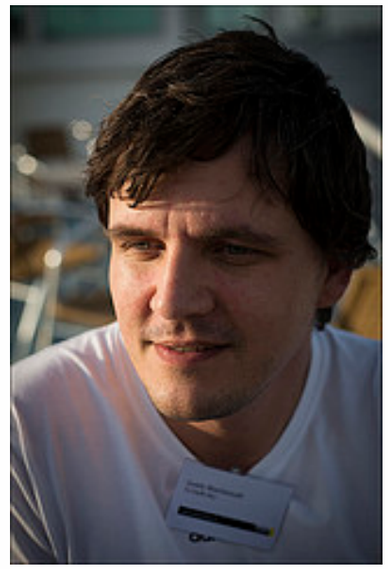
A Theory of Name Resolution

Pierre Neron[1]

Andrew Tolmach[2]



Eelco Visser[1]

Guido Wachsmuth[1]

1 TU Delft Delft University of Technology

2 Portland State UNIVERSITY

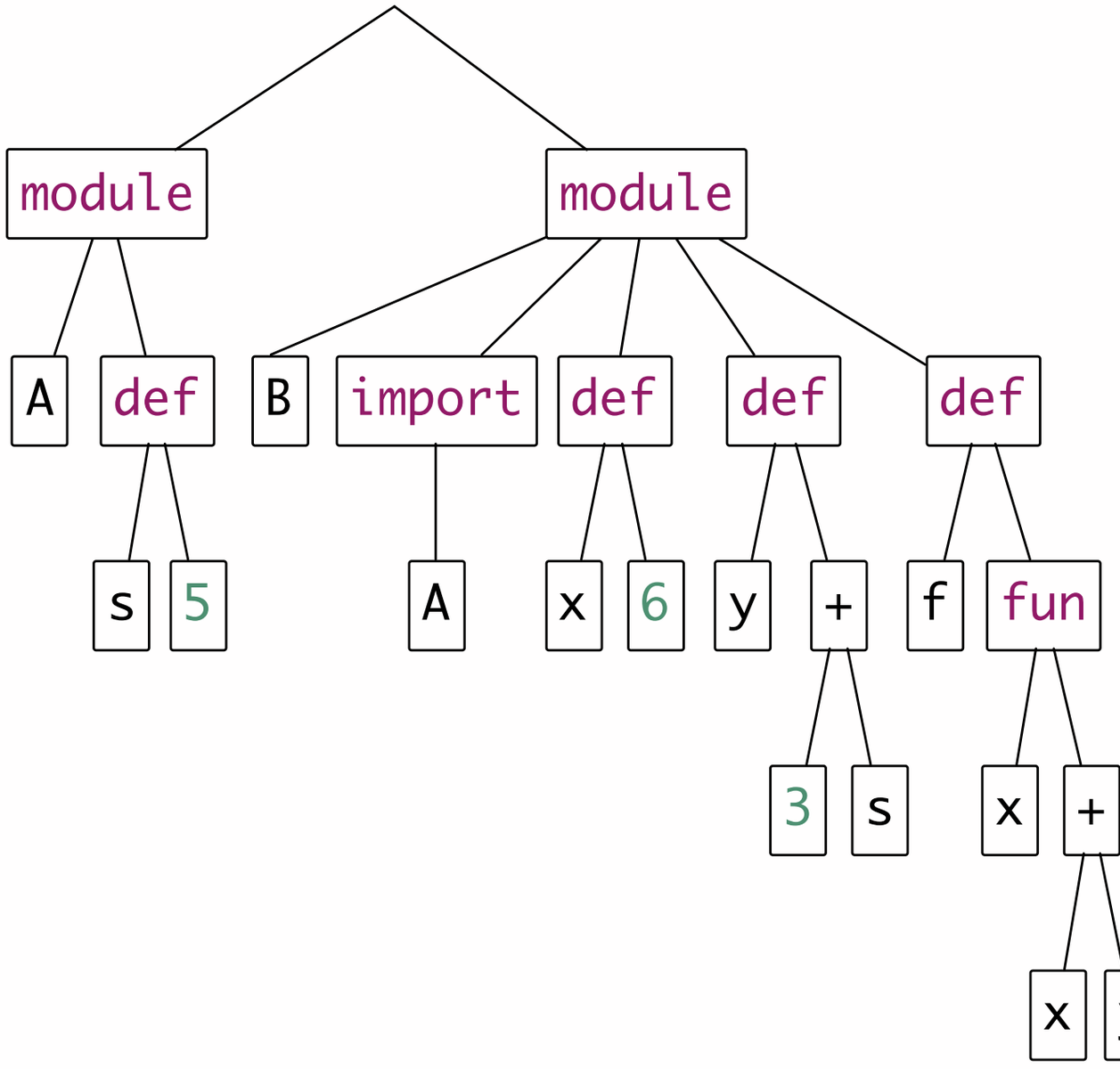
[ESOP15]

Static Name Resolution

```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```

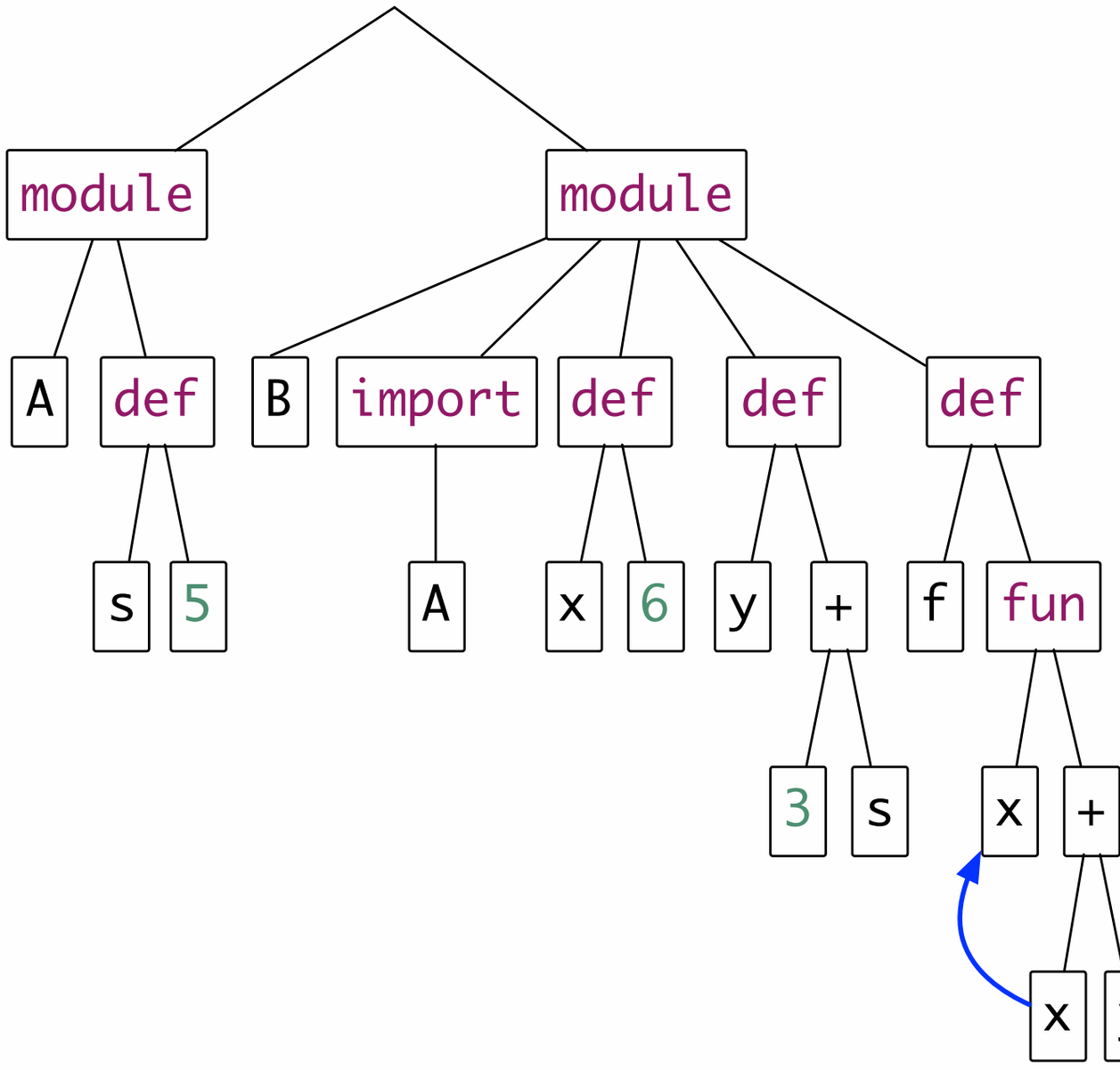
Static Name Resolution

