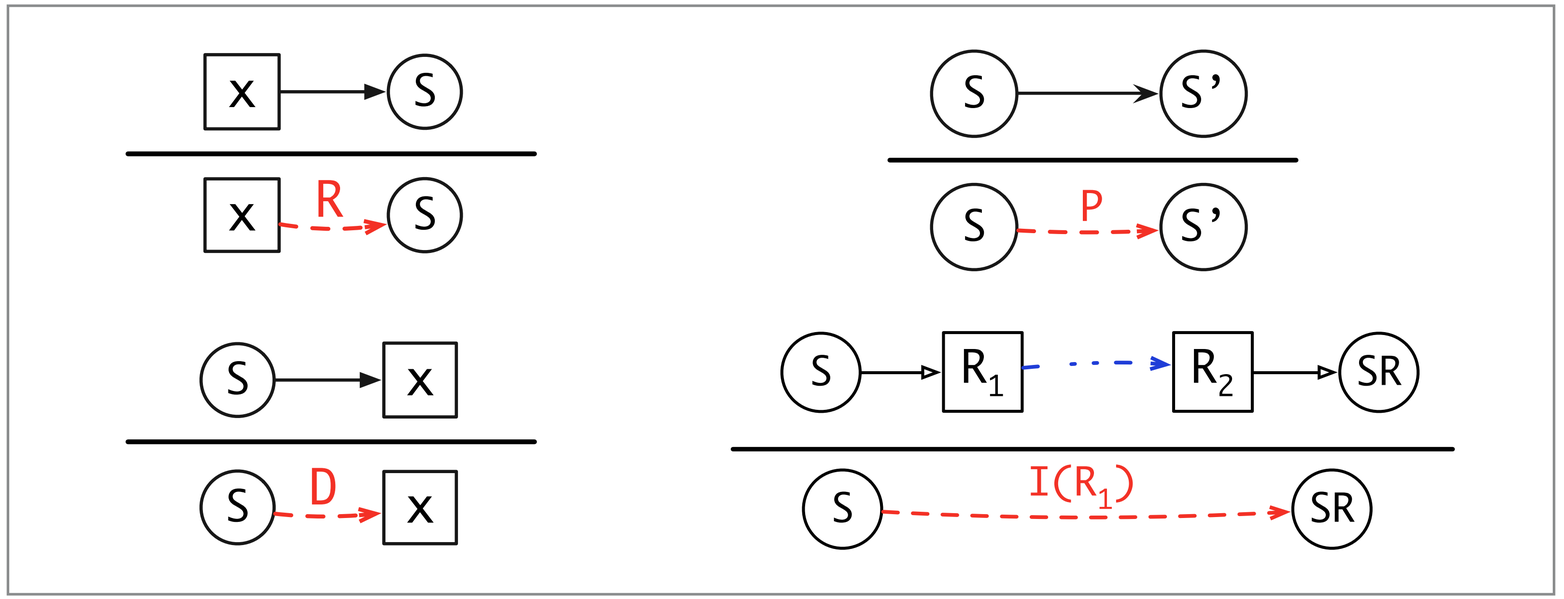
Qualified Names

```
module N1 {                                  S0
     def s1 = 5
}

module M1 {
     def x1 = 1 + N2.s2
}
```



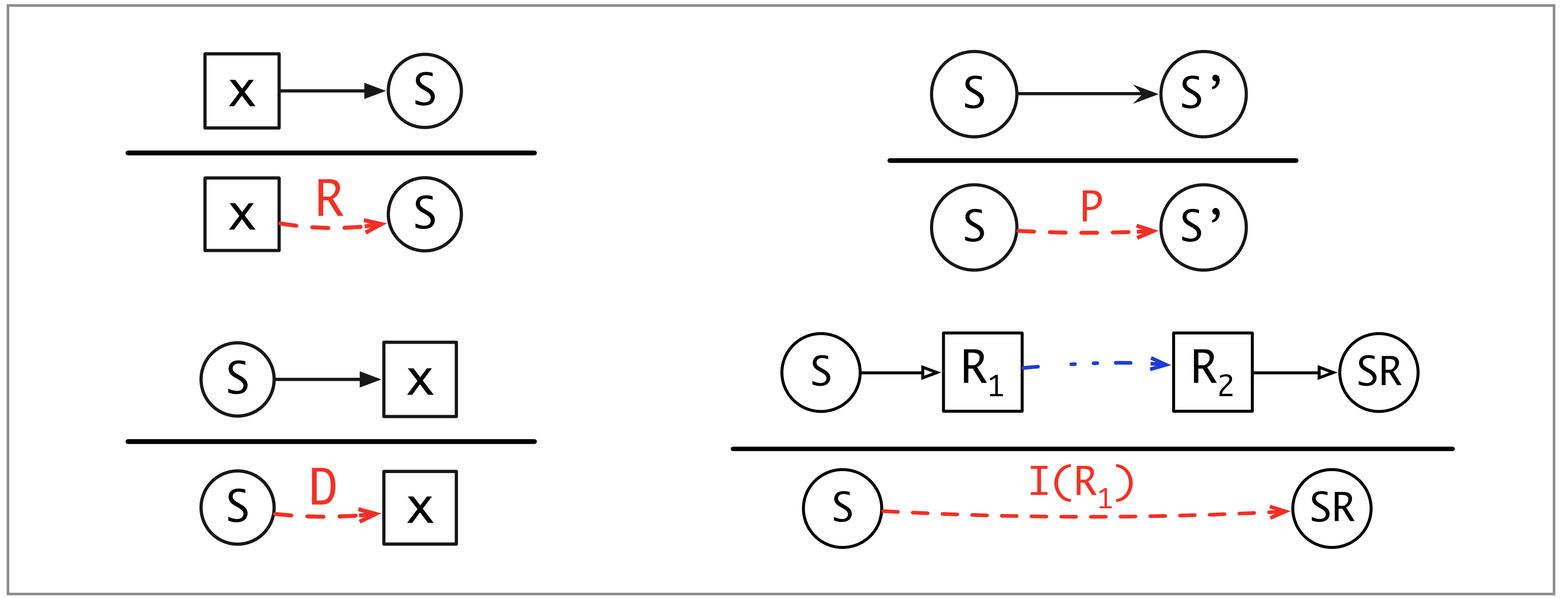
A Calculus for Name Resolution

Reachability of declarations from
references through scope graph edges



How about ambiguities?
References with multiple paths

A Calculus for Name Resolution



Reachability

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

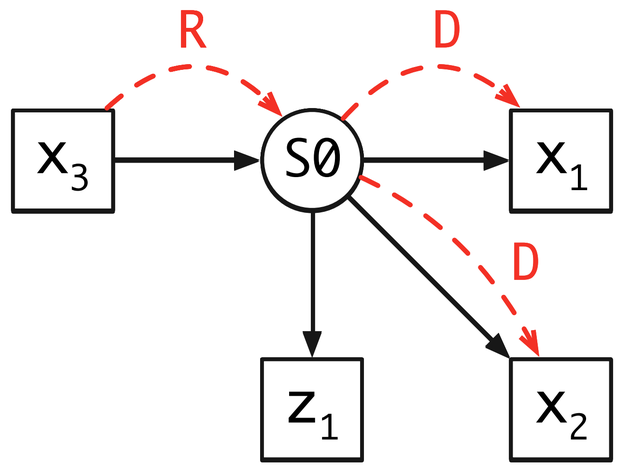
$$\frac{}{D < P.p} \qquad \frac{}{I(\_).p' < P.p}$$

