Scope Graphs

A fresh look at name binding in programming languages

Eelco Visser

Curry On 2017
Barcelona
June 2017

TU Delft

```
/* A program to solve the 8-queens problem */

let
    var N := 8

    type intArray = array of int

    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

    function printboard() =
       (for i := 0 to N-1
    do (for j := 0 to N-1
         do print(if col[i]=j then " O" else " .");
        print("\n"));
         print("\n"))

    function try(c:int) =
(    if c=N
     then printboard()
     else for r := 0 to N-1
      do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
         then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
          col[c]:=r;
                try(c+1);
          row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)
)
 in try(0)
end
```

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

```
/* define valid recursive types */
let
/* define a list */
type intlist = {hd: int, tl: intlist}

/* define a tree */
type tree ={key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}

var lis:intlist := intlist { hd=0, tl=
nil }

in
   lis
end
```

A Language Design

Tiger by Andrew Appel, 1996

workspace - Java - org.metaborg.lang.tiger.example/examples/fac.tig - Eclipse

Quick Access

Packa JUnit

- > examples
  - arith.aterm
  - arith.tig
  - fac-error.pp.tig
  - fac-error.tig
  - fac.aterm
  - fac.des.tig
  - fac.pp.tig
  - > fac.tig
  - fact-anf.aterm
  - fact-anf.tig
  - fact-anf2.tig
  - for.tig
  - incomplete.tig
  - let-binding.tig
  - list-type.tig
  - nested.tig
  - point.aterm
  - point.pp.tig
  - point.tig
  - queens.aterm
  - queens.tig
  - rec-types.aterm
  - rec-types.tig
  - record-errors.aterm
  - record-errors.tig
  - recursion.aterm
  - recursion.tig
  - redeclarations.tig
  - tiny.aterm
  - tiny.pp.tig
  - tiny.tig
  - type-dec.aterm
  - type-dec.tig
  - while.aterm
  - while.tig
- natives

queens.tig

```
/* A program to solve the 8-queens problem */

let
    var N := 8

    type intArray = array of int

    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

    function printboard() =
        print("\n"));
        print("\n"))

    function try(c:int) =
(    if c=N
     then printboard()
     else for r := 0 to N-1
       do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
          then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
            col[c]:=r;
                try(c+1);
            row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)
)
 in try(0)
end
```

fac.tig

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

test05.tig

```
/* define valid recursive types */
let
/* define a list */
type intlist = {hd: int, tl: intlist}

/* define a tree */
type tree ={key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}

var lis:intlist := intlist { hd=0, tl= nil }

in
    lis
end
```

Want: An Implementation

fac.tig - org.metaborg.lang.tiger.example/examples

workspace - Java - org.metaborg.lang.tiger.example/examples/fac.tig - Eclipse

Quick Access

Packa JUnit

> examples
- arith.aterm
- arith.tig
- fac-error.pp.tig
- fac-error.tig
- fac.aterm
- fac.des.tig
- fac.pp.tig
- > fac.tig
- fact-anf.aterm
- fact-anf.tig
- fact-anf2.tig
- for.tig
- incomplete.tig
- let-binding.tig
- list-type.tig
- nested.tig
- point.aterm
- point.pp.tig
- point.tig
- queens.aterm
- queens.tig
- rec-types.aterm
- rec-types.tig
- record-errors.aterm
- record-errors.tig
- recursion.aterm
- recursion.tig
- redeclarations.tig
- tiny.aterm
- tiny.pp.tig
- tiny.tig
- type-dec.aterm
- type-dec.tig
- while.aterm
- while.tig

natives

queens.tig

```
1  /* A program to solve the 8-
2
3  let
4      var N := 8
5
6      type intArray = array of
7
8      var row
9      var col
10     var dia
11     var dia
12
13     functio
14        (for i := 0 to N-1
15     do (for j := 0 to N-1
16           do print(if
17          print("\n"));
18          print("\n"))
19
20     function try(c:int
21  (     if c=N
22        then printboard()
23        else for r :=
24          do if row[r]
25             then (row
26               col[c]:
27                   try
28               row[r]:
29  )
30   in try(0)
31  end
32
33
```

fac.tig

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

st05.tig

```
1  /* define valid recursive types */
2  let
3  /* define a list */
4  type intlist = {hd: int, tl: intlist}
5
   efine a tree */
     tree ={key: int, children: treelist}
     treelist = {hd: tree, tl: treelist}

   lis:intlist := intlist { hd=0, tl= nil }
11
12 in
13     lis
14 end
15
```

fac.tig - org.metaborg.lang.tiger.example/examples

Parser

Type Checker

Compiler

Interpreter

Start: Syntax

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

Parser

Context-Free Grammar

```
module Functions

imports Identifiers
imports Types

context-free syntax

  Dec.FunDecs = <<{FunDec "\n"}+>> {longest-match}

  FunDec.ProcDec = <
    function <Id>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <Id>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >

  FArg.FArg = <<Id> : <Type>>

  Exp.Call = <<Id>(<{Exp ", "}*>)>
```

Abstract Syntax

```
Mod(
  Let(
    [ FunDecs(
        [ FunDec(
            "fact"
          , [FArg("n", Tid("int"))]
          , Tid("int")
          , If(
              Lt(Var("n"), Int("1"))
            , Int("1")
            , Seq(
                [ Times(
                    Var("n")
                  , Call("fact", [Minus(Var("n"), Int("1"))])
                  )
                ]
              )
            )
        ]
      )
    ]
  , [Call("fact", [Int("10")])]
  )
)
```

Quick Access

Packa JUnit

- editor
- src
- src-gen
- syntax
  - Arrays.sdf3
  - ATerms.sdf3
  - Base.sdf3
  - Bindings.sdf3
  - Control-Flow.sdf3
  - Functions.sdf3
  - Identifiers.sdf3
  - Numbers.sdf3
  - Records.sdf3
  - Strings.sdf3
  - Tiger.sdf3
  - Types.sdf3
  - Variables.sdf3
  - Whitespace.sdf3
- target
- trans
- dynsem.properties
- metaborg.yaml
- pom.xml
- org.metaborg.lang.tiger.eclipse
- org.metaborg.lang.tiger.eclipse
- org.metaborg.lang.tiger.eclipse
- > org.metaborg.lang.tiger.exam
- appel
- > examples
  - arith.aterm
  - arith.tig
  - fac-error.pp.tig
  - fac-error.tig
  - fac.aterm
  - fac.des.tig
  - fac.pp.tig
  - > fac.tig

Functions.sdf3 | Records.sdf3

```
1  module Functions
2
3  imports Identifiers
4  imports Types
5
6  context-free syntax
7
8    Dec.FunDecs = <<{FunDec "\n"}+>> {longest-match}
9
10   FunDec.ProcDec = <
11     function <Id>(<{FArg ", "}*>) =
12       <Exp>
13   >
14
15   FunDec.FunDec = <
16     function <Id>(<{FArg ", "}*>) : <Type> =
17       <Exp>
18   >
19
20   FArg.FArg = <<Id> : <Type>>
21
22   Exp.Call = <<Id>(<{Exp ", "}*>)>
```

fac.tig

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

fac.aterm

```
1  Mod(
2    Let(
3      [ FunDecs(
4          [ FunDec(
5              "fact"
6            , [FArg("n", Tid("int"))]
7            , Tid("int")
8            , If(
9                Lt(Var("n"), Int("1"))
10           , Int("1")
11           , Seq(
12               [ Times(
13                   Var("n")
14                 , Call("fact", [Minus(Var("n"), Int("1"))])
15                 )
16               ]
17             )
18           )
19         )
20       ]
21     )
22   ]
23 , [Call("fact", [Int("10")])]
24 )
25 )
```

Writable | Insert | 12 : 12

workspace - Java - org.metaborg.lang.tiger.example/examples/point.tig - Eclipse

Quick Access

Packa JUnit

- trans
- dynsem.properties
- metaborg.yaml
- pom.xml
- org.metaborg.lang.tiger.eclipse
- org.metaborg.lang.tiger.eclipse
- org.metaborg.lang.tiger.eclipse
- > org.metaborg.lang.tiger.exam
- appel
- > examples
  - arith.aterm
  - arith.tig
  - fac-error.pp.tig
  - fac-error.tig
  - fac.aterm
  - fac.des.tig
  - fac.pp.tig
  - > fac.tig
  - fact-anf.aterm
  - fact-anf.tig
  - fact-anf2.tig
  - for.tig
  - incomplete.tig
  - let-binding.tig
  - list-type.tig
  - nested.tig
  - point.aterm
  - point.pp.tig
  - point.tig
  - queens.aterm
  - queens.tig
  - rec-types.aterm
  - rec-types.tig
  - record-errors.aterm
  - record-errors.tig
  - recursion.aterm
  - recursion.tig

Functions.sdf3 | Records.sdf3 | *point.tig

```
1 let
2   type point = {x : int, y : int}
3   var origin : point := point { x = 1, y = 2 }
4  in origin.x;
5    origin := nil
6
7 end
8
```

- Seq
- Minus
- Eq
- Record
- Int
- If
- And
- Let
- NilExp
- IfThen
- Geq
- Var

```
if $Exp then
  $Exp
else
  $Exp
```

Syntactic Completion

*fac.tig

```
1 let function fact(n : int) : int =
2       if n < 1 then 1 else (n * fact(n - 1))
3  in fact(10)
4 en
```

Syntax Checking

Writable | Insert | 6 : 4

Separation of Concerns in Syntax Definition

Representation

- Parse Trees / Abstract Syntax Trees

Declarative Rules

- Context-free grammar + disambiguation

A language for talking about syntax

Language-Independent Tooling

- Parser generation
- Syntax aware editor
- Syntactic completion
- Formatter
- …

More in ECOOP Summer School lecture on Thursday afternoon

Parser ✓

Type Checker

Compiler

Interpreter

Parser ✓

Name Binding

Type Checker

Compiler

Interpreter

?