```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```



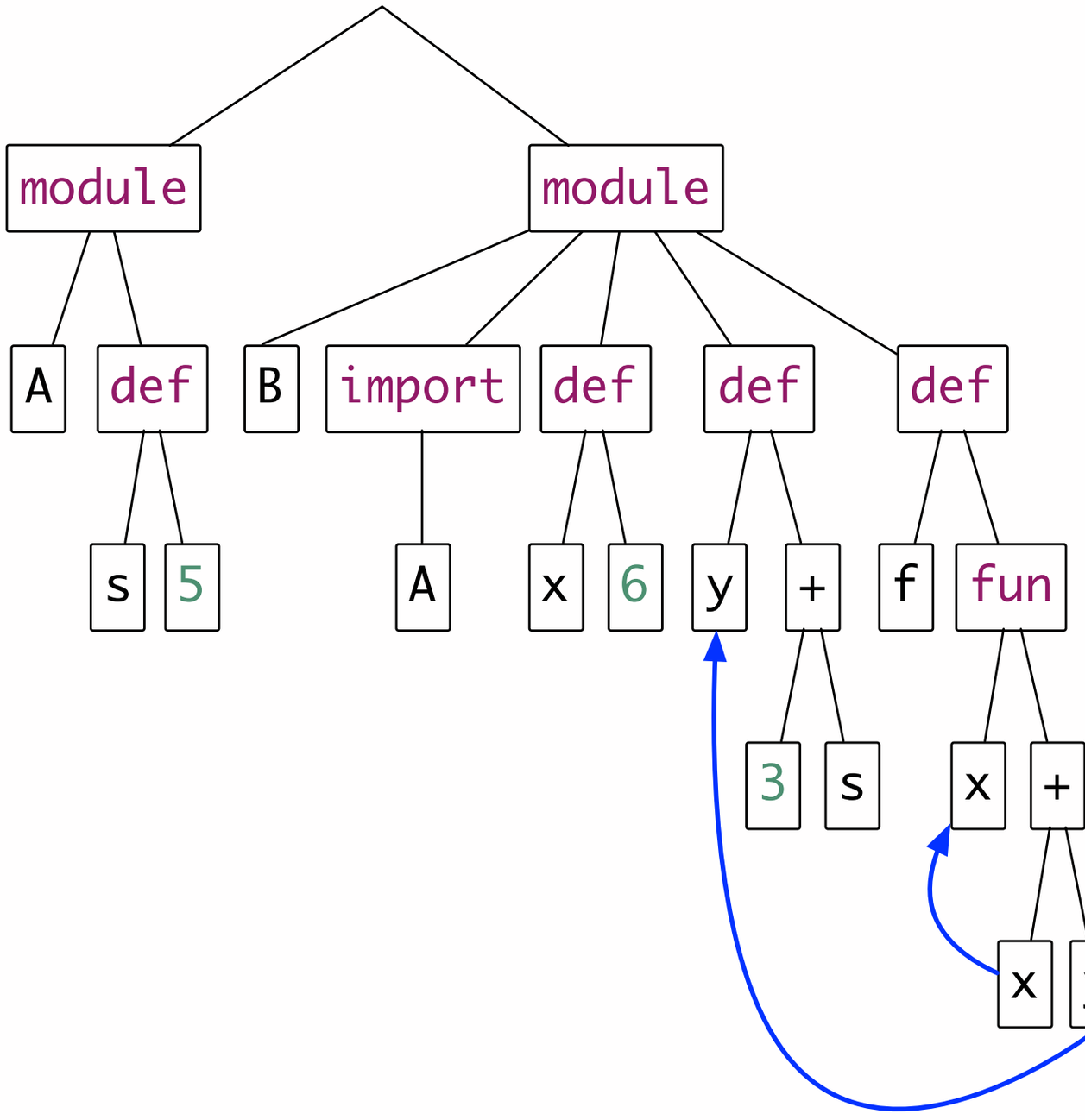
Static Name Resolution

```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```



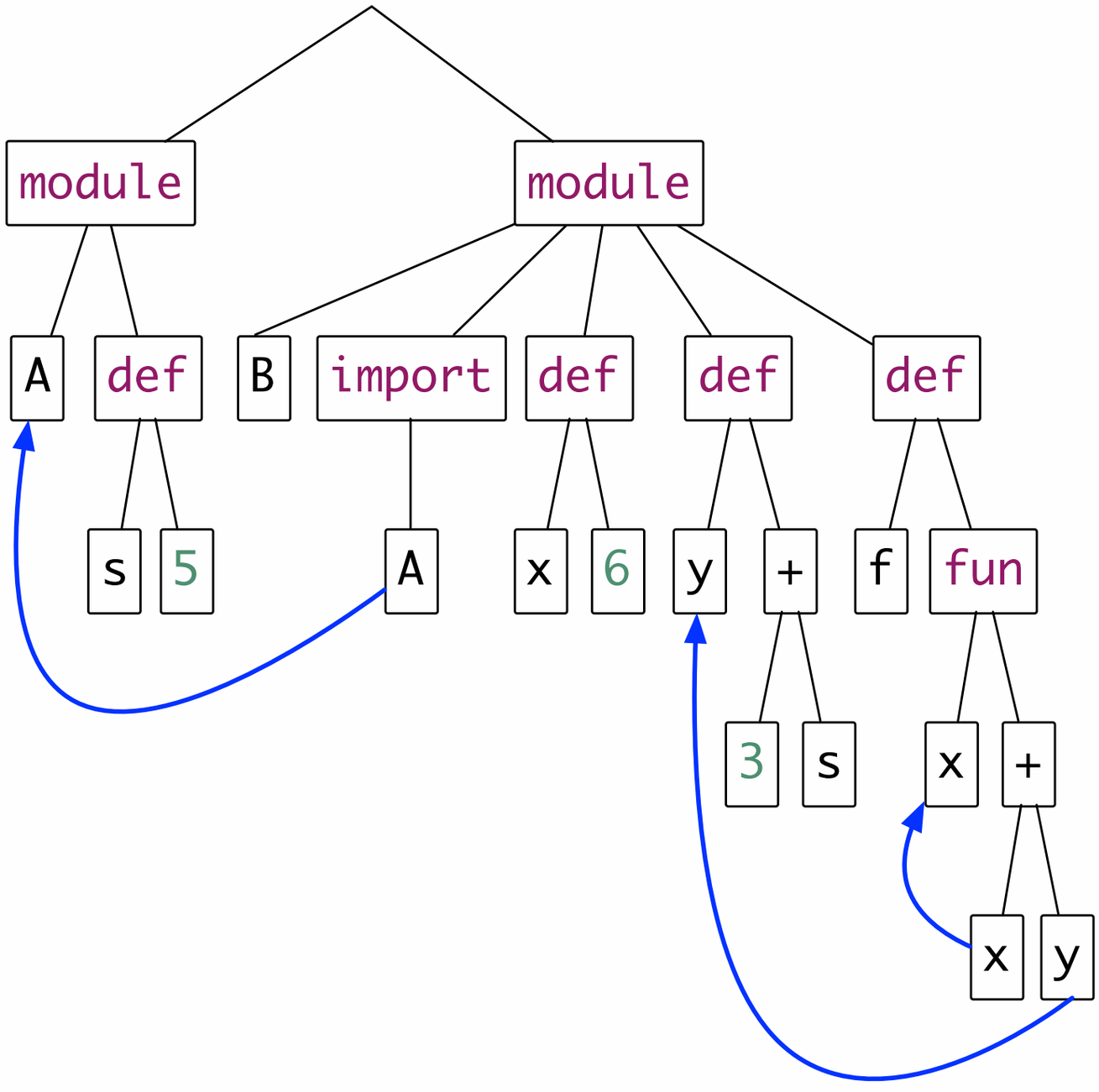
Static Name Resolution

```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```



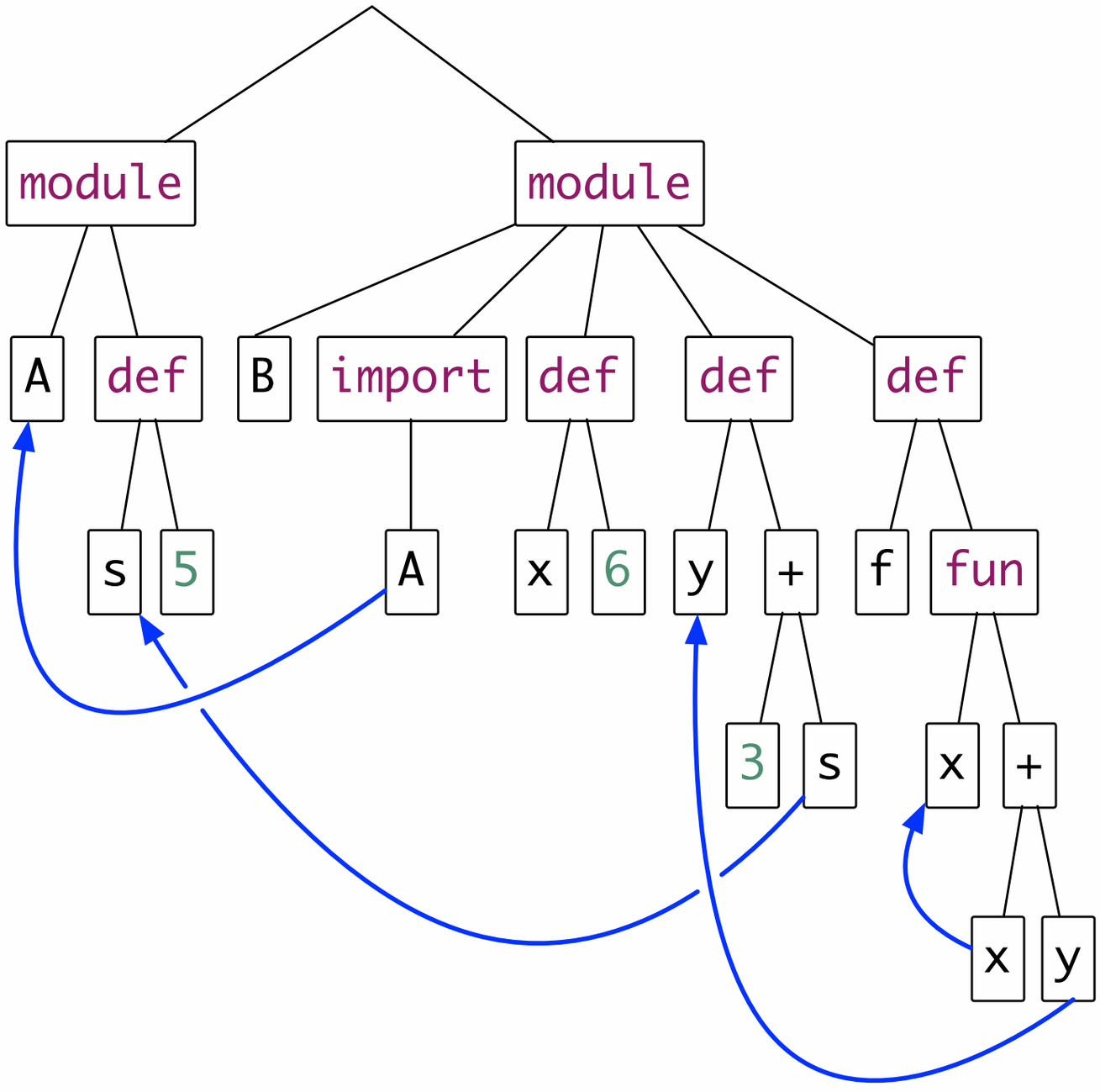
Static Name Resolution

```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```



Static Name Resolution