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

Visibility

Ambiguous Resolutions

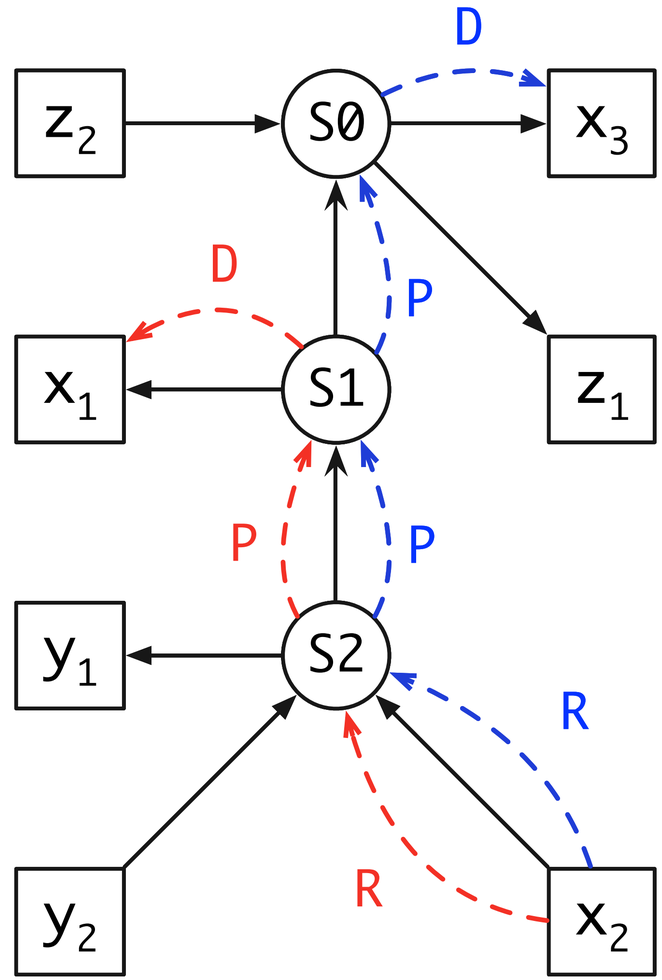
```
                                    S0
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```