Name Binding

Variables

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

```
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

How to define the name binding rules of a language

?

What is the BNF of name binding

?

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

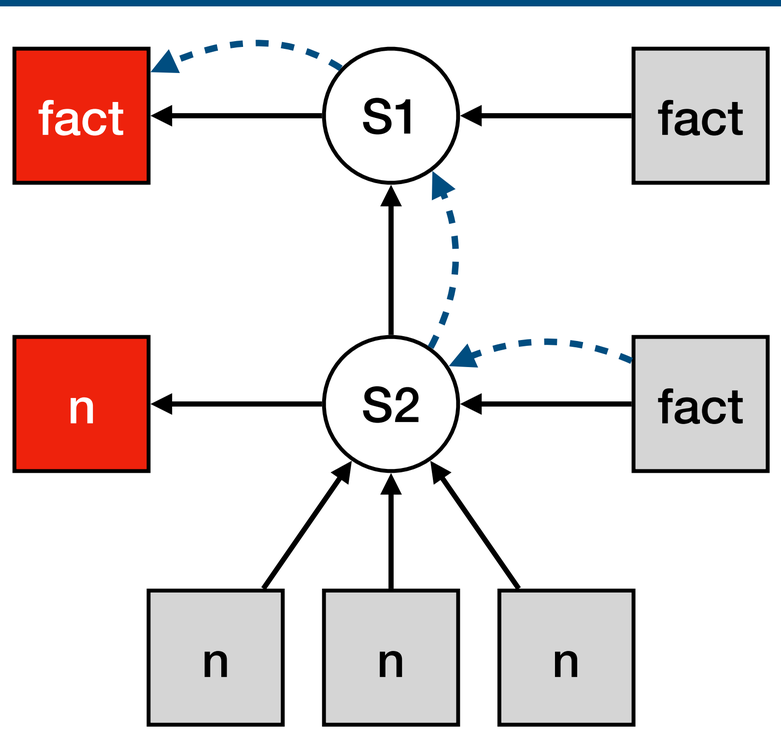
Language-Independent Tooling

- 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
```

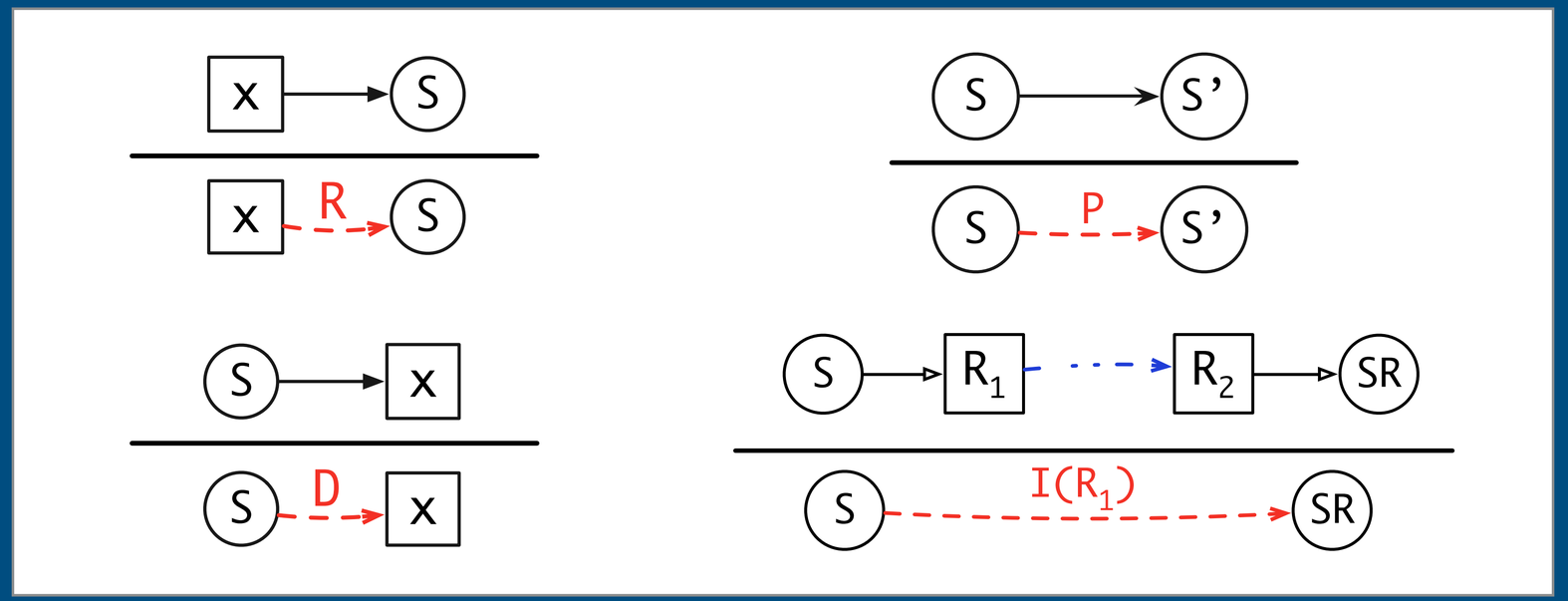
Scope Graph



Name Resolution

A Calculus for Name Resolution

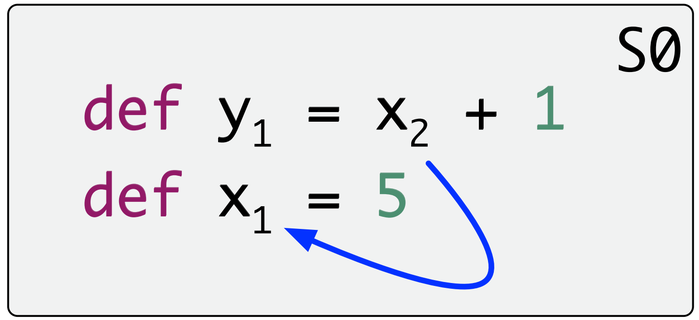
Scopes, References, Declarations, Parents, Imports

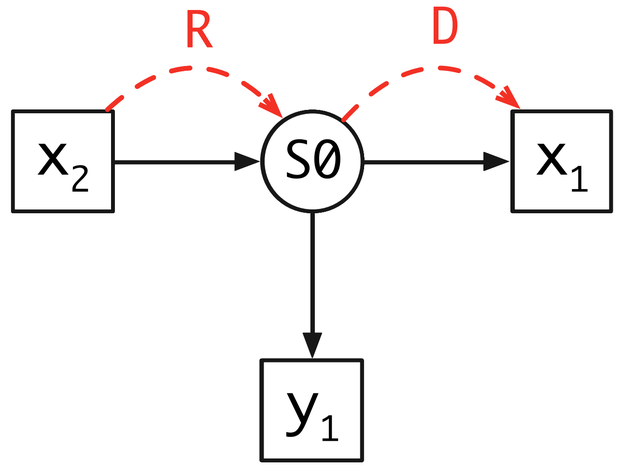


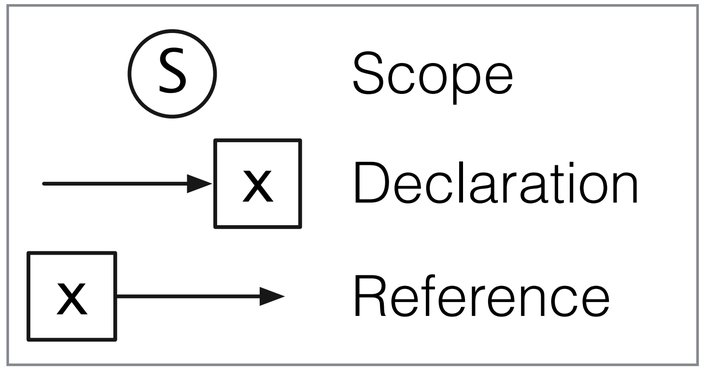
Path in scope graph connects reference to declaration

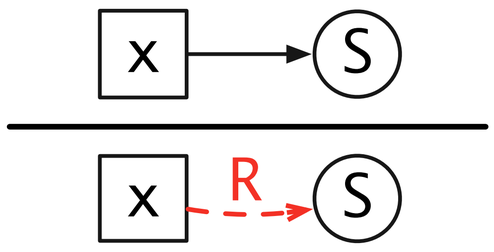
Neron, Tolmach, Visser, Wachsmuth
A Theory of Name Resolution
ESOP 2015