```
module A {
    def s = 5
}

module B {
    import A
    def x = 6
    def y = 3 + s
    def f =
      fun x { x + y }
}
```



Use Scope Graphs to Describe Name Resolution

Use Scope Graphs to Describe Name Resolution ??

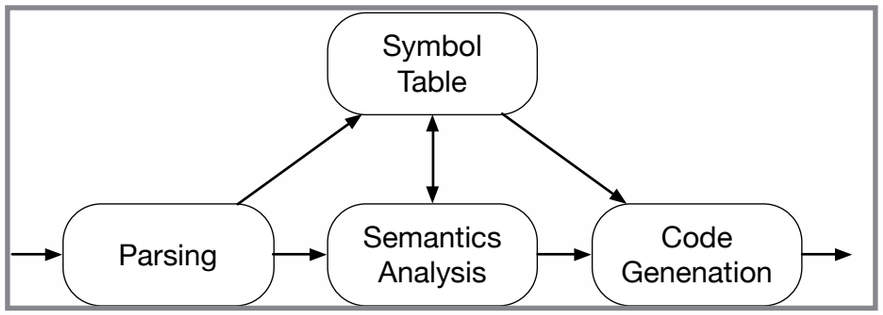
Name Resolution is Pervasive

Name Resolution is Pervasive

Appears in many different artifacts…

Name Resolution is Pervasive

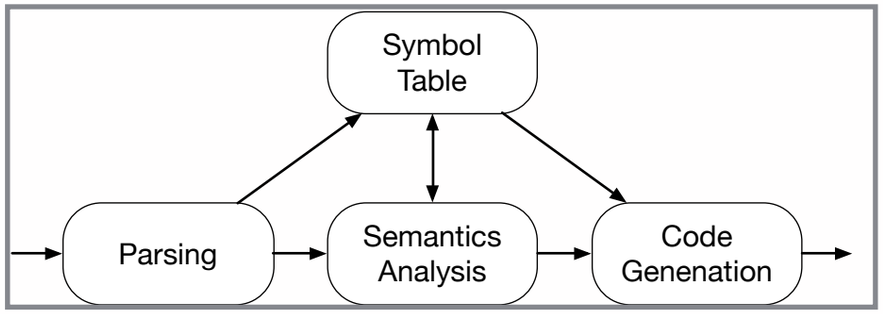
Appears in many different artifacts…



Compiler

Name Resolution is Pervasive

Appears in many different artifacts…



Compiler

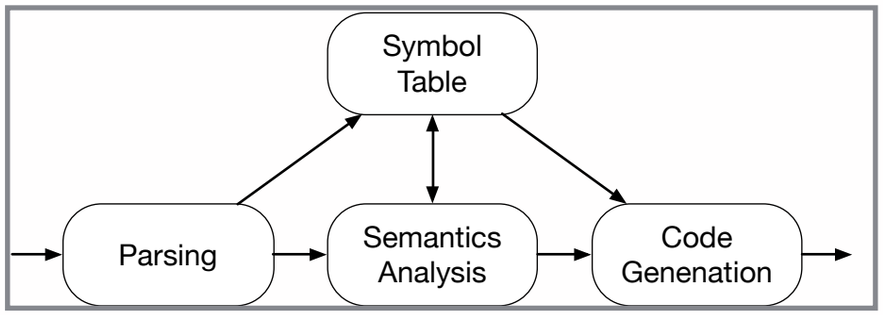
$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics

Name Resolution is Pervasive

Appears in many different artifacts…

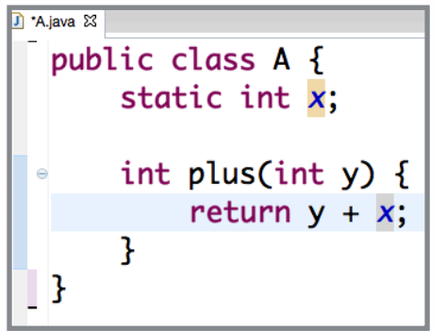


Compiler

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

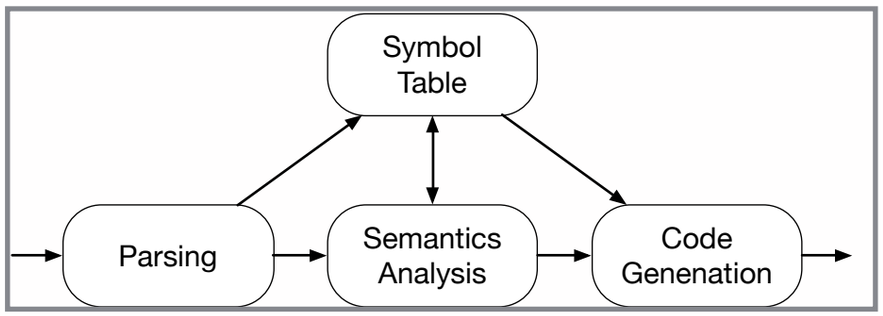
Semantics



IDE

Name Resolution is Pervasive

Appears in many different artifacts…

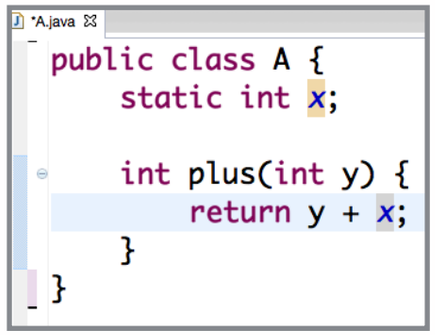


Compiler

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics

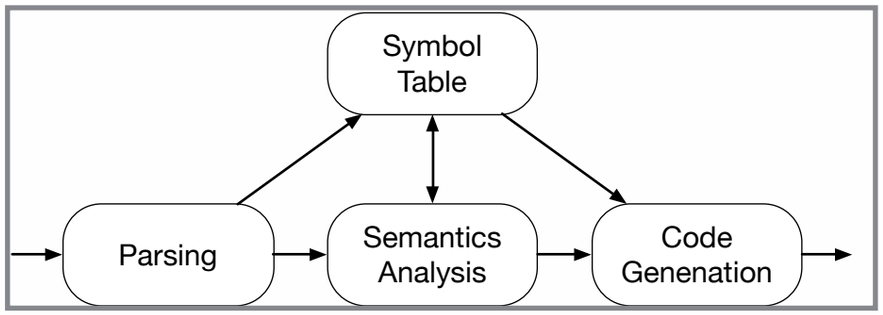


IDE

… with rules encoded in many different ad-hoc ways

Name Resolution is Pervasive

Appears in many different artifacts…

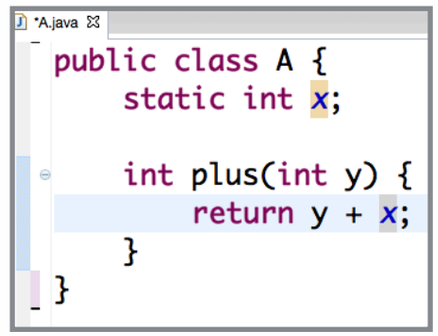


Compiler

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

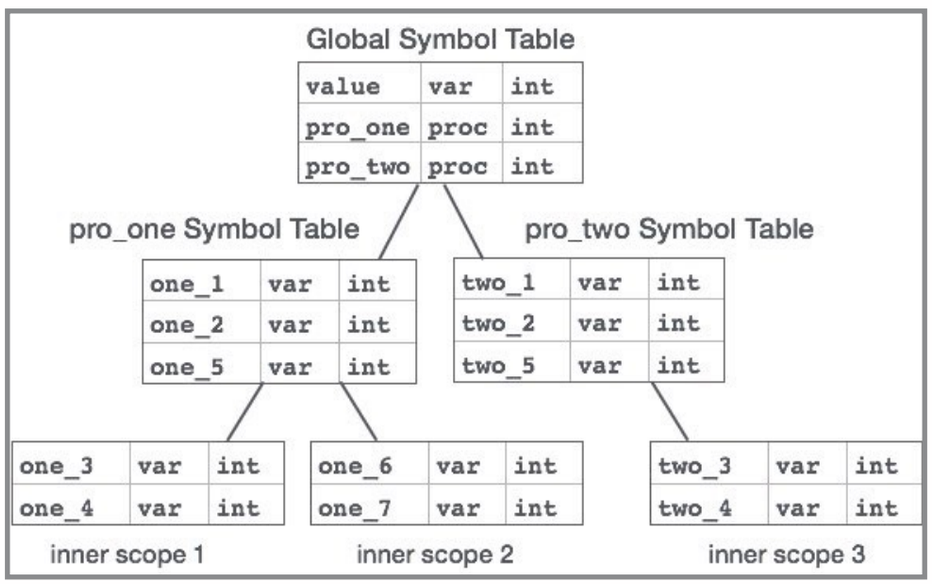
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics



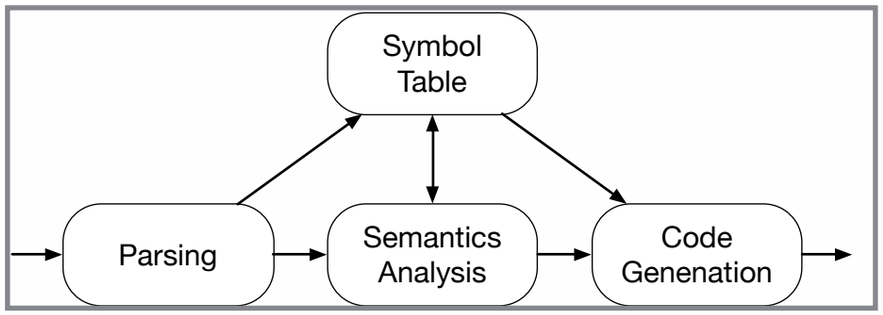
IDE

… with rules encoded in many different ad-hoc ways



Name Resolution is Pervasive

Appears in many different artifacts…

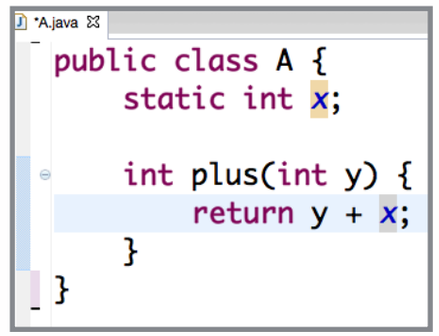


Compiler

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

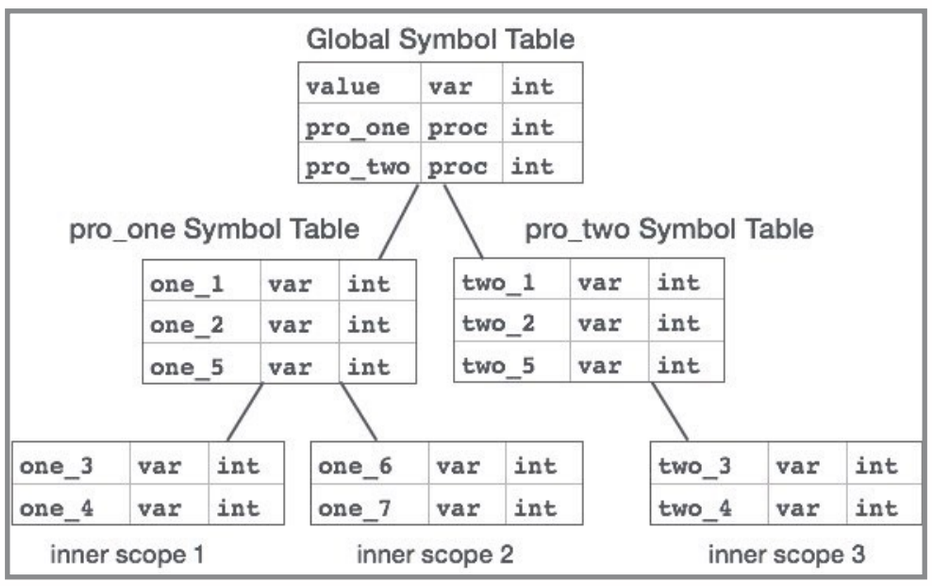
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics



IDE

… with rules encoded in many different ad-hoc ways

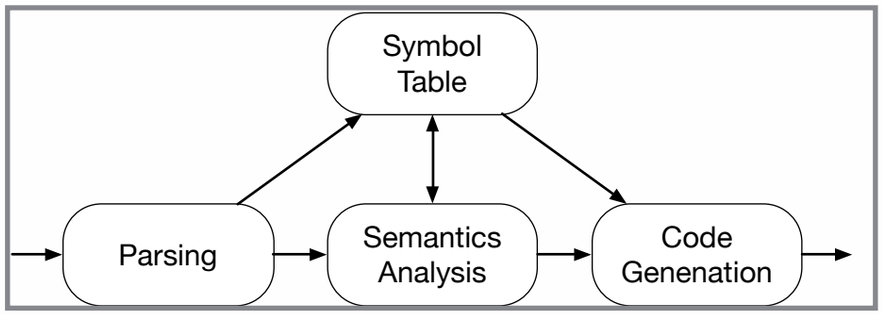


`x:int, Γ`

`[3/x].σ`

Name Resolution is Pervasive

Appears in many different artifacts…

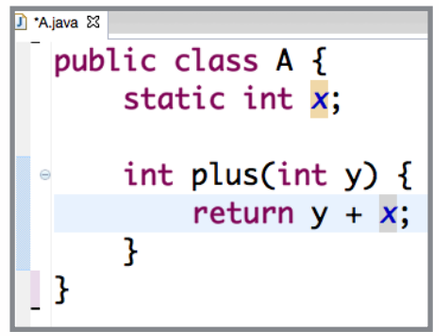


Compiler

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

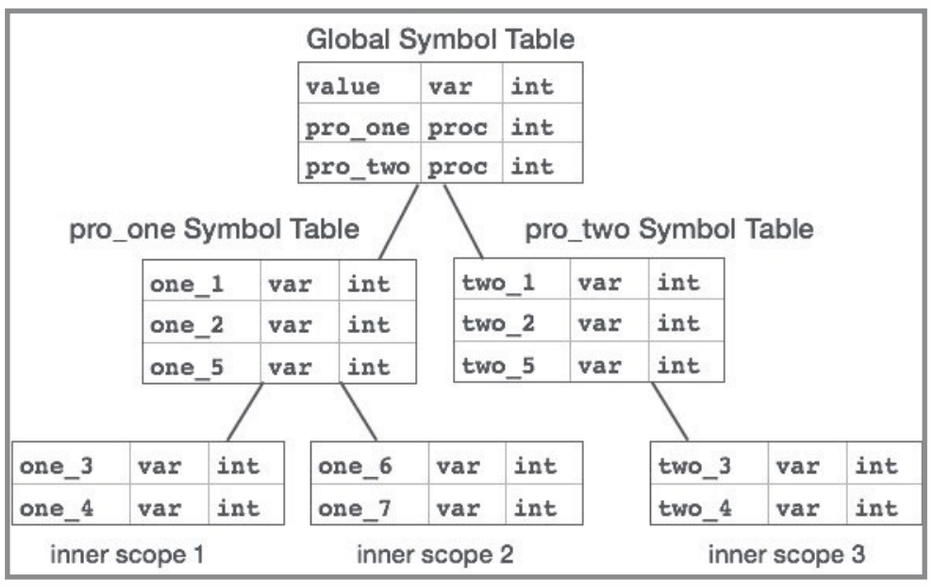
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics



IDE

… with rules encoded in many different ad-hoc ways



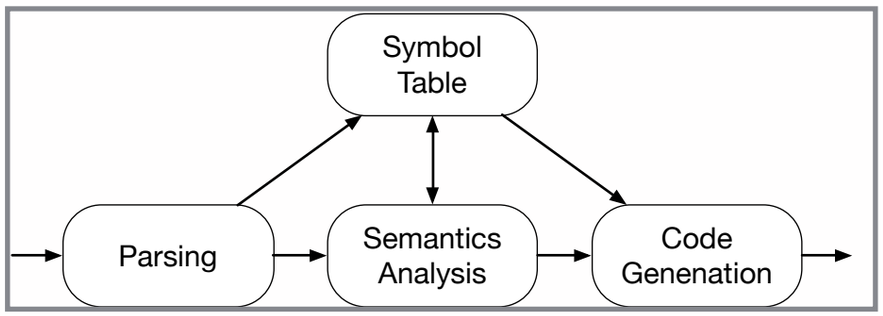
`x:int, Γ`

`lookup(xᵢ)`

`[3/x].σ`

Name Resolution is Pervasive

Appears in many different artifacts…

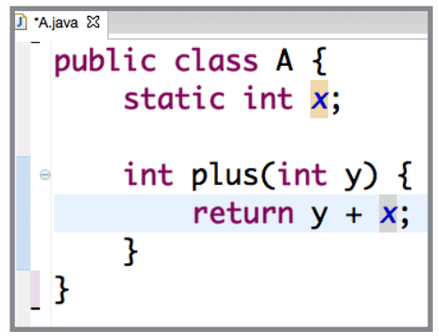


Compiler

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

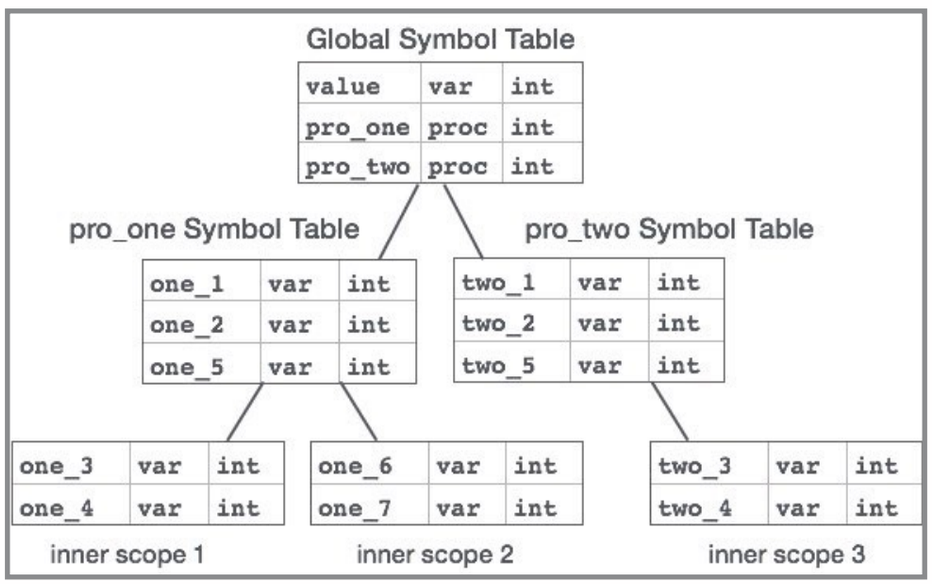
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics



IDE

… with rules encoded in many different ad-hoc ways



`x:int, Γ`

`lookup(xᵢ)`

`[3/x].σ`

No standard approach, no re-use

Contrast with Syntax

Contrast with Syntax

A standard formalism

Context-Free Grammars

Contrast with Syntax

A unique definition

A standard formalism