```
match t with
  | A x | B x => …
```

Shadowing

```
def x3 = z2 5 7                    S0

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



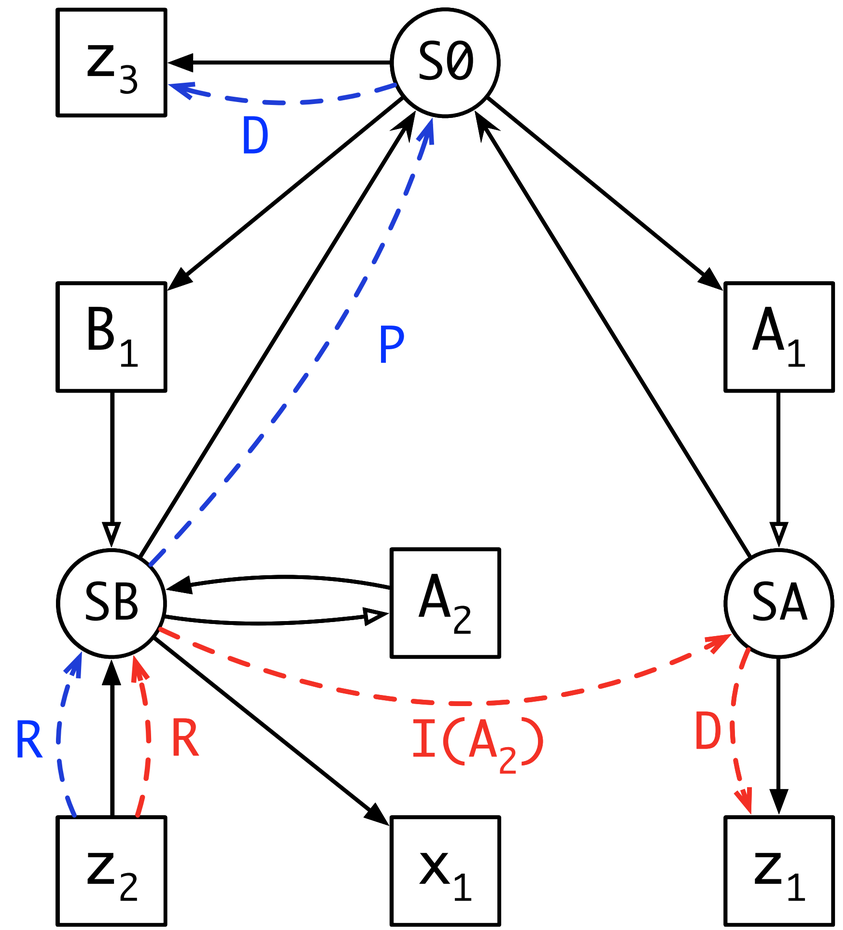
$$\frac{}{D < P.p}$$

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

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

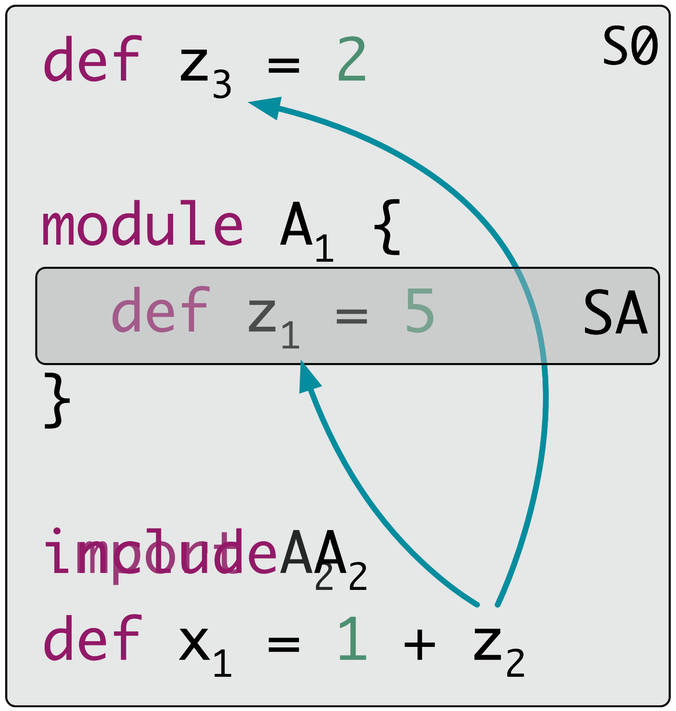
Imports shadow Parents

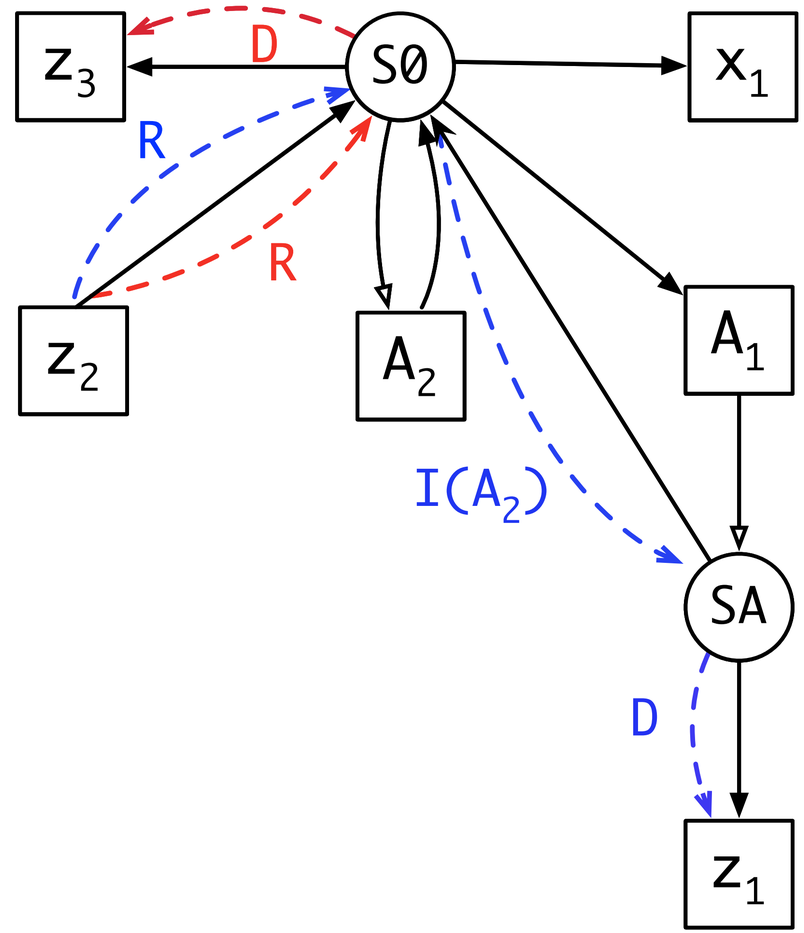
```
def z3 = 2                                  S0

module A1 {
  def z1 = 5                                SA
}
module B1 {
  import A2                                 SB
  def x1 = 1 + z2
}
```

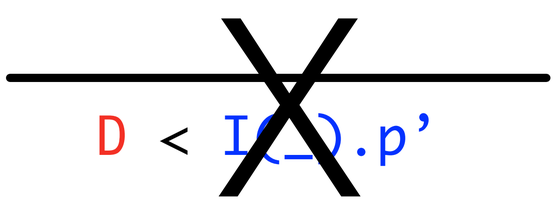


$$\frac{}{\texttt{I(\_).p'} < \texttt{P.p}} \quad \Rightarrow \quad \texttt{R.I(A2).D} < \texttt{R.P.D}$$