Simple Scopes

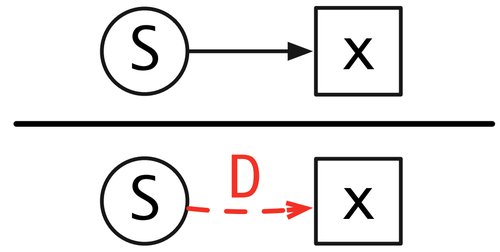






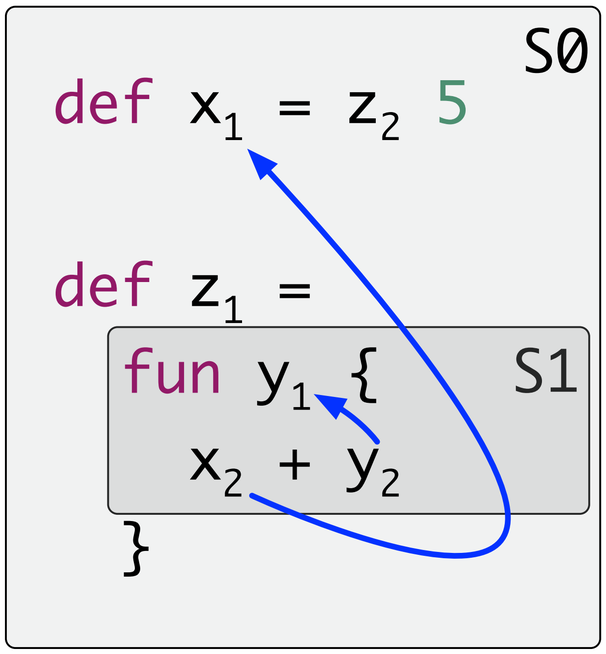


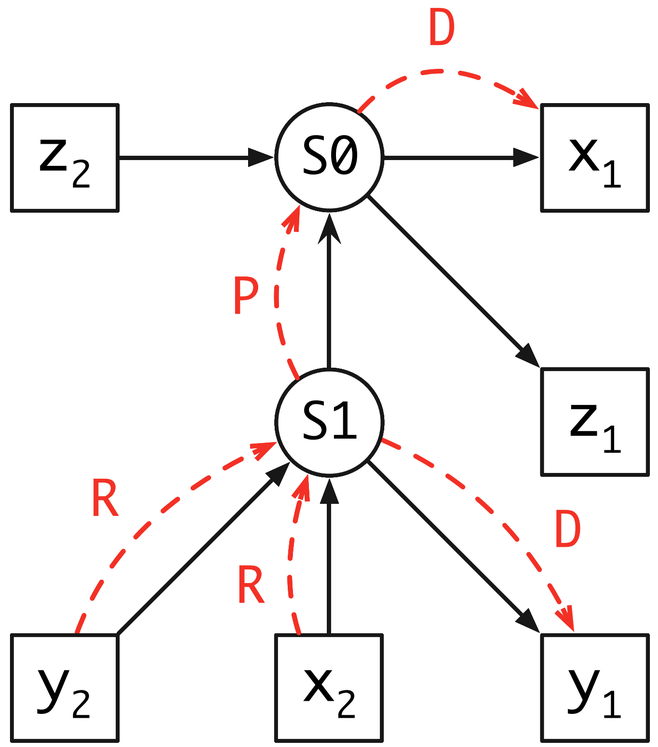
Reference Step

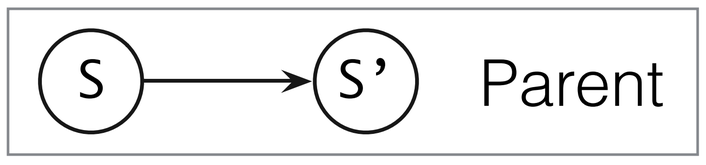


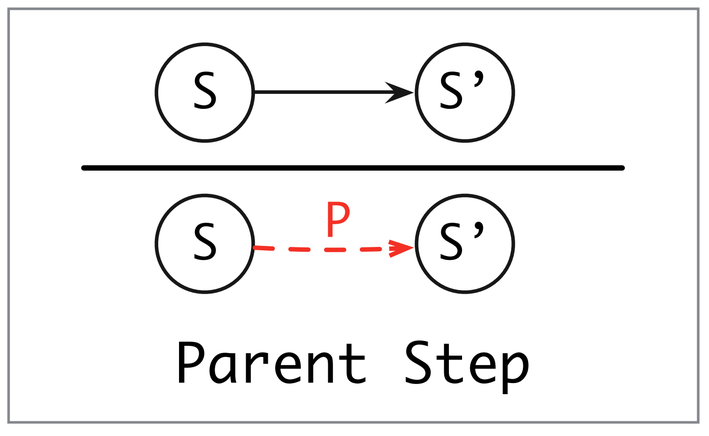
Declaration Step

Lexical Scoping

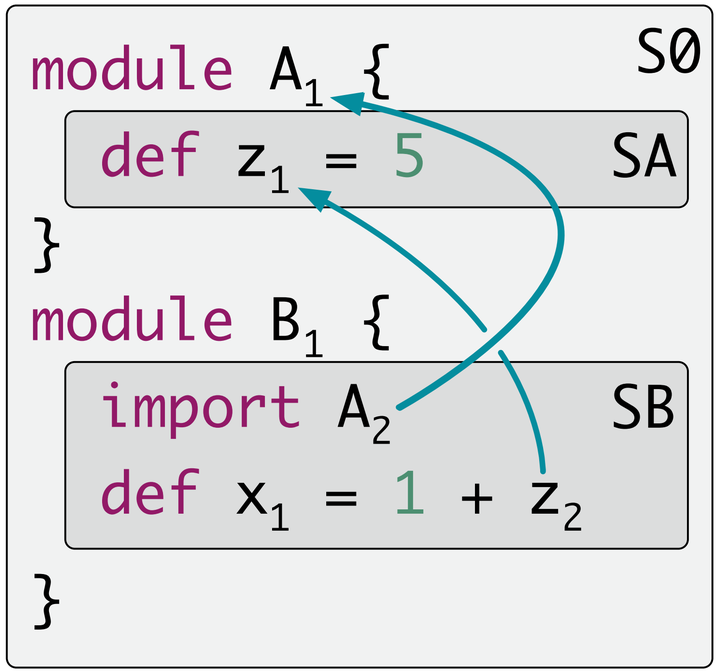


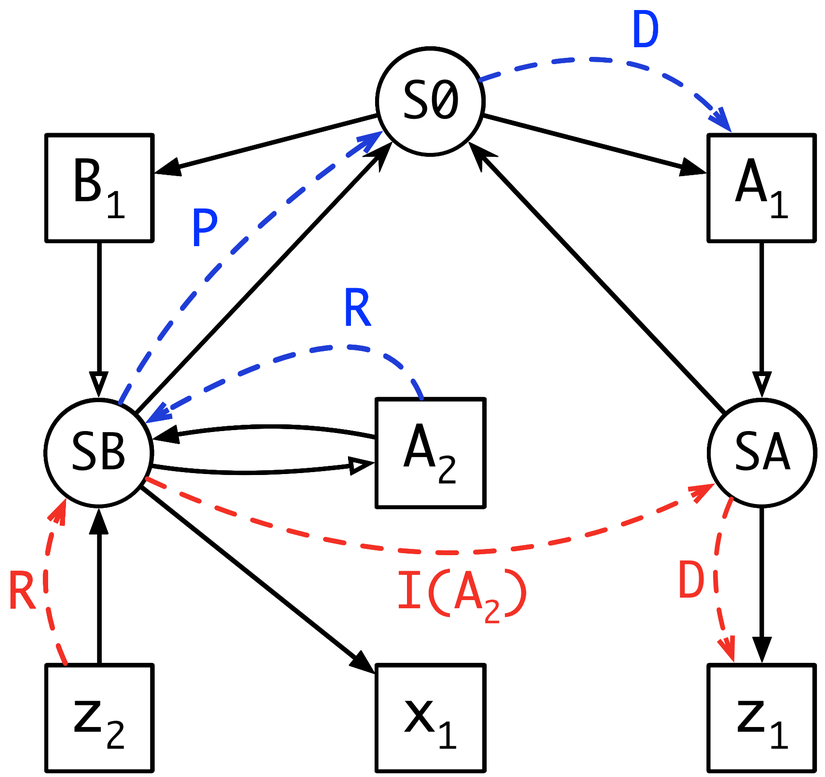


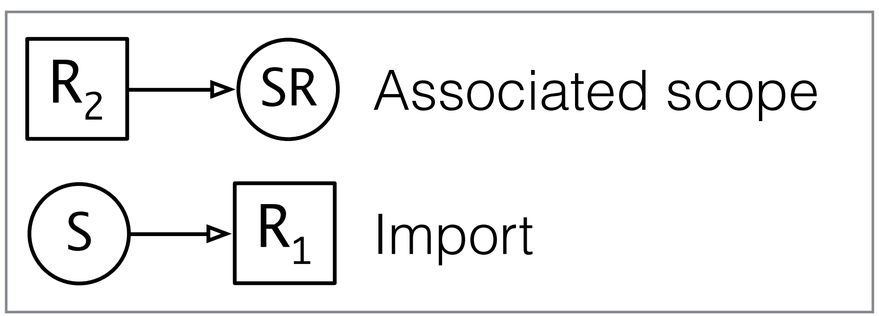


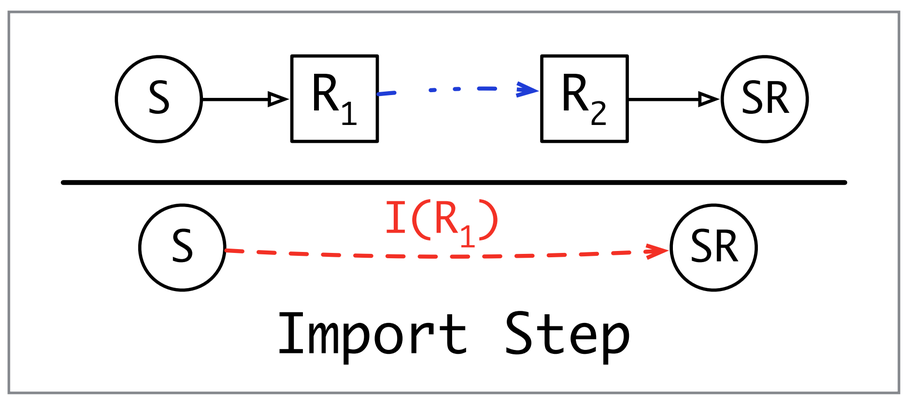


Imports


S0
module A1 {
def z1 = 5
SA
}
module B1 {
import A2
def x1 = 1 + z2
SB
}

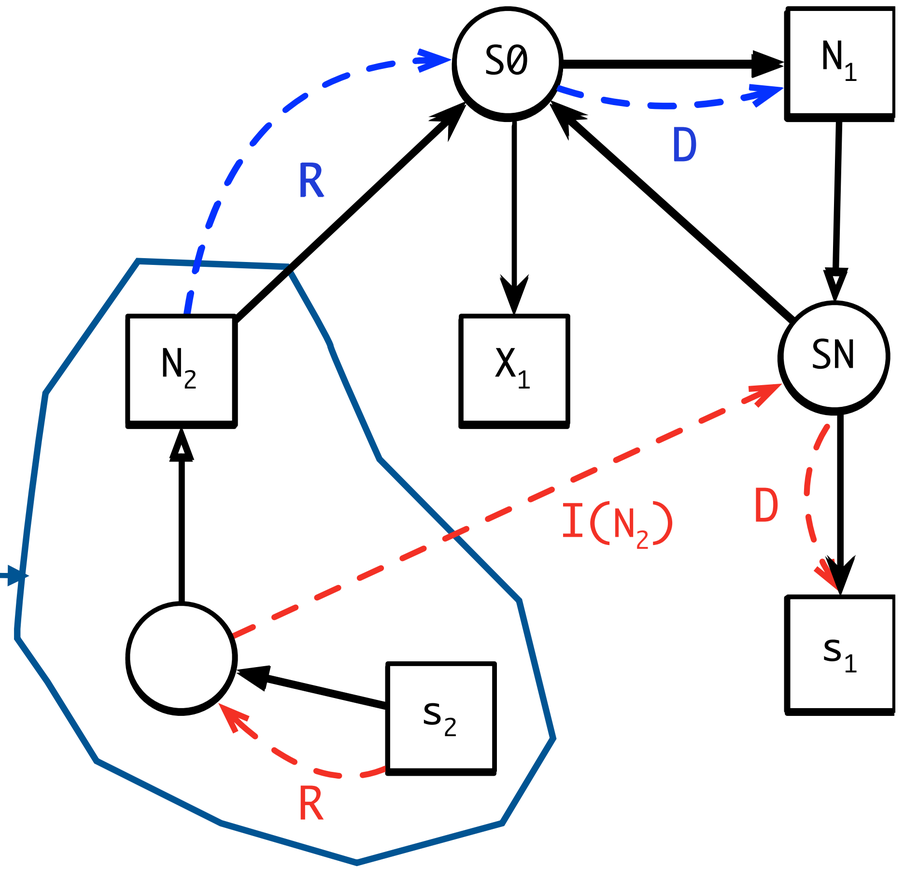

S0
B1
A1
P
D
R
SB
A2
SA
R
I(A2)
D
z2
x1
z1


R2
SR
Associated scope
S
R1
Import


S
R1
R2
SR
I(R1)
S
SR
Import Step

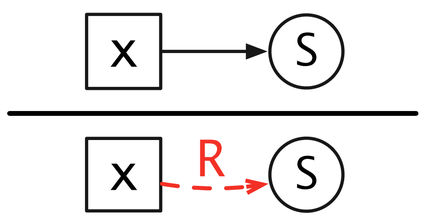
Qualified Names

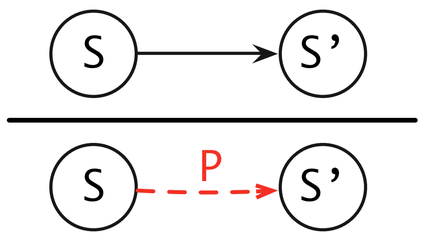