```
program  =  decl*
   decl  =  module id { decl* }
         |  import qid
         |  def id = exp
    exp  =  qid
         |  fun id { exp }
         |  fix id { exp }
         |  let bind* in exp
         |  letrec bind* in exp
         |  letpar bind* in exp
         |  exp exp
         |  exp ⊕ exp
         |  int
    qid  =  id
         |  id . qid
   bind  =  id = exp
```

Context-Free Grammars

Contrast with Syntax

A unique definition

```
program  =  decl*
   decl  =  module id { decl* }
         |  import qid
         |  def id = exp
    exp  =  qid
         |  fun id { exp }
         |  fix id { exp }
         |  let bind* in exp
         |  letrec bind* in exp
         |  letpar bind* in exp
         |  exp exp
         |  exp ⊕ exp
         |  int
    qid  =  id
         |  id . qid
   bind  =  id = exp
```

A standard formalism

Context-Free Grammars

Supports

- Parser
- AST
- Pretty-Printing
- Highlighting

Representing Bound Programs

Representing Bound Programs

- Many approaches to representing the results of name resolution within an (extended) AST, e.g.
  - numeric indexing [deBruijn72]
  - higher-order abstract syntax [PfenningElliott88]
  - nominal logic approaches [GabbayPitts02]

Representing Bound Programs

- Many approaches to representing the results of name resolution within an (extended) AST, e.g.
  - numeric indexing [deBruijn72]
  - higher-order abstract syntax [PfenningElliott88]
  - nominal logic approaches [GabbayPitts02]
- Good support for binding-sensitive AST manipulation