Imports vs. Includes

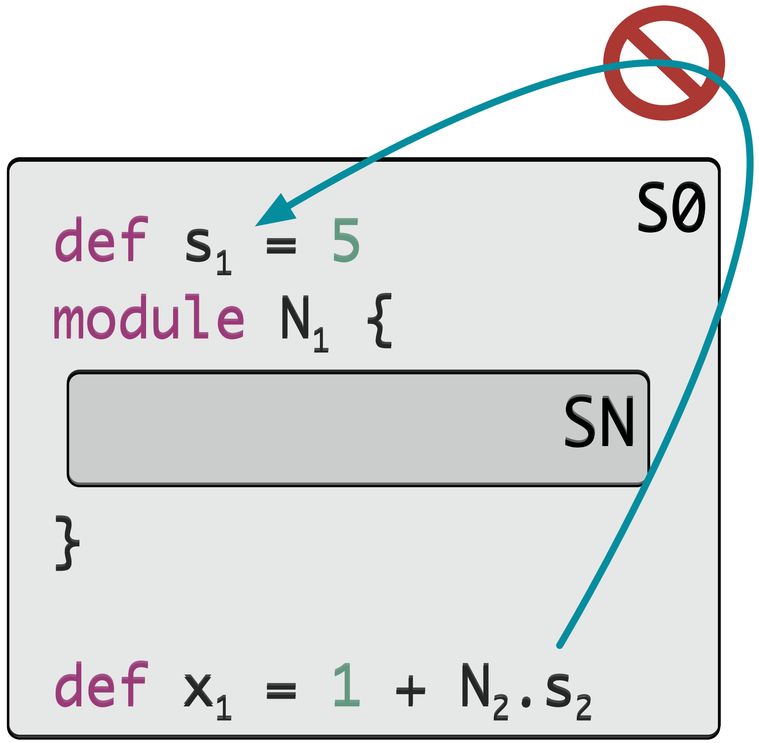


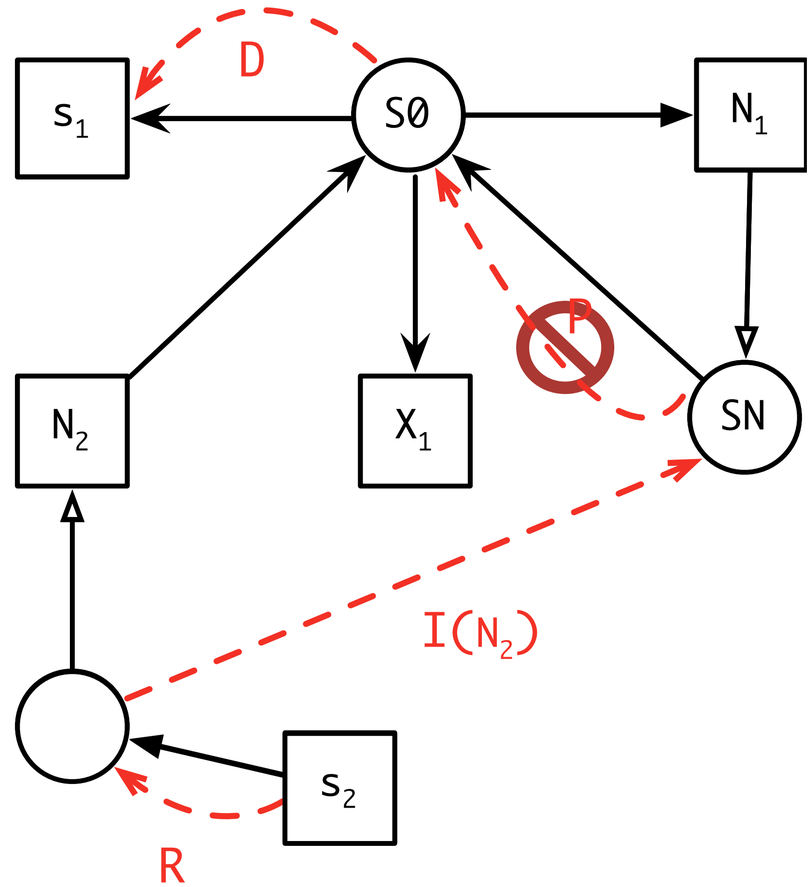




$\Rightarrow$ R.D < R.I(A_2).D

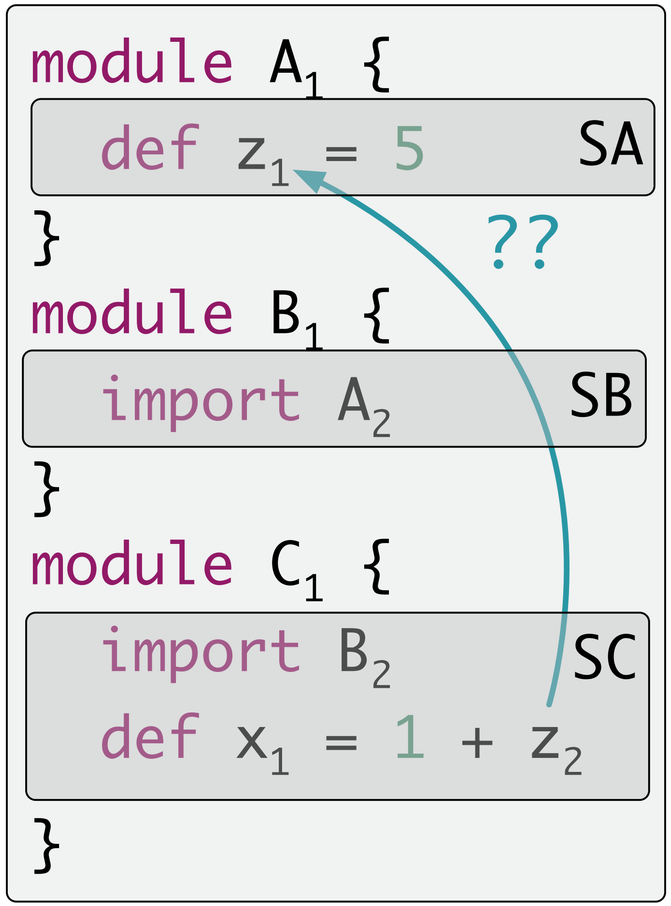
Import Parents

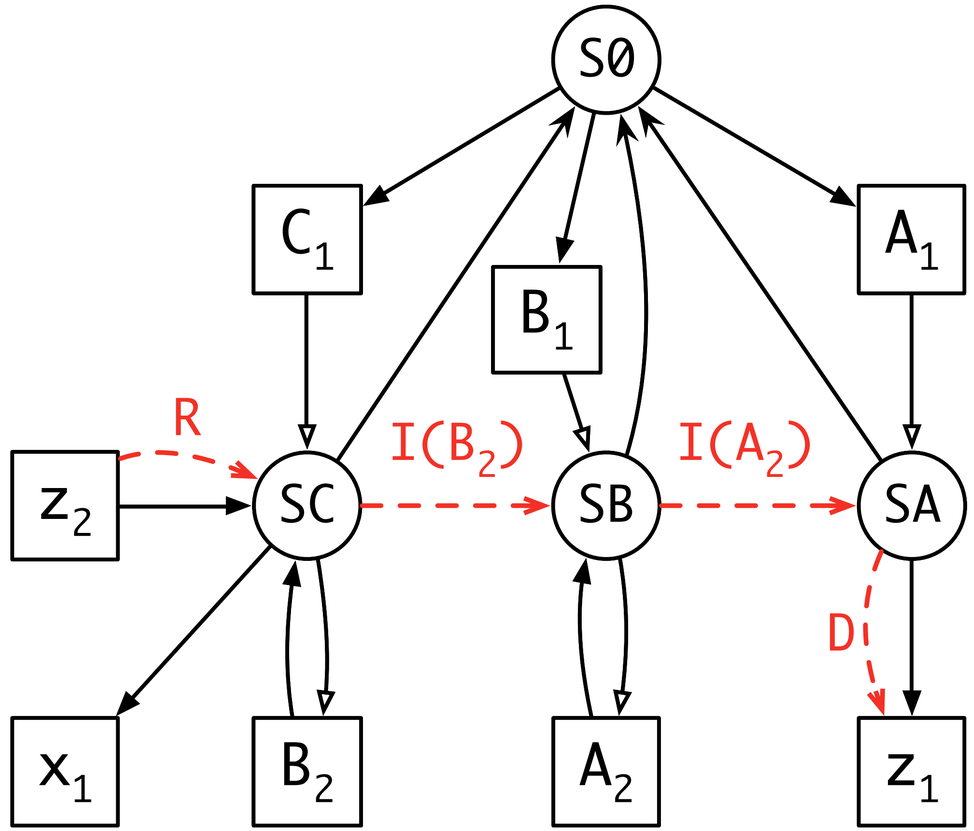




Well formed path: `R.P*.I(_)*.D`

Transitive vs. Non-Transitive

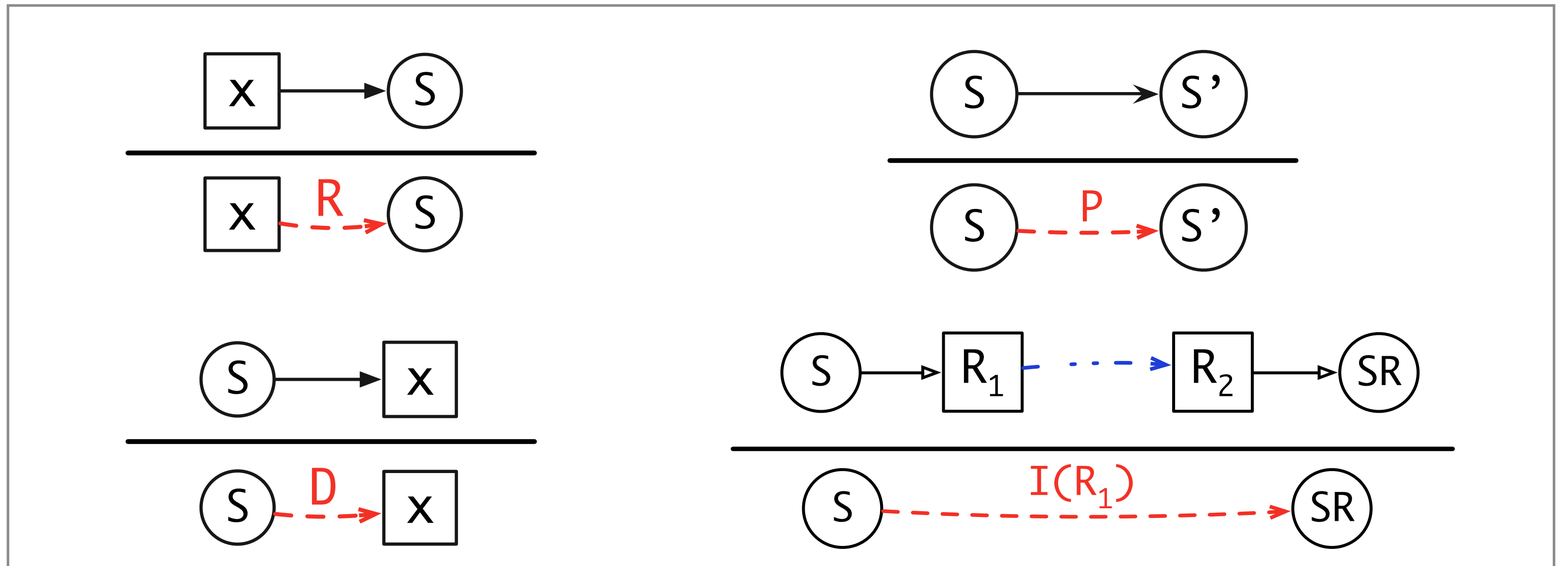




With transitive imports, a well formed path is `R.P*.I(_)*.D`

With non-transitive imports, a well formed path is `R.P*.I(_)?.D`

A Calculus for Name Resolution



Reachability

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

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

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

Visibility

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

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