```
module N1 {                                  S0
     def s1 = 5
}

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

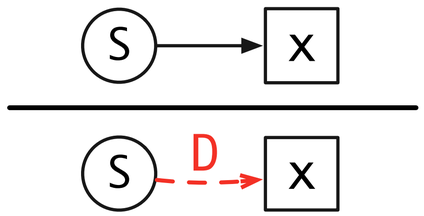


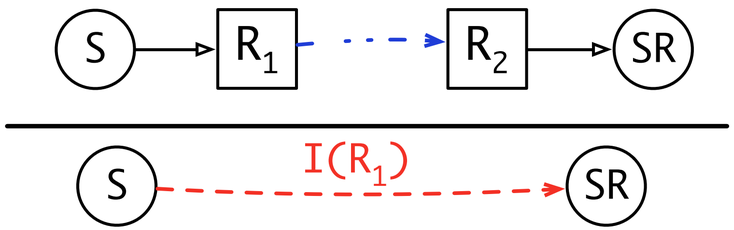
A Calculus for Name Resolution

Reachability of declarations from
references through scope graph edges



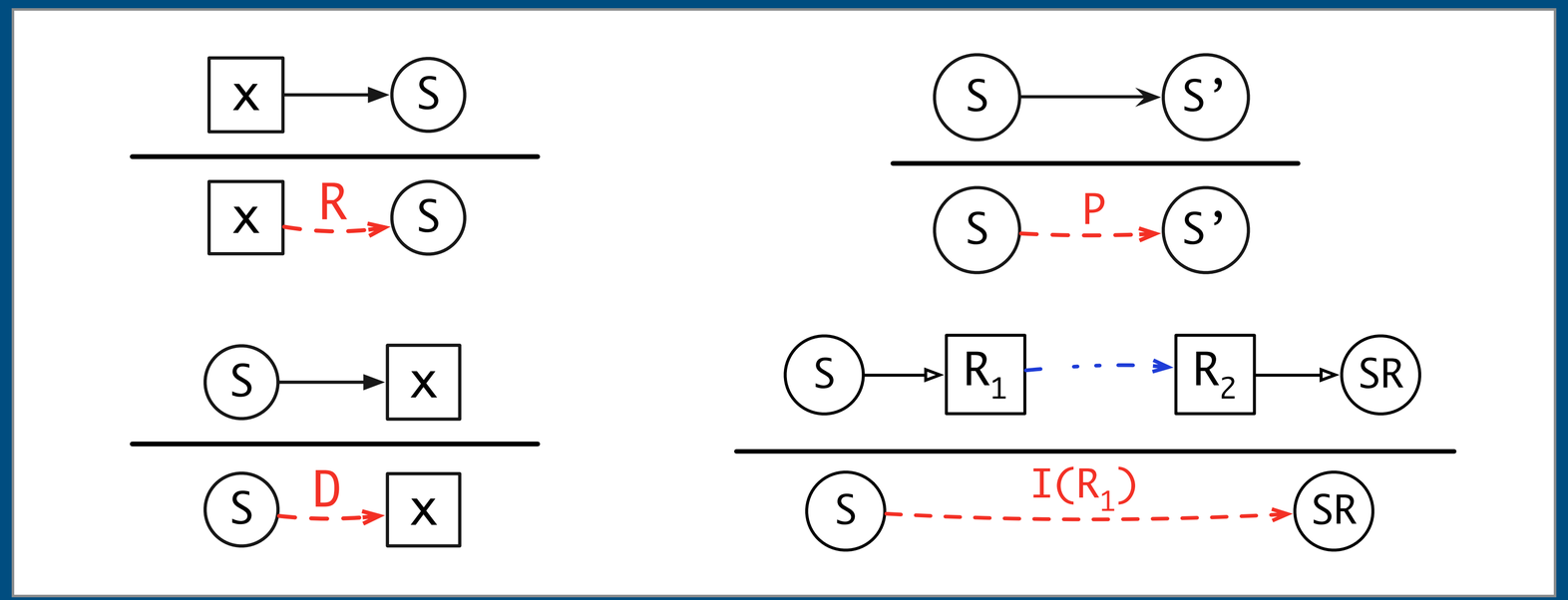






How about ambiguities?
References with multiple paths

A Calculus for Name Resolution



Reachability

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

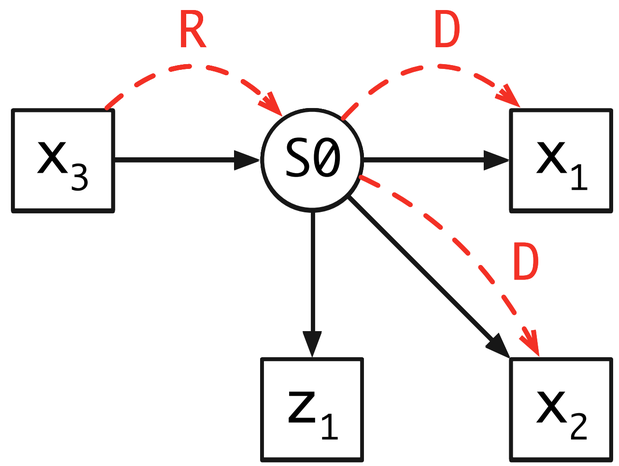
$$\frac{}{D < P.p} \qquad \frac{}{I(\_).p' < P.p}$$

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

Visibility

Ambiguous Resolutions

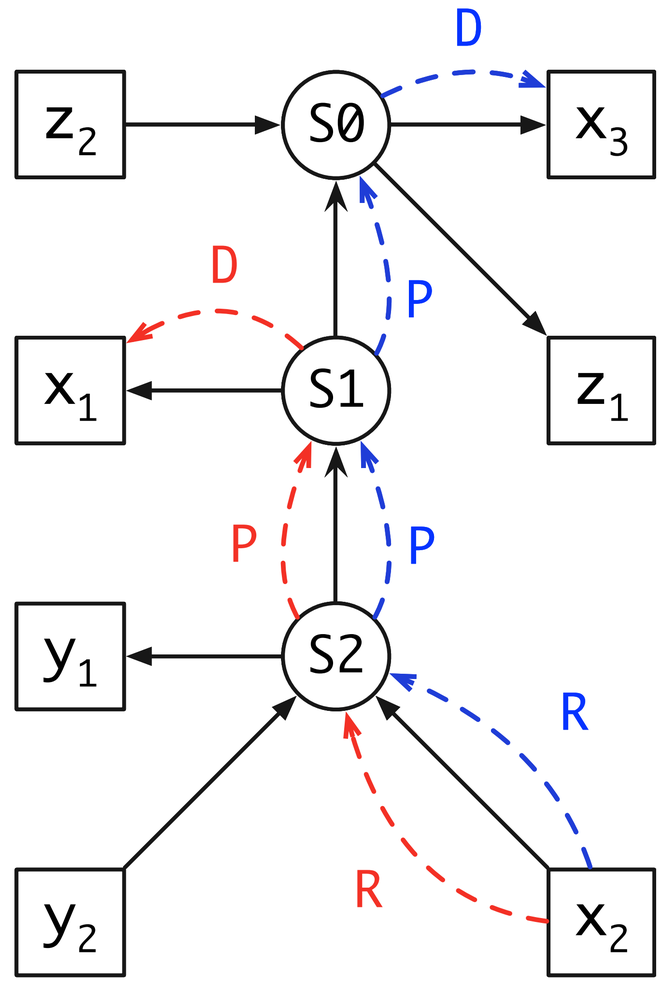
```
S0
def x1 = 5