Representing Bound Programs

- Many approaches to representing the results of name resolution within an (extended) AST, e.g.
  - numeric indexing [deBruijn72]
  - higher-order abstract syntax [PfenningElliott88]
  - nominal logic approaches [GabbayPitts02]

- Good support for binding-sensitive AST manipulation
- But: Do not say how to resolve identifiers in the first place!
  - Also: Can’t represent ill-bound programs
  - And: Tend to be biased towards lambda-like bindings

Binding Specification Languages

Binding Specification Languages

- Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  - Ott [Sewell+10]
  - Romeo [StansiferWand14]
  - Unbound [Weirich+11]
  - Cαml [Pottier06]
  - NaBL [Konat+12]

Binding Specification Languages

- Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  - Ott [Sewell+10]
  - Romeo [StansiferWand14]
  - Unbound [Weirich+11]
  - Cαml [Pottier06]
  - NaBL [Konat+12]
- Generate code to do resolution and record results

Binding Specification Languages

- Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  - Ott [Sewell+10]
  - Romeo [StansiferWand14]
  - Unbound [Weirich+11]
  - Cαml [Pottier06]
  - NaBL [Konat+12]
- Generate code to do resolution and record results
- But: what are the **semantics** of such a language?

The Missing Piece

The Missing Piece

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

The Missing Piece

- Answer: 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...

The Missing Piece