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

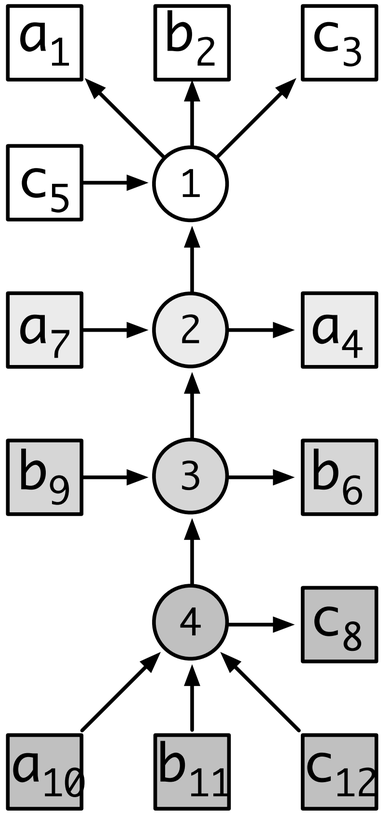
More Examples

From TUD-SERG-2015-001

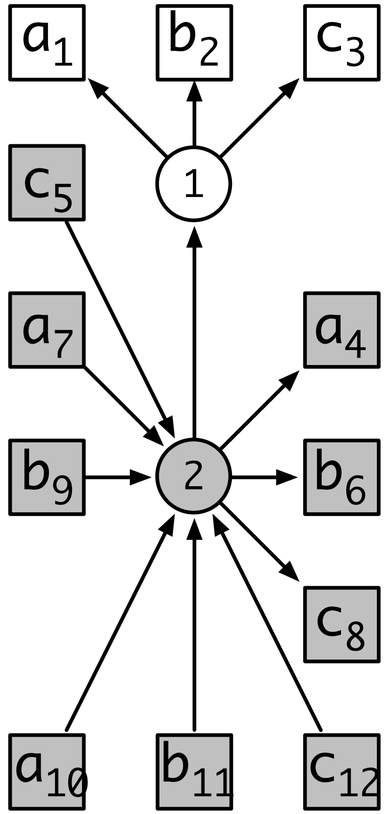
TU Delft

Let Bindings

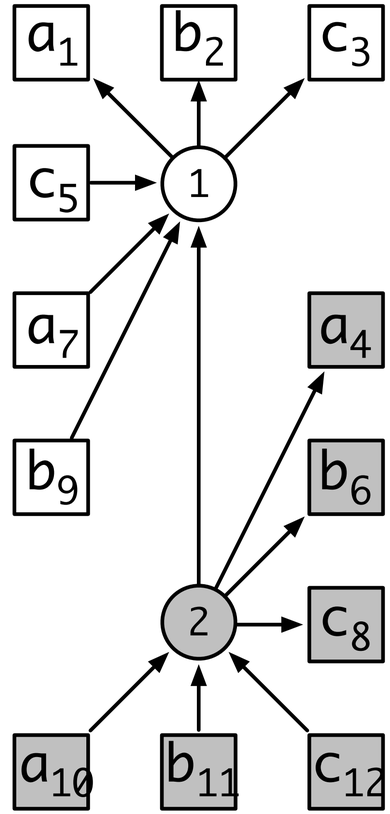
```
def a1 = 0
def b2 = 1
def c3 = 2

let
  a4 = c5
  b6 = a7
  c8 = b9
in
  a10+b11+c12
```




```
def a1 = 0
def b2 = 1
def c3 = 2

letrec
  a4 = c5
  b6 = a7
  c8 = b9
in
  a10+b11+c12
```

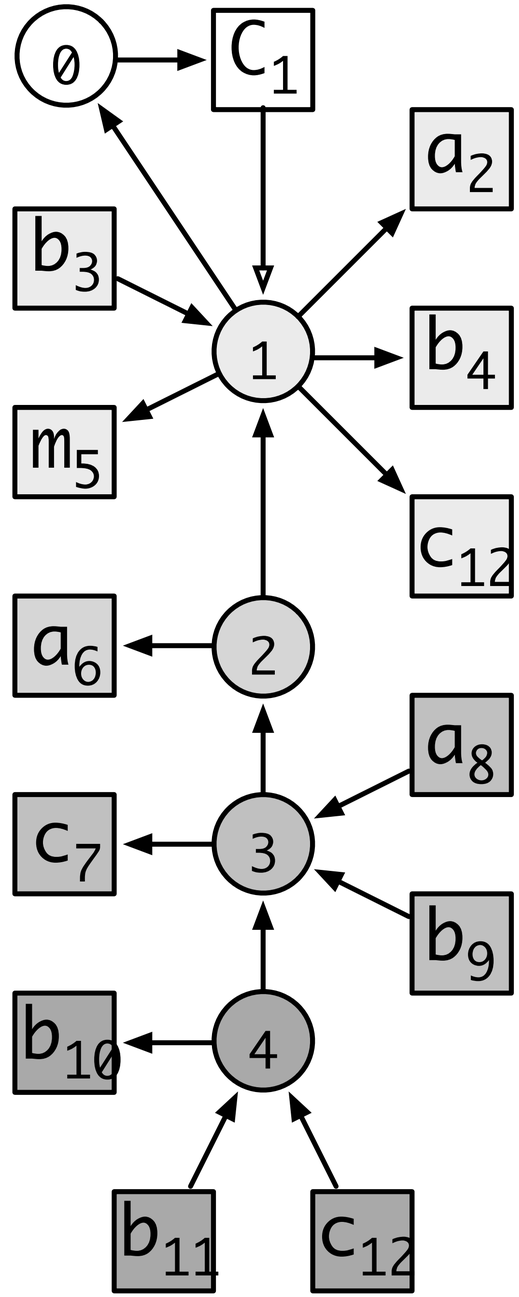



```
def a1 = 0
def b2 = 1
def c3 = 2

letpar
  a4 = c5
  b6 = a7
  c8 = b9
in
  a10+b11+c12
```



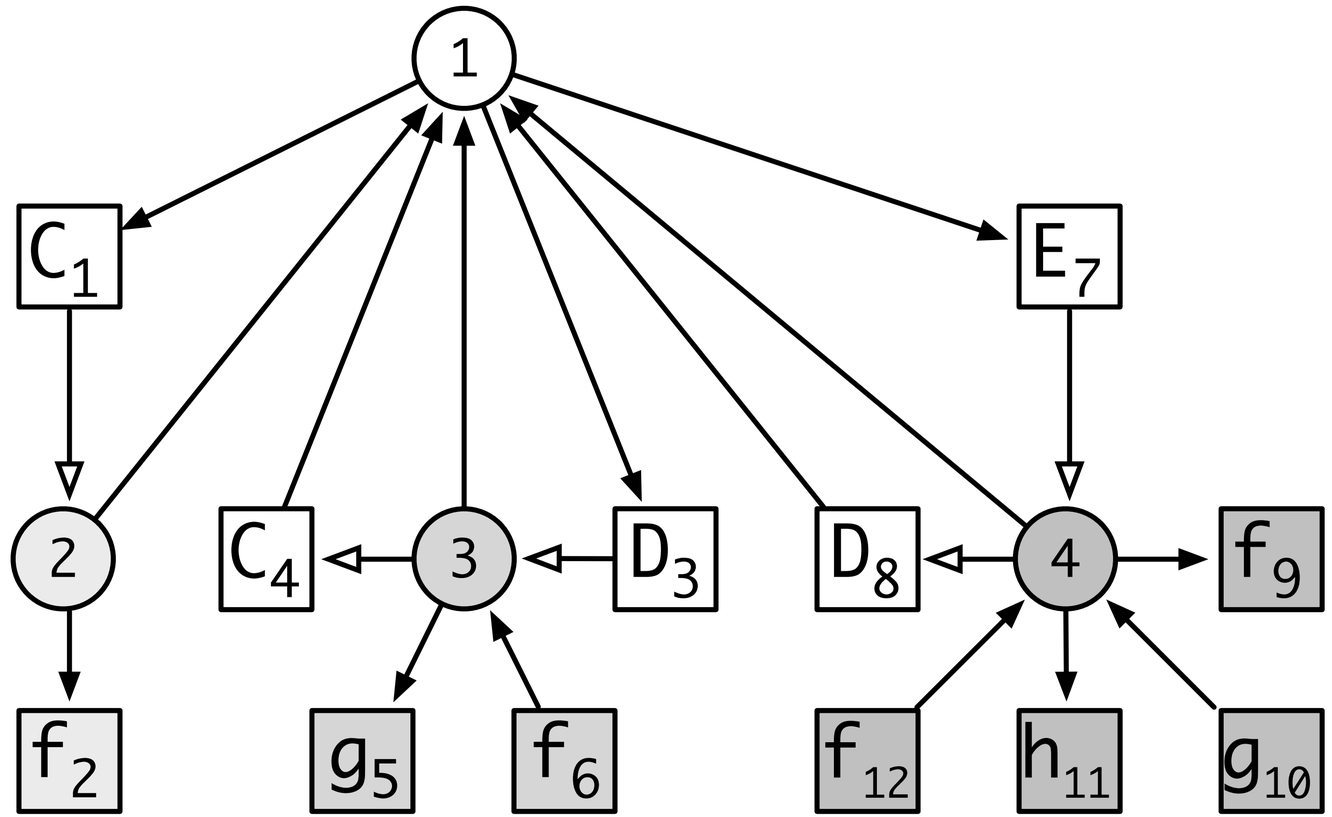
Definition before Use / Use before Definition

```
class C1 {
  int a2 = b3;
  int b4;
  void m5 (int a6) {

    int c7  = a8 + b9;
    int b10 = b11 + c12;

  }
  int c12;
}
```