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
    }
  }
```



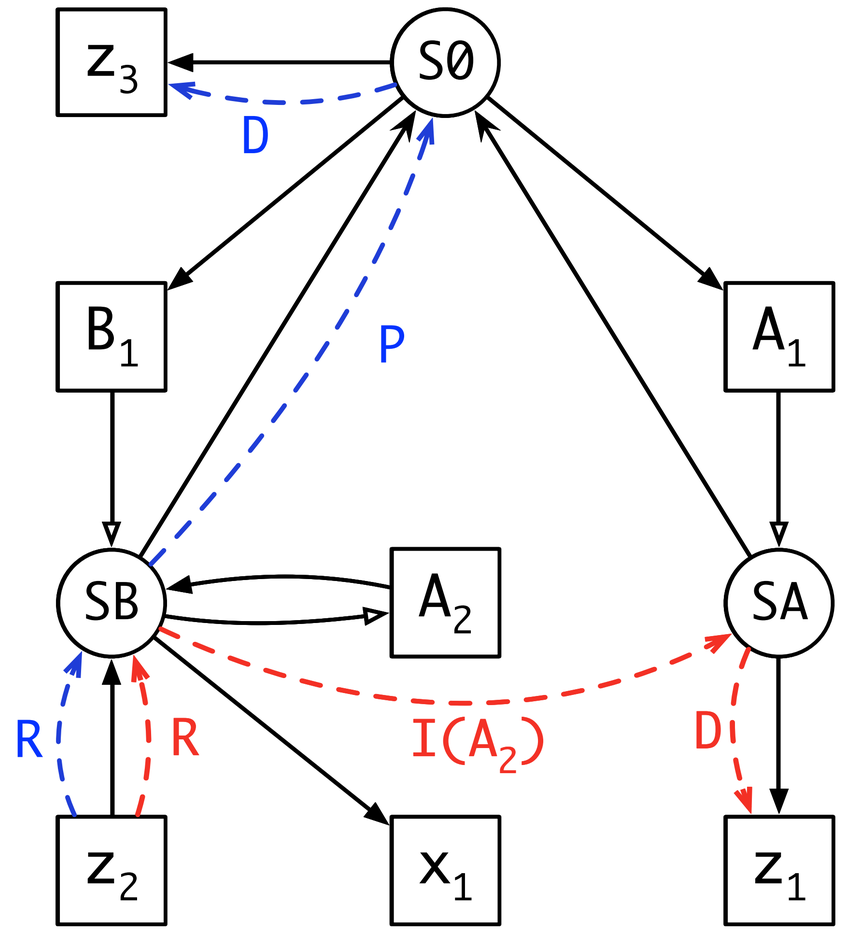
$$\frac{}{D < P.p}$$

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

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

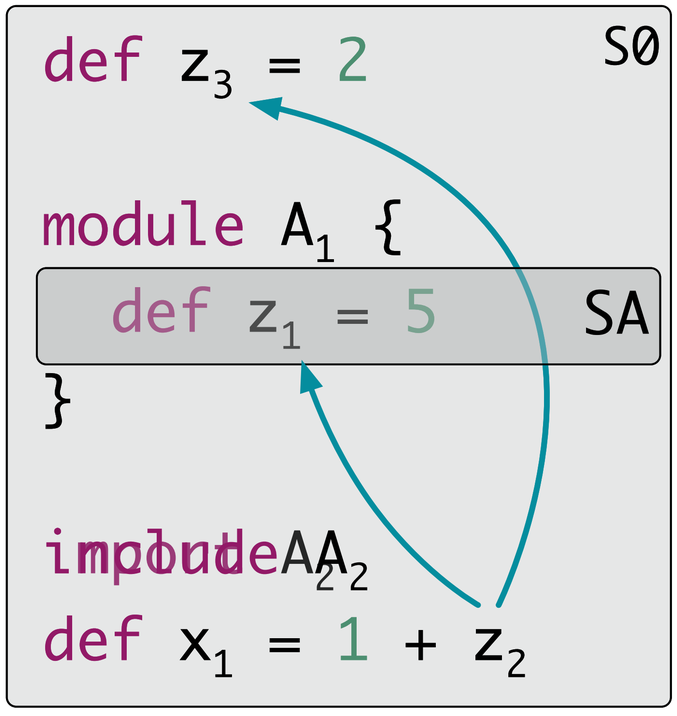
Imports shadow Parents

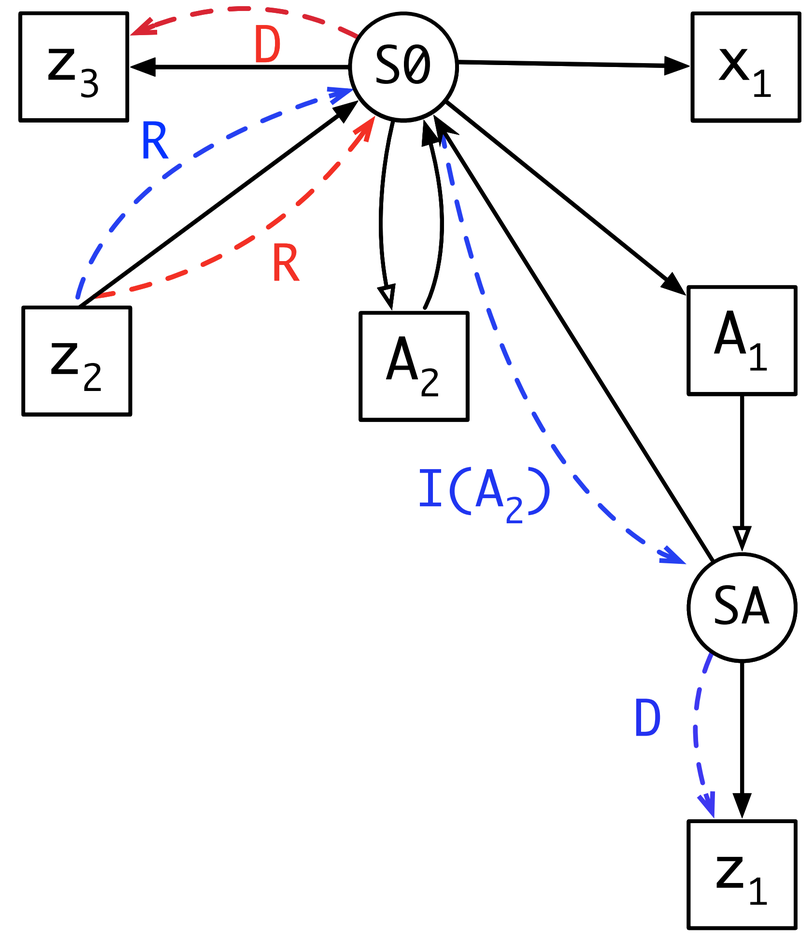
```
def z3 = 2                                  S0

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

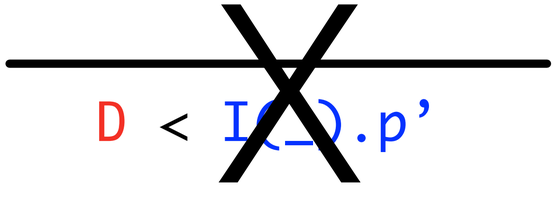


$$\frac{}{\texttt{I(\_).p'} < \texttt{P.p}} \Rightarrow \texttt{R.I(A2).D} < \texttt{R.P.D}$$