- Answer: 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

The Missing Piece

- Answer: 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)

Design Goals

Design Goals

- Handle broad range of language binding features...

Design Goals

- Handle broad range of language binding features...
- ...using minimal number of constructs

Design Goals

- Handle broad range of language binding features...
- ...using minimal number of constructs
- Make resolution structure language-independent

Design Goals

- 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

Design Goals

- 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

A Theory of Name Resolution

A Theory of Name Resolution

*For **statically lexically scoped** languages*

A Theory of Name Resolution

*For **statically lexically scoped** languages*

A standard formalism

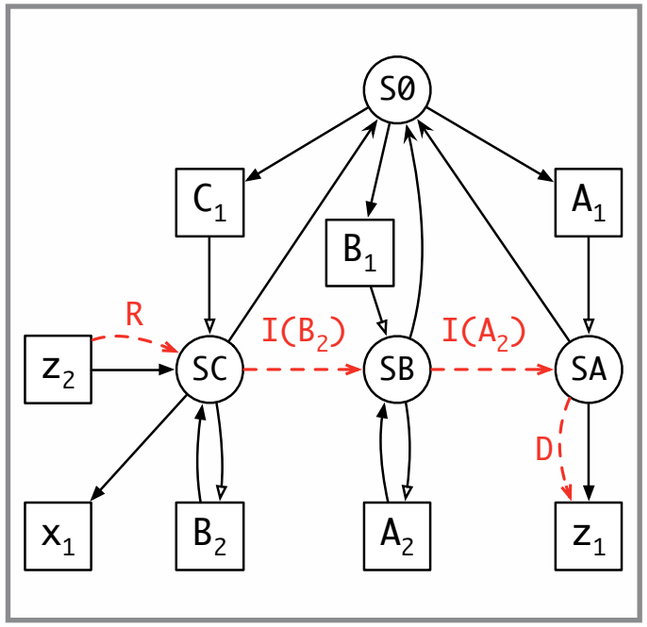
Scope Graphs

A Theory of Name Resolution

*For **statically lexically scoped** languages*

A unique representation

A standard formalism



Scope Graphs

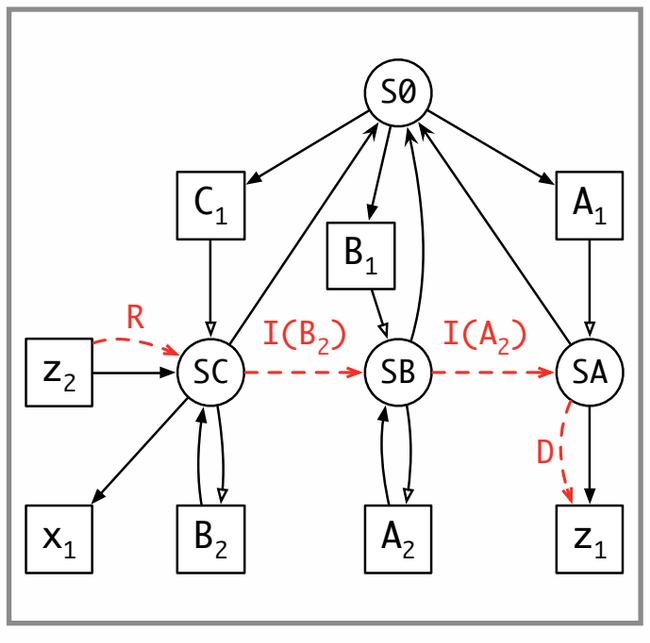
A Theory of Name Resolution

*For **statically lexically scoped** languages*

A unique representation

A standard formalism

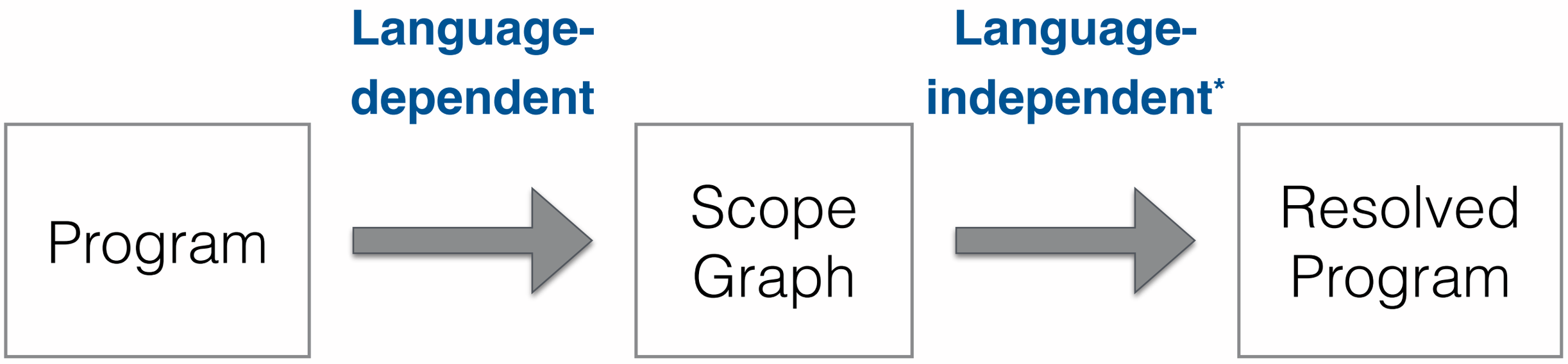
Supports



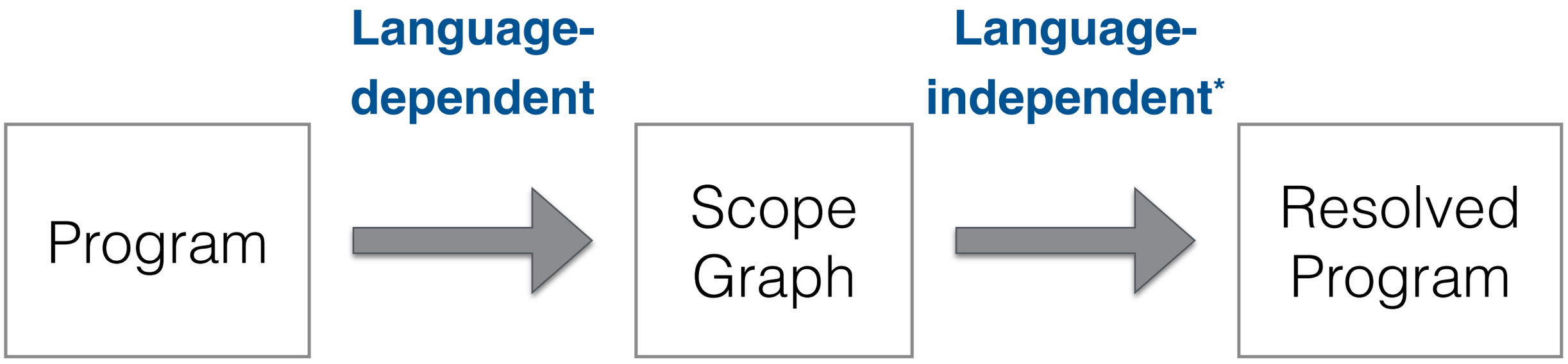
Scope Graphs

- Resolution
- α-equivalence
- IDE Navigation
- Refactoring tools
- Reasoning tools

Resolution Scheme



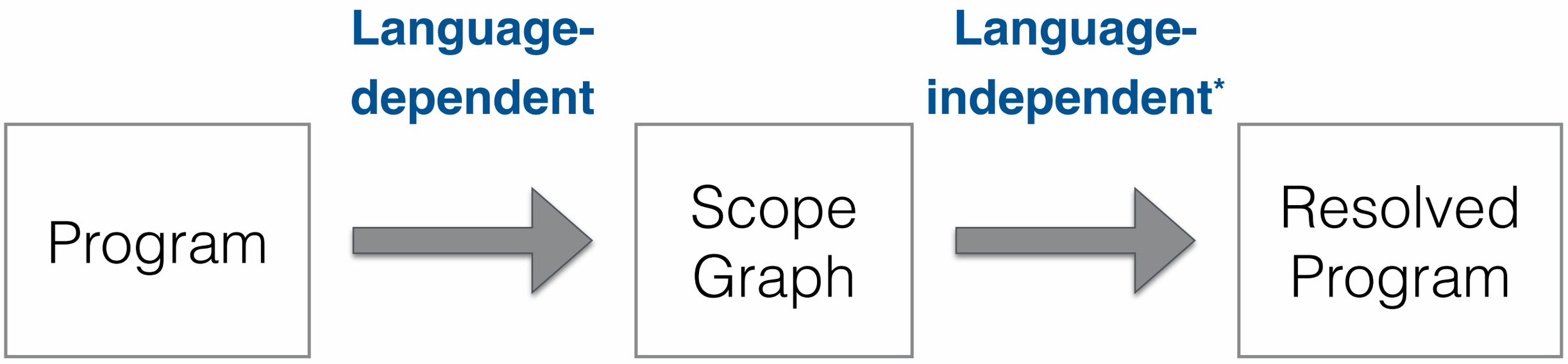
Resolution Scheme



Resolution of a reference in a scope graph:

Building a **path**
from a **reference** node
to a **declaration** node
following path construction **rules**

Resolution Scheme



Resolution of a reference in a scope graph:

Building a **path**
from a **reference** node
to a **declaration** node
following path construction **rules**

*Parameterized by notions of path **well-formedness** and **ordering**

Scope Graphs by Example

Simple Scopes

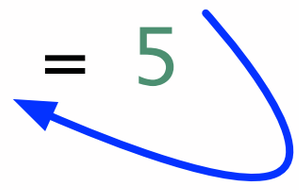
```
def y = x + 1
def x = 5
```

Simple Scopes

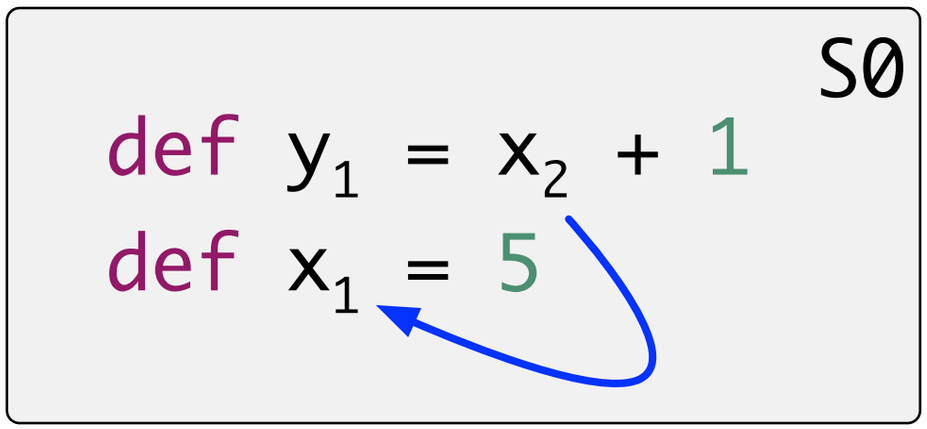
```
def y1 = x2 + 1
def x1 = 5
```

Simple Scopes

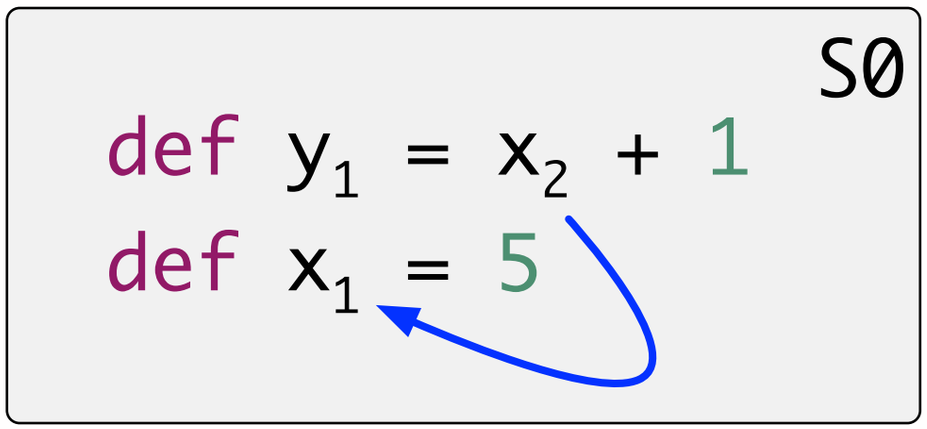
```
def y1 = x2 + 1
def x1 = 5
```

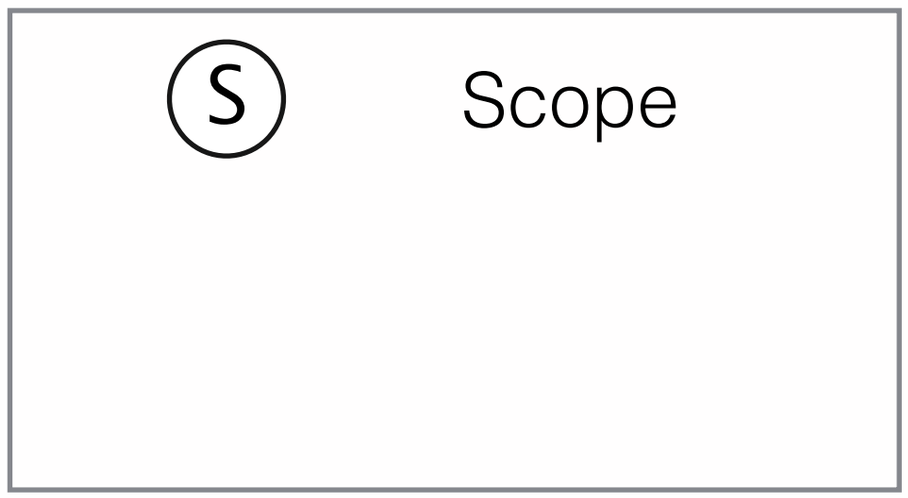


Simple Scopes



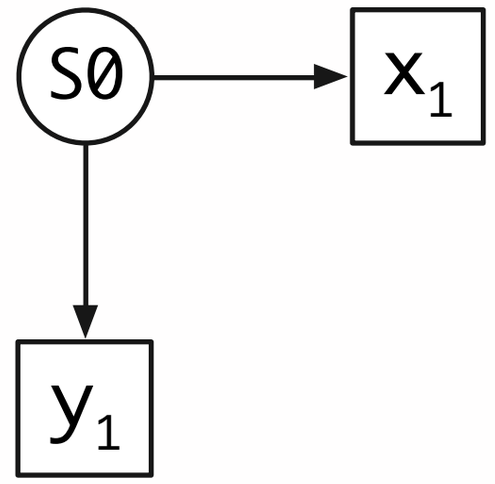
Simple Scopes

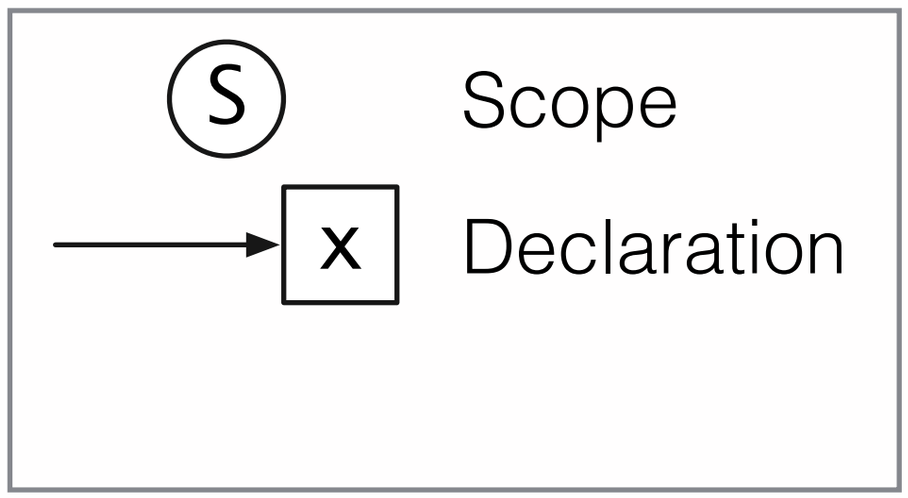




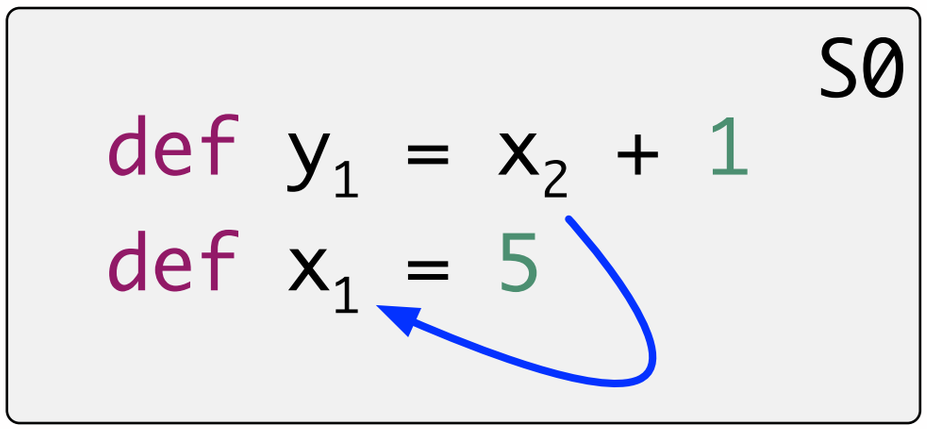
Simple Scopes

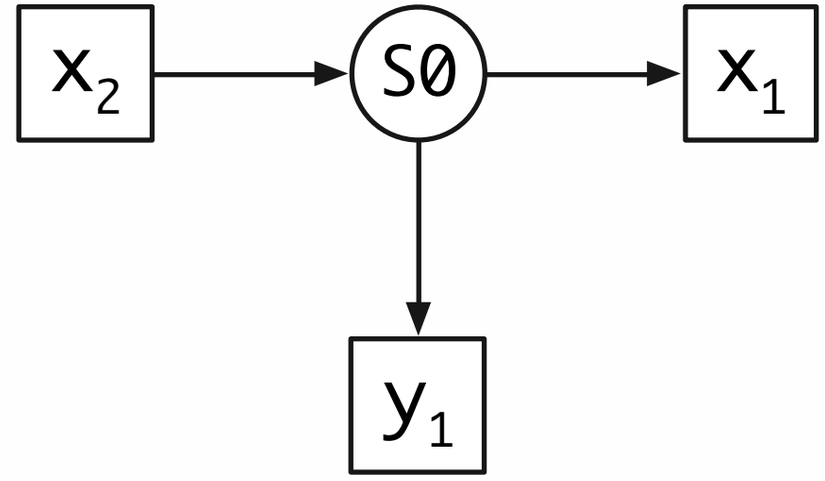
```
                                    S0
def y1 = x2 + 1
def x1 = 5
```

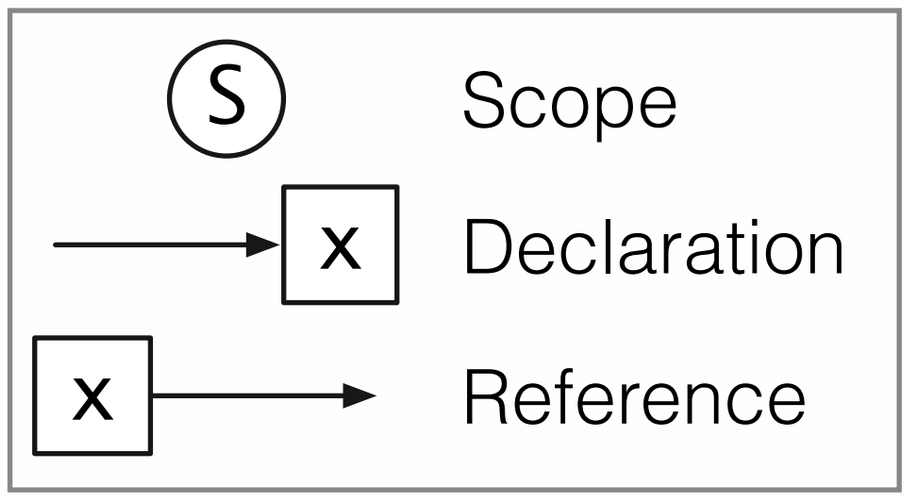




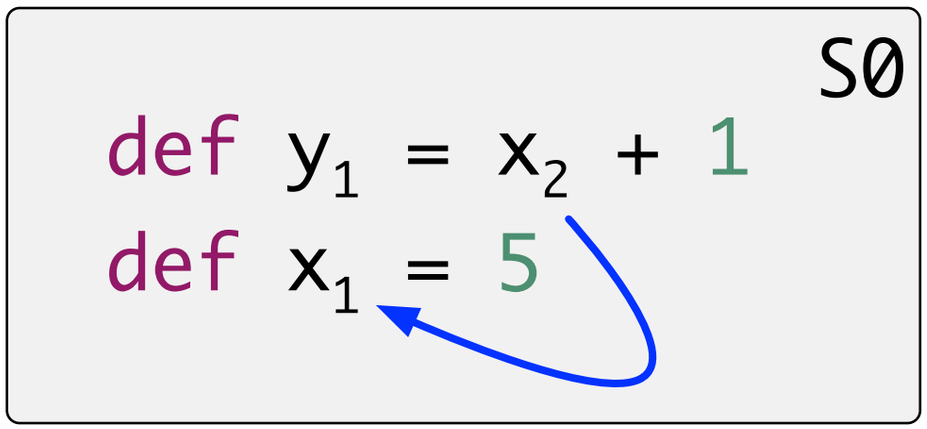
Simple Scopes

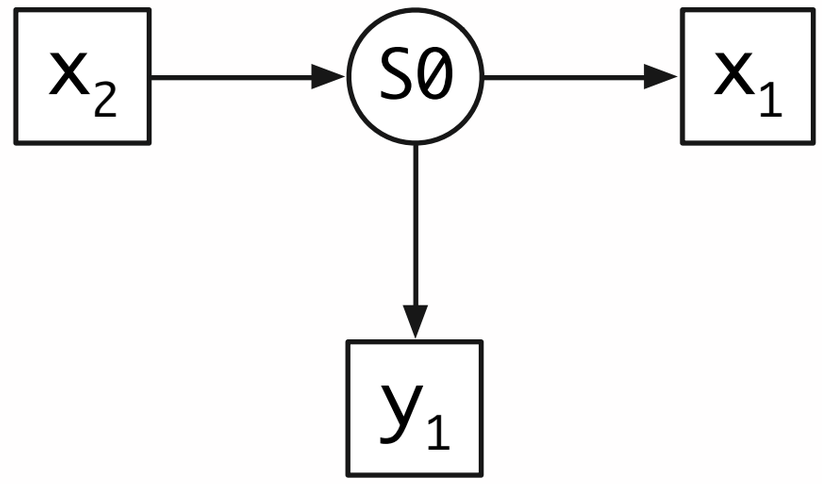


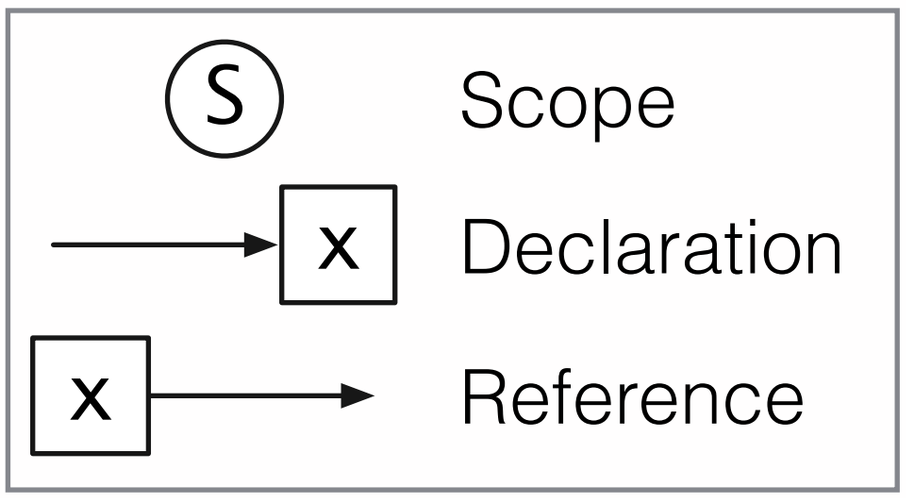


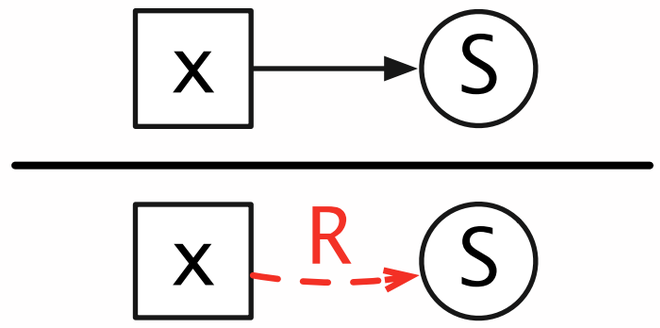


Simple Scopes



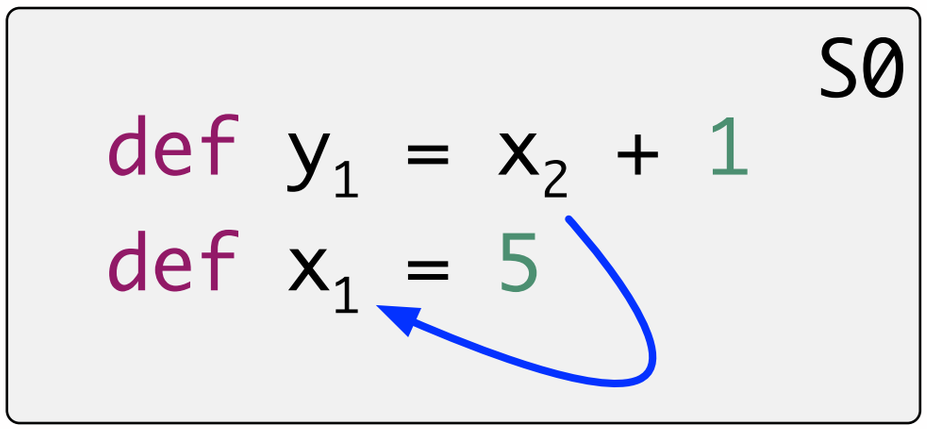


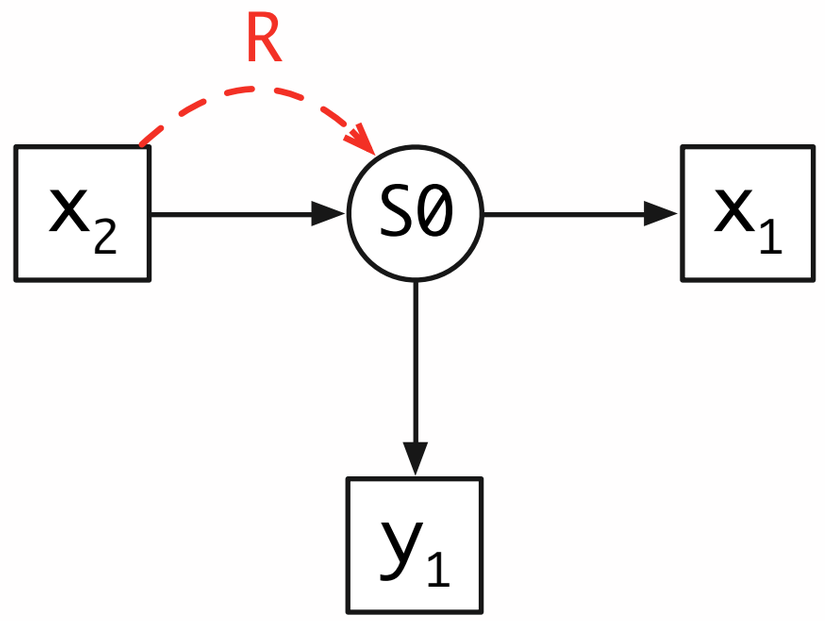


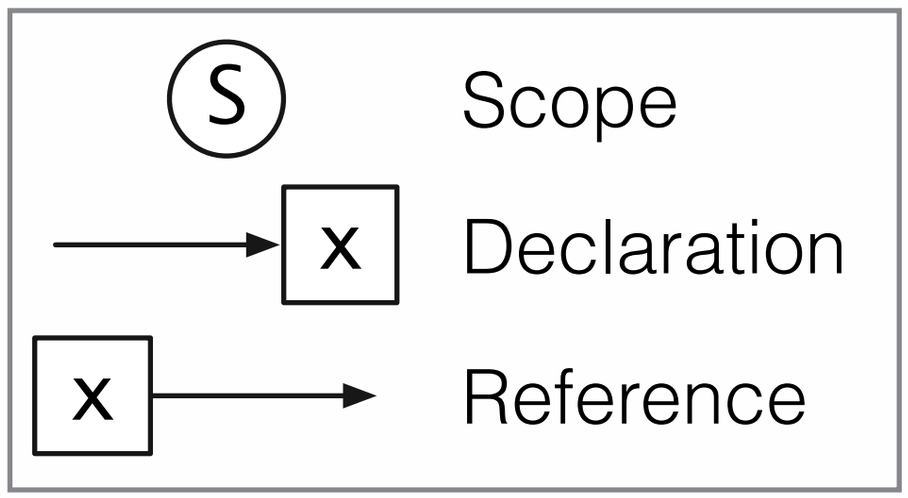


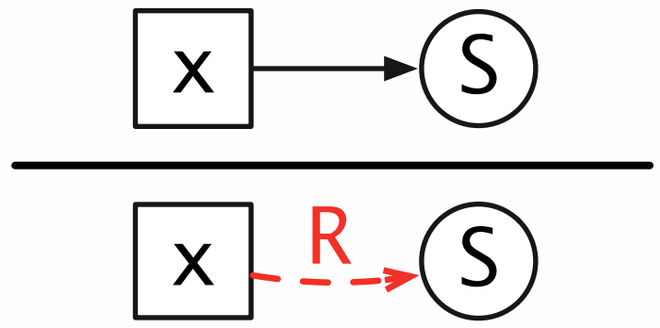
Reference Step

Simple Scopes



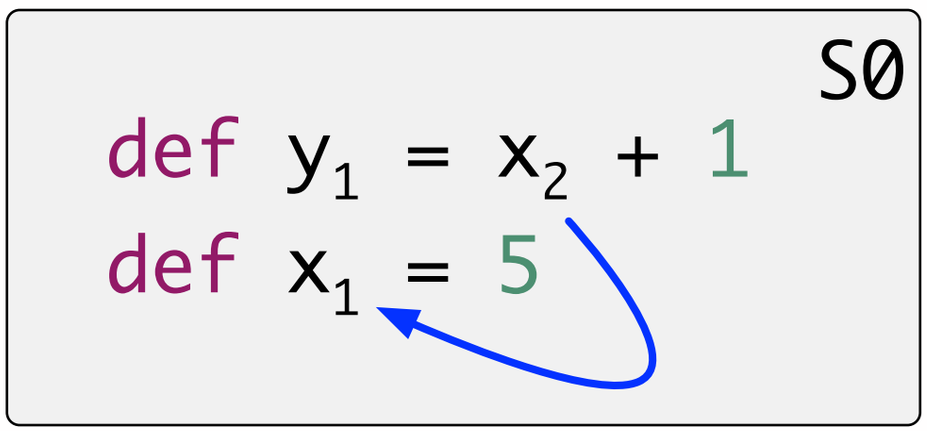


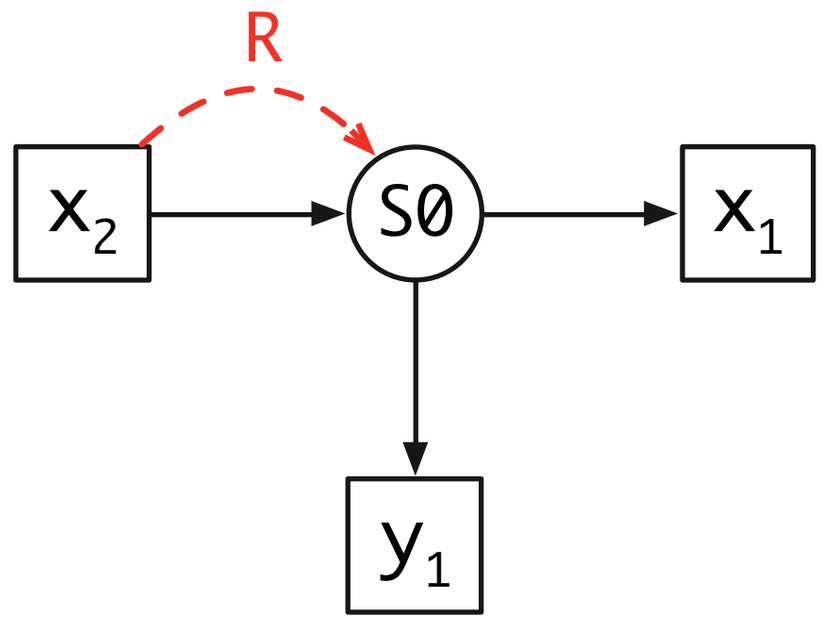