Inheritance

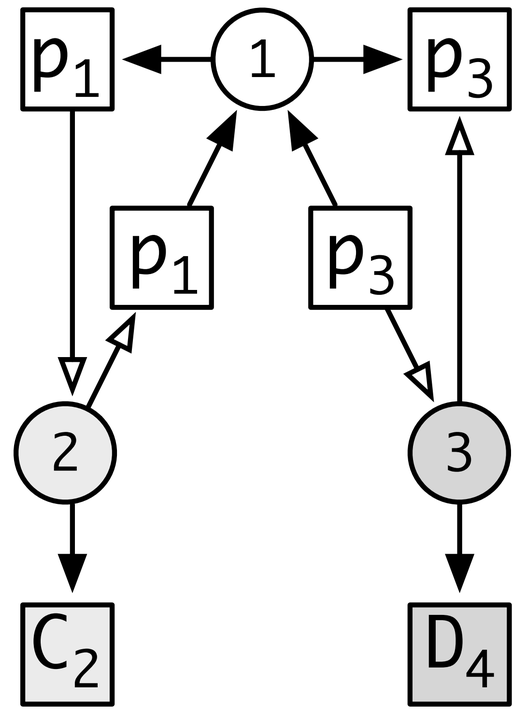
```
class C1 {
   int f2 = 42;
}
class D3 extends C4 {
   int g5 = f6;
}
class E7 extends D8 {
   int f9  = g10;
   int h11 = f12;
}
```



Java Packages

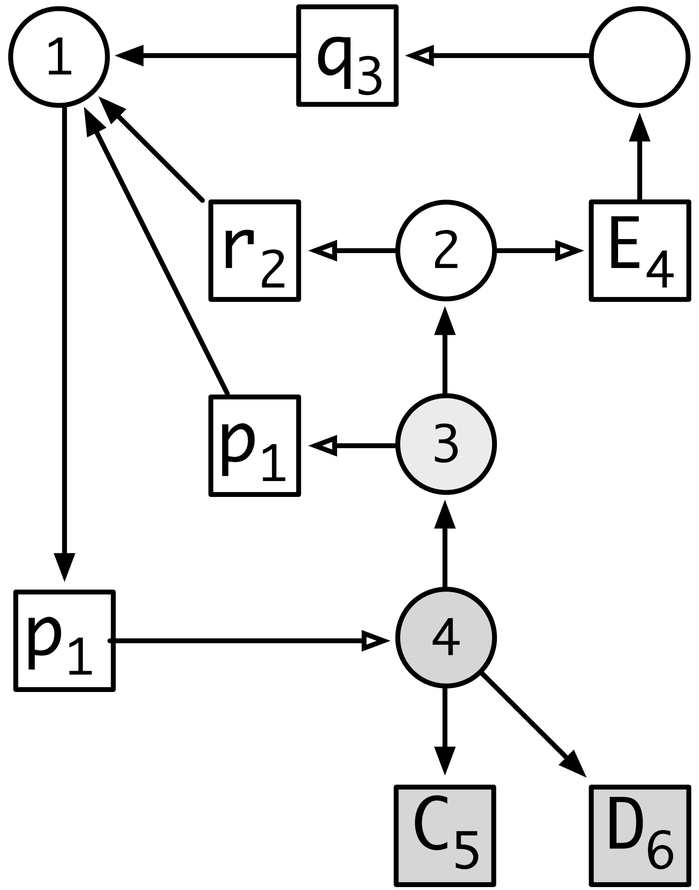
```
package p1;

class C2 {}
```

```
package p3;

class D4 {}
```



Java Import

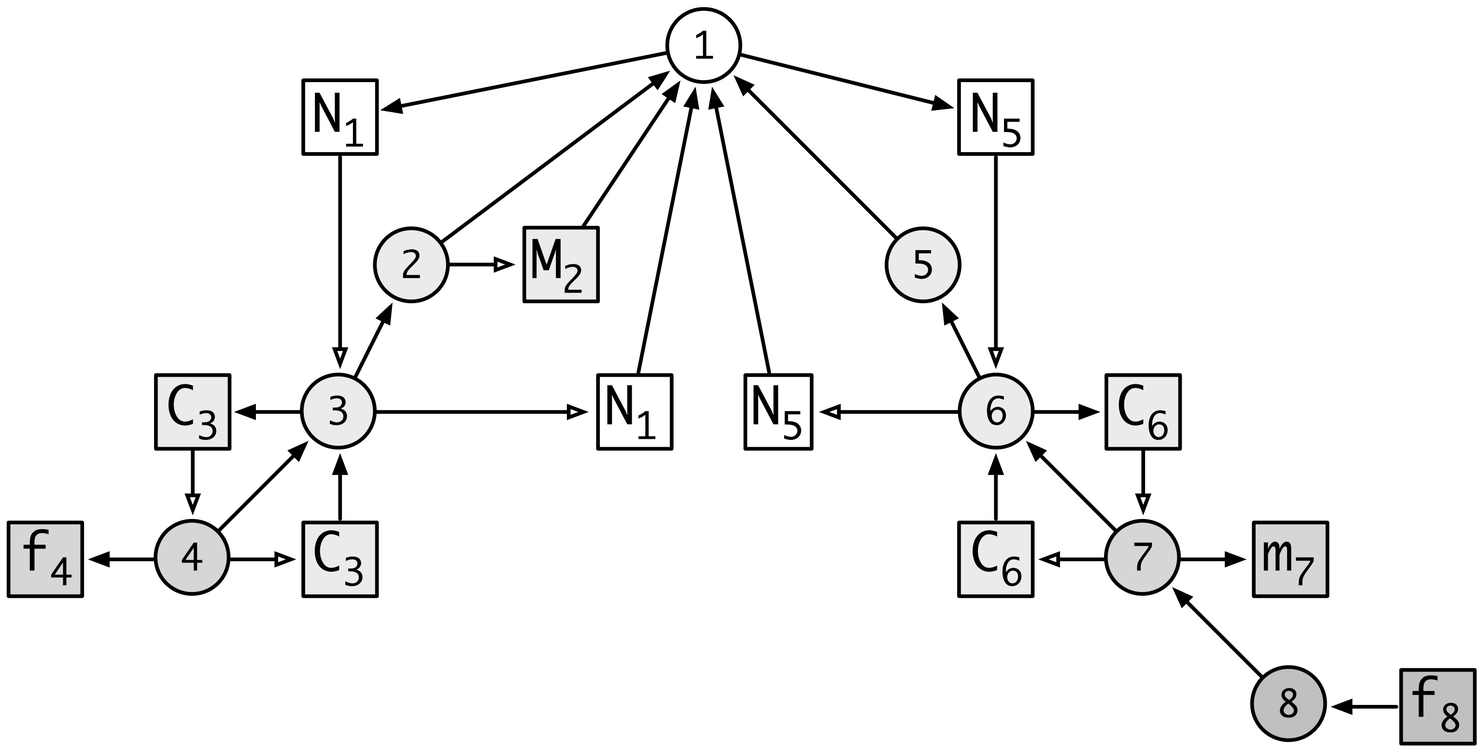
```
package p1;

imports r2.*;
imports q3.E4;

public class C5 {}

class D6 {}
```