Imports vs. Includes

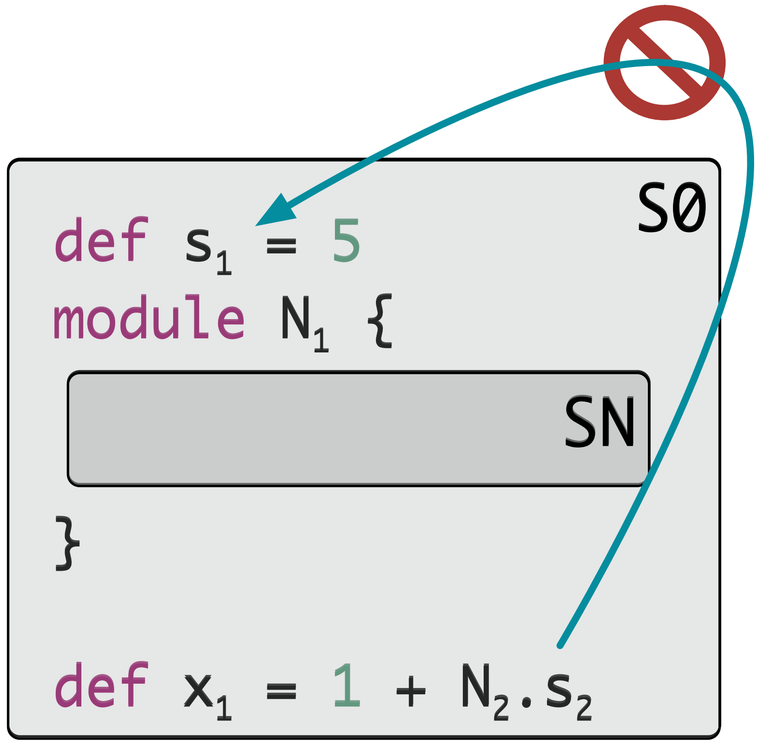


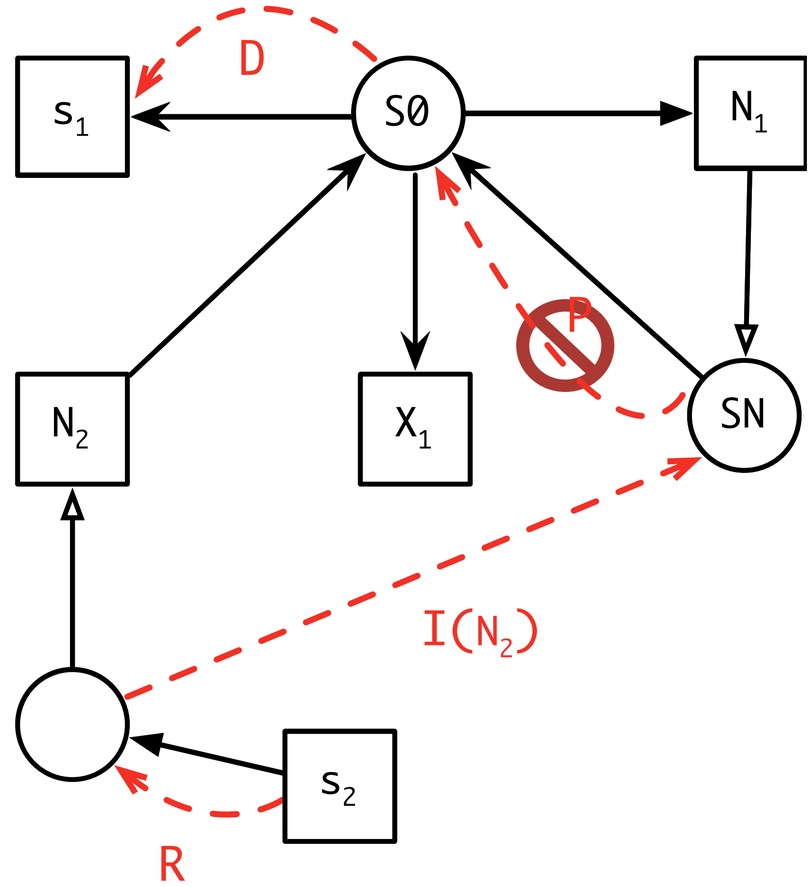




$\Rightarrow$ $R.D < R.I(A_2).D$

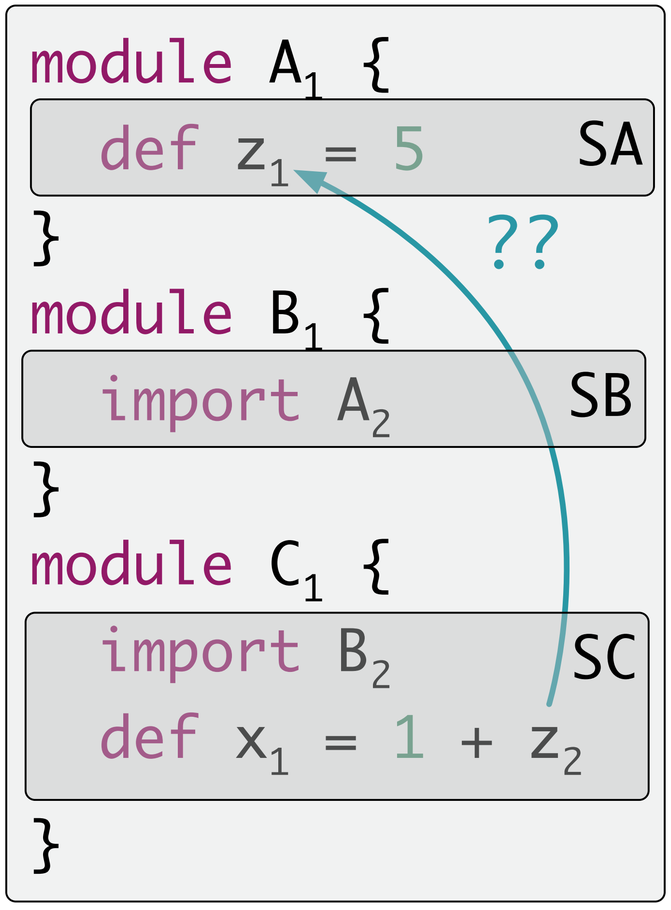
Import Parents

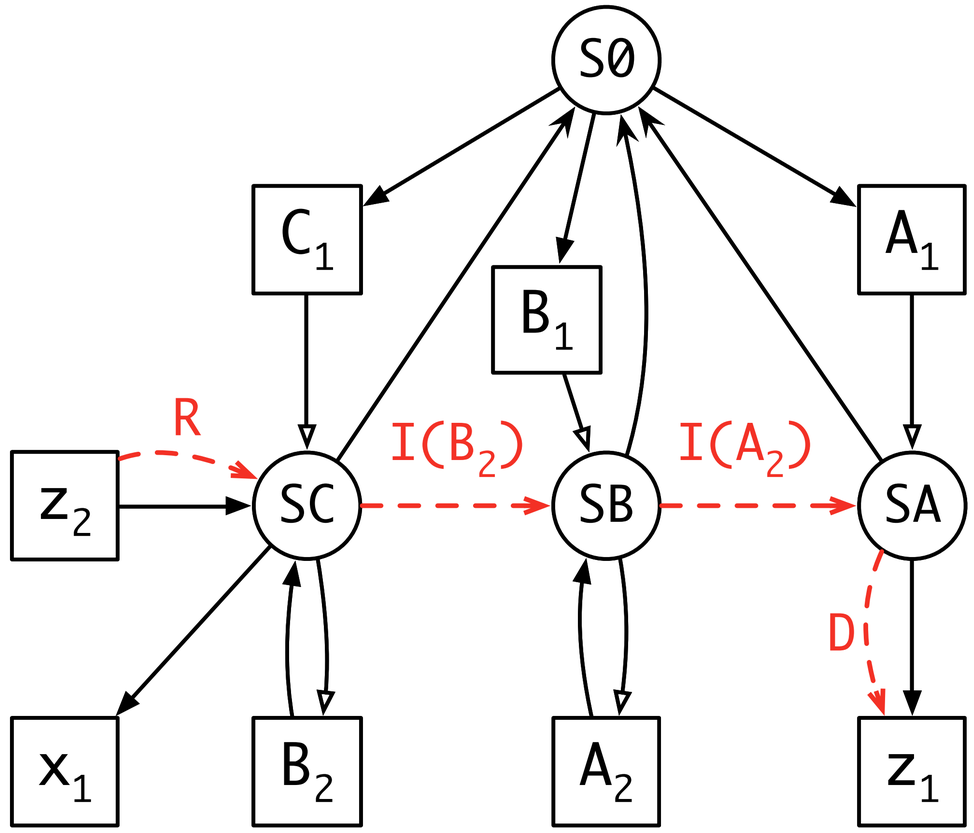




Well formed path: R.P*.I(_)*.D

Transitive vs. Non-Transitive

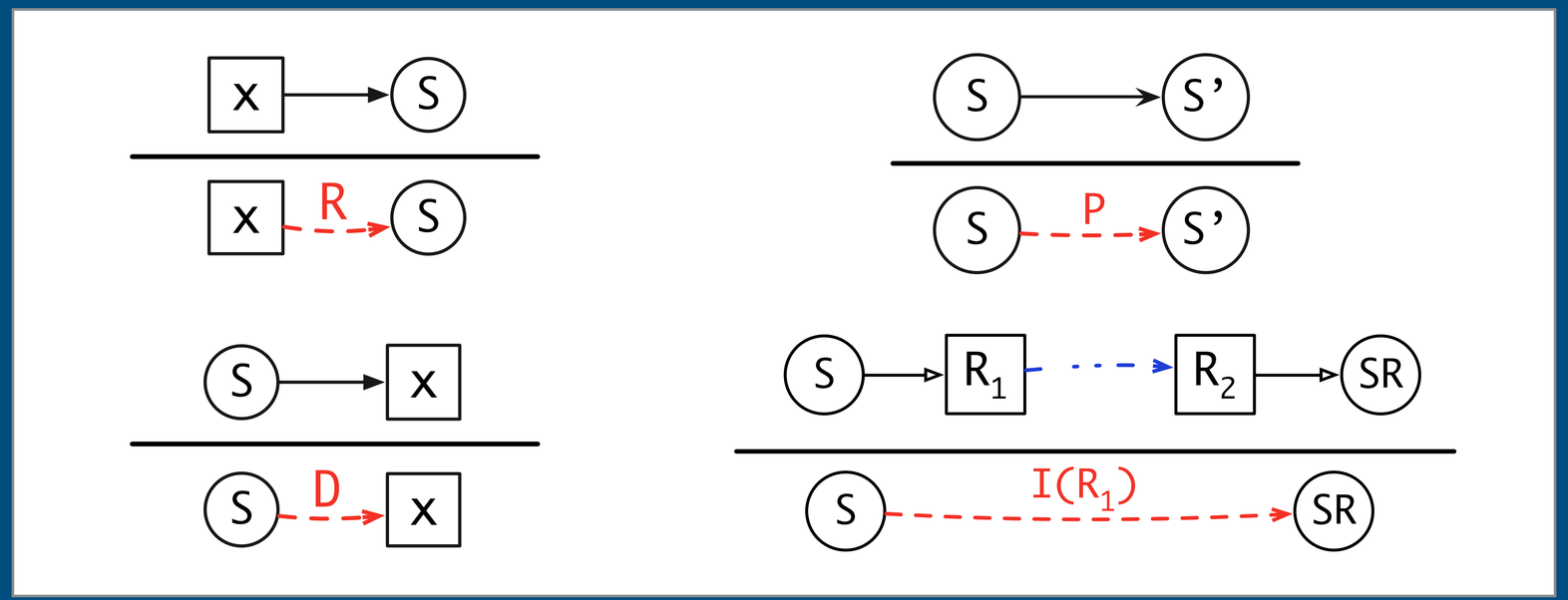




With transitive imports, a well formed path is `R.P*.I(_)*.D`

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

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**

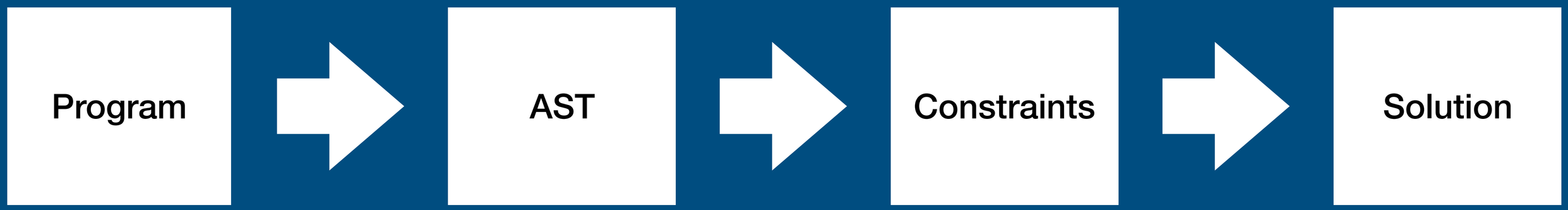
Declarative Rules

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

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Architecture



Parse

Generate Constraints

Resolve Constraints

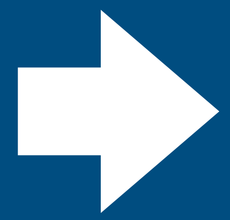
Language Specific

Language Independent

Scope Graph Constraints

```
new s              // new scope

s1 -L-> s2         // labeled edge from scope s1 to scope s2

N{x} <- s          // x is a declaration in scope s for namespace N

N{x} -> s          // x is a reference in scope s for namespace N

N{x} |-> d         // x resolves to declaration d

[[ e ^ (s) ]]      // constraints for expression e in scope s
```

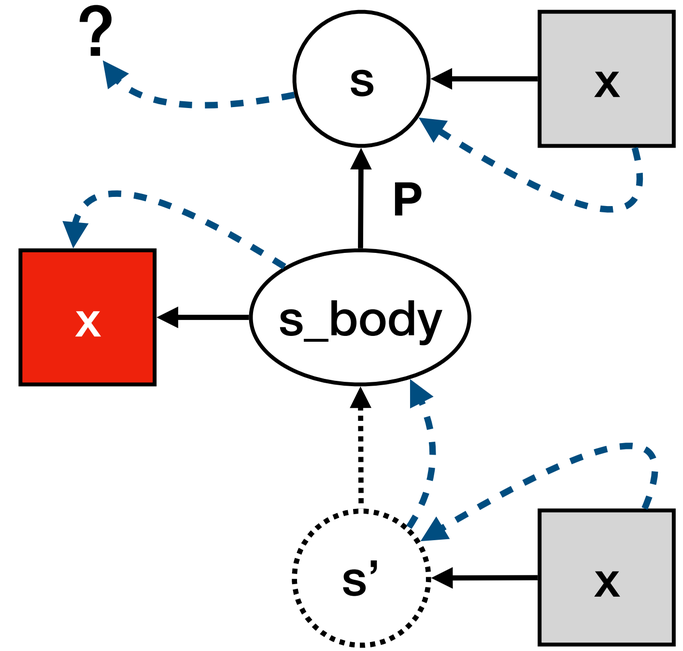
```
let
  var x : int := x + 1
 in
   x + 1
end
```



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

```
[[ Let([VarDec(x, t, e)], [e_body]) ^ (s) ]] :=
  new s_body,                  // new scope
  s_body -P-> s,               // parent edge to enclosing scope
  Var{x} <- s_body,            // x is a declaration in s_body
  [[ e ^ (s) ]],               // init expression
  [[ e_body ^ (s_body) ]].     // body expression
```

```
[[ Var(x) ^ (s') ]] :=
  Var{x} -> s', // x is a reference in s'
  Var{x} |-> d, // check that x resolves to a declaration
```



How about types?

Type Constraints

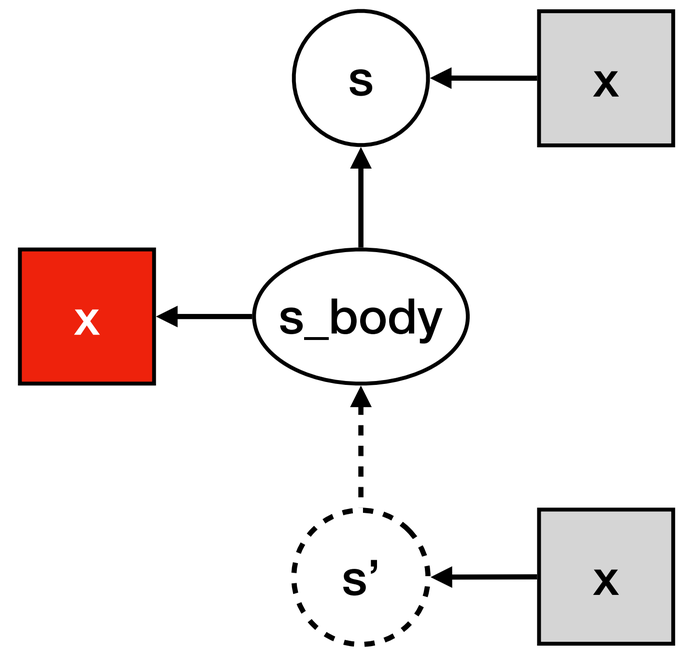
```
d : ty                  // declaration has type

t1 == ty2               // type equality

ty1 <! ty2              // declare sub-type

ty1 <? ty2              // query sub-type

. . .                   // extensions

[[ e ^ (s) : ty ]]      // type of expression in scope
```

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

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

```
[[ Let([VarDec(x, t, e)], [e_body]) ^ (s) : ty' ]] :=
   new s_body,                  // new scope
   s_body -P-> s,               // parent edge to enclosing scope
   Var{x} <- s_body,            // x is a declaration in s_body
   Var{x} : ty,                 // associate type
   [[ t ^ (s) : ty ]],          // type of type
   [[ e ^ (s) : ty ]],          // type of expression
   [[ e_body ^ (s_body) : ty' ]]. // constraints for body
```

```
[[ Var(x) ^ (s') : ty ]] :=
   Var{x} -> s', // x is a reference in s'
   Var{x} |-> d, // check that x resolves to a declaration
   d : ty.       // type of declaration is type of reference
```