C# Namespaces and Partial Classes

```
namespace N1 {
  using M2;

  partial class C3 {
    int f4;
  }
}
```

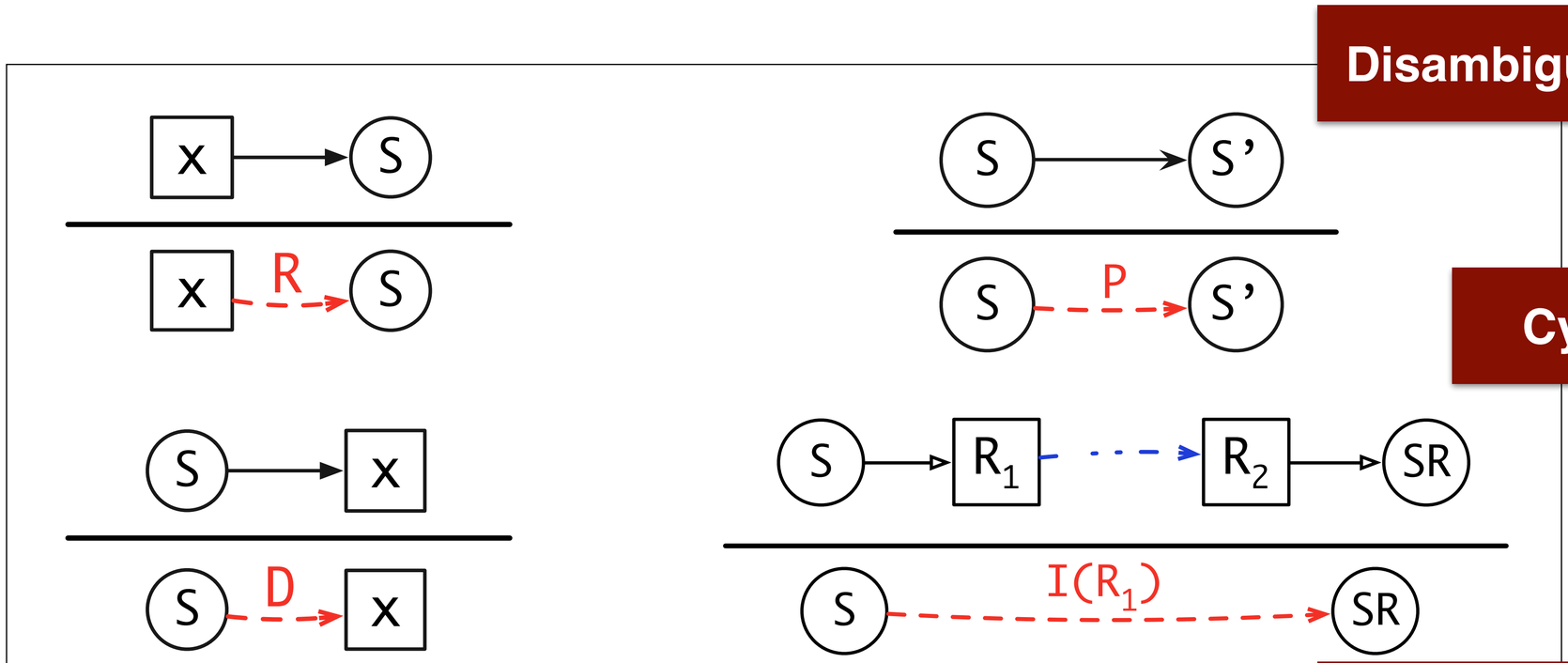
```
namespace N5 {
  partial class C6 {
    int m7() {
      return f8;
    }
  }
}
```



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

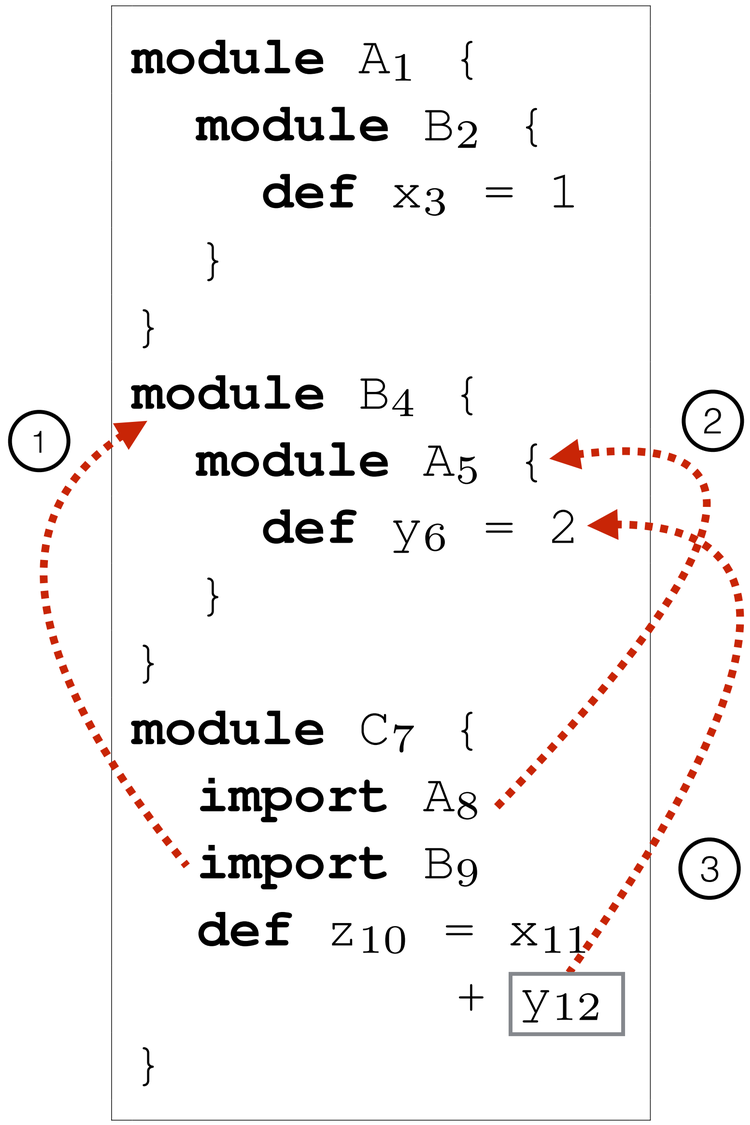
$$\dfrac{\dfrac{A_2^{\mathsf{D}}{:}S_{A_2} \in \mathcal{D}(S_{A_1}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{I}(S_{root}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{R}(S_{root}) \qquad A_1^{\mathsf{D}}{:}S_{A_1} \in \mathcal{D}(S_{root})}{A_4^{\mathsf{R}} \longmapsto A_1^{\mathsf{D}}{:}S_{A_1}}}{S_{root}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

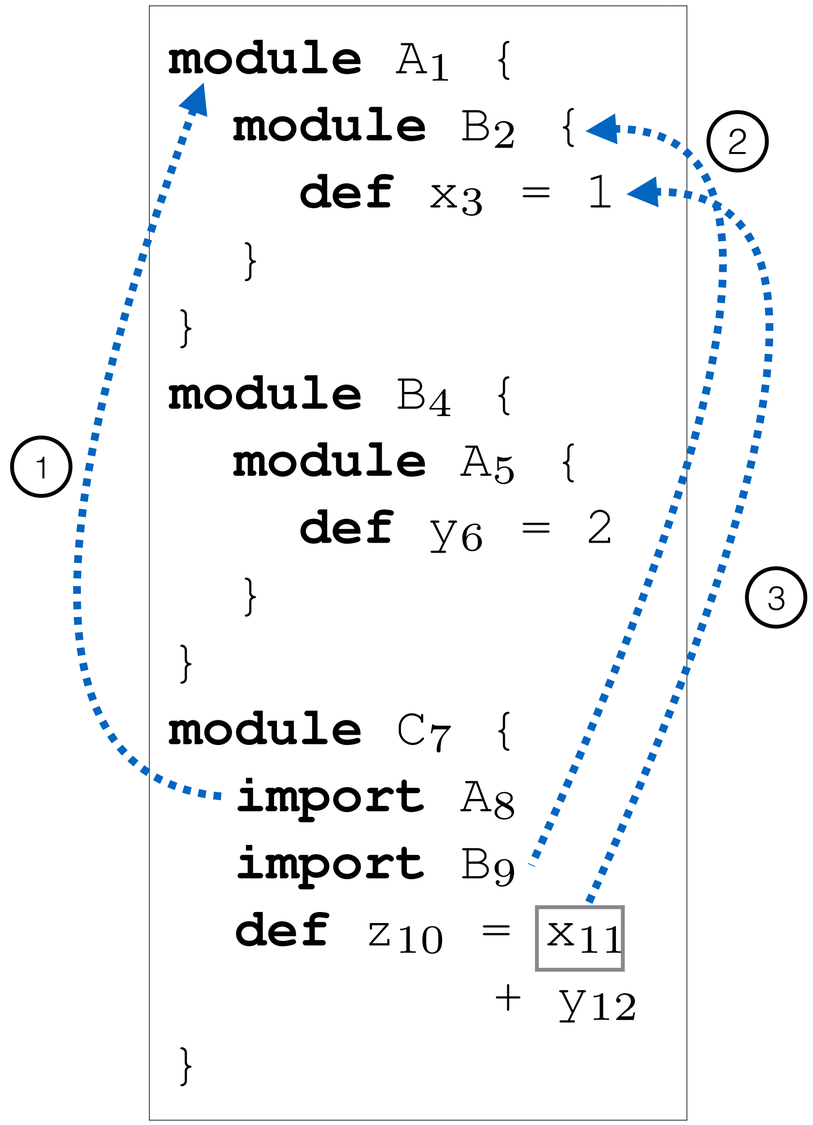
```
module A1 {
  module A2 {
    def a3 = ...
  }
}                    ?
import A4
def b5 = a6
```

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

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

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

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

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

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

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

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

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

Alpha Equivalence

Language-independent α-equivalence

Program similarity

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

Position equivalence

$$\frac{\texttt{x}_i \dashrightarrow \texttt{x}_{i'}}{i \overset{\texttt{P}}{\simeq} i'} \qquad \frac{i' \overset{\texttt{P}}{\simeq} i}{i \overset{\texttt{P}}{\simeq} i'} \qquad \frac{i \overset{\texttt{P}}{\simeq} i' \quad i' \overset{\texttt{P}}{\simeq} i''}{i \overset{\texttt{P}}{\simeq} i''} \qquad \frac{}{i \overset{\texttt{P}}{\simeq} i}$$

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

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

Closing

Summary: A Theory of Name Resolution

Representation: Scope Graphs

- Standardized representation for lexical scoping structure of programs
- Path in scope graph relates reference to declaration
- Basis for syntactic and semantic operations

Formalism: Name Binding Constraints

- References + Declarations + Scopes + Reachability + Visibility
- Language-specific rules map AST to constraints

Language-Independent Interpretation

- Resolution calculus: correctness of path with respect to scope graph
- Name resolution algorithm
- Alpha equivalence
- Mapping from graph to tree (to text)
- Refactorings
- And many other applications

Validation

We have modeled a large set of example binding patterns

- definition before use
- different let binding flavors
- recursive modules
- imports and includes
- qualified names
- class inheritance
- partial classes

Next goal: fully model some real languages

- In progress: Go, Rust, TypeScript
- Java, ML

Future Work

Scope graph semantics for binding specification languages

- starting with NaBL
- or rather: a redesign of NaBL based on scope graphs

Resolution-sensitive program transformations

- renaming, refactoring, substitution, …

Dynamic analogs to static scope graphs

- how does scope graph relate to memory at run-time?

Supporting mechanized language meta-theory

- relating static and dynamic bindings

Separation of Concerns in Name Binding

Representation

- To conduct and represent the results of name resolution

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Separation of Concerns in Name Binding

Representation

- **Scope Graphs**

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Separation of Concerns in Name Binding

Representation

- **Scope Graphs**

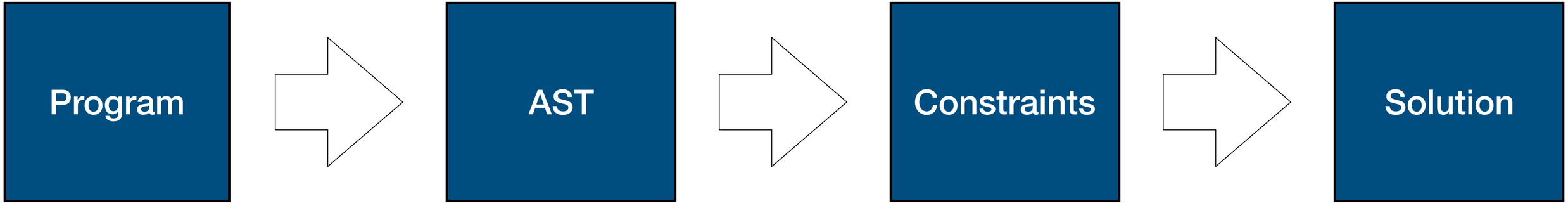
Declarative Rules

- **Scope & Type Constraint Rules [PEPM16]**

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Next: Constraint-Based Type Checkers



Parse

Generate Constraints

Resolve Constraints

Language Specific

Language Independent