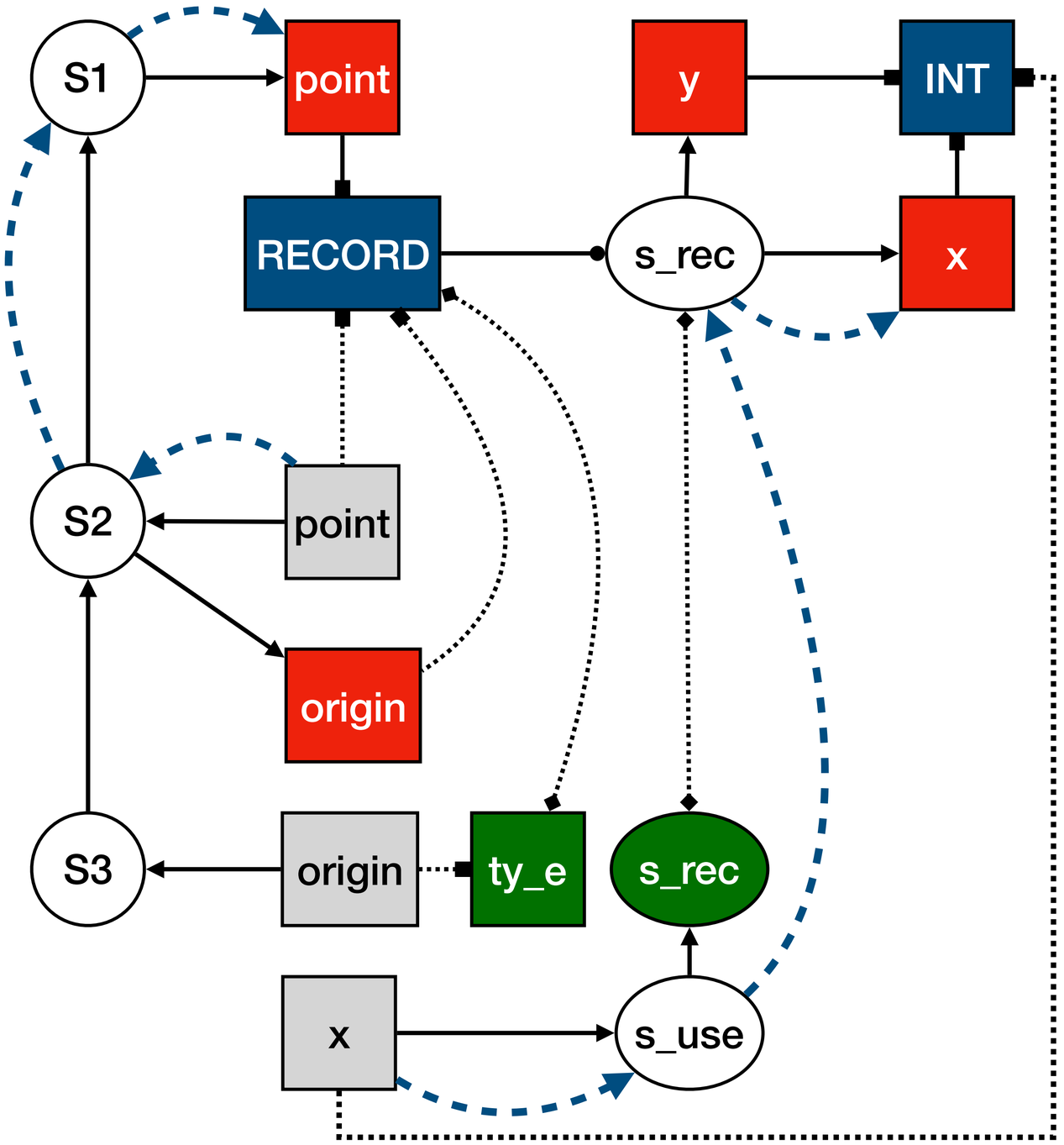
```
let
  type point = {x : int, y : int}
  var origin : point := …
 in origin.x
end
```

```
[[ RecordTy(fields) ^ (s) : ty ]] :=
   ty == RECORD(s_rec),
   new s_rec,
   Map2[[ fields ^ (s_rec, s) ]].
```

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

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

Type Dependent Name Resolution



Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- Scope & Type Constraint Rules

A language for talking about name binding

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

NaBL2 in Spoofax Language Workbench

workspace - Java - org.metaborg.lang.tiger/trans/statics/records.nabl2 - Eclipse

Quick Access

records.nabl2

```
20
21 rules // literals
22
23   [[ NilExp()  ^ (s) : NIL() ]] := true.
24
25 rules // record creation
26
27   [[ r@Record(t, inits) ^ (s) : ty ]] :=
28      [[ t ^ (s) : ty ]],
29      ty == RECORD(s_rec) | error $[record type expected],
30      new s_use, s_use -I-> s_rec,
31      D(s_rec)/Field subseteq/name R(s_use)/Field | error $[Field [NAME] not initialized] @r,
32      distinct/name R(s_use)/Field | error $[Duplicate initialization of field [NAME]] @NAMES,
33      Map2[[ inits ^ (s_use, s) ]].
34
35   [[ InitField(x, e) ^ (s_use, s) ]] :=
36      Field{x} -> s_use,
37      Field{x} |-> d,
38      d : ty1,
39      [[ e ^ (s) : ty2 ]],
40      ty2 <? ty1 | error $[type mismatch got [ty2] where [ty1] expected].
41
42 rules // record field access
43
44   [[ FieldVar(e, f) ^ (s) : ty ]] :=
45      [[ e ^ (s) : ty_e ]],
46      ty_e == RECORD(s_rec),
47      new s_use, s_use -I-> s_rec,
48      Field{f} -> s_use,
49      Field{f} |-> d,
50      d : ty.
```

record.tig

```
1 let
2   type point = {x : int, y : int}
3   var origin : point := point { x = 1, y = 2 }
4  in origin.x
5 end
6
```

Writable | Insert

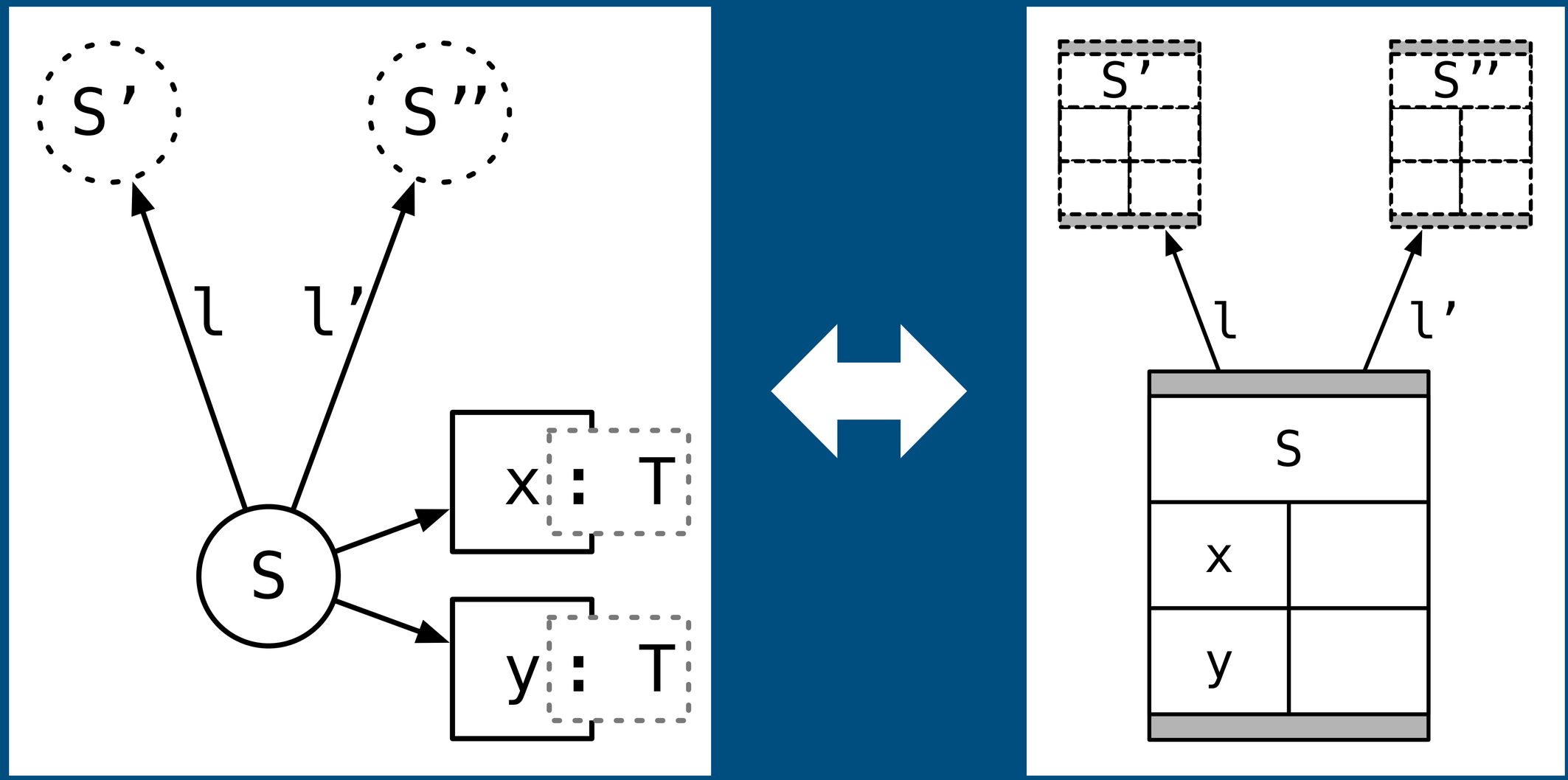
http://spoofax.org

Applications

Languages

- IceDust2 [ECOOP17]
- Education
  - Mini-Java, Tiger, Calc
- Programming languages
  - Pascal, TypeScript, F#
  - (student projects in progress)
- Bootstrapping language workbench
  - NaBL2, …

Scopes Describe Frames [ECOOP16]



A Uniform Model for Memory Layout
in Dynamic Semantics

Scope Graphs for Name Binding: Status

Theory

- Resolution calculus
- Name binding and type constraints
- Resolution algorithm sound wrt calculus
- Mapping to run-time memory layout

Declarative specification

- NaBL2: generation of name and type constraints

Tooling

- Solver (second version)
- Integrated in Spoofax Language Workbench
  - editors with name and type checking
  - navigation

Scope Graphs for Name Binding: Limitations

A domain-specific (= restricted) model

- Cannot describe all name resolution algorithms implemented in Turing complete languages

Normative model

- ‘this is name binding’

Claim/hypothesis

- Describes all sane models of name binding

Scope Graphs for Name Binding: Future Work

Theory

- Scopes = structural types?
  - operations for scope / type comparison
- Generics
  - DOT-style?
- Type soundness of interpreters — automatically

Tooling

- Tune name binding language (notation)
- Incremental analysis (in progress)
- Code completion
- Refactoring (renaming)

Scope Graphs for Name Binding: The Future

A common (cross-language) understanding of name binding?

A foundation for formalization and implementation of programming languages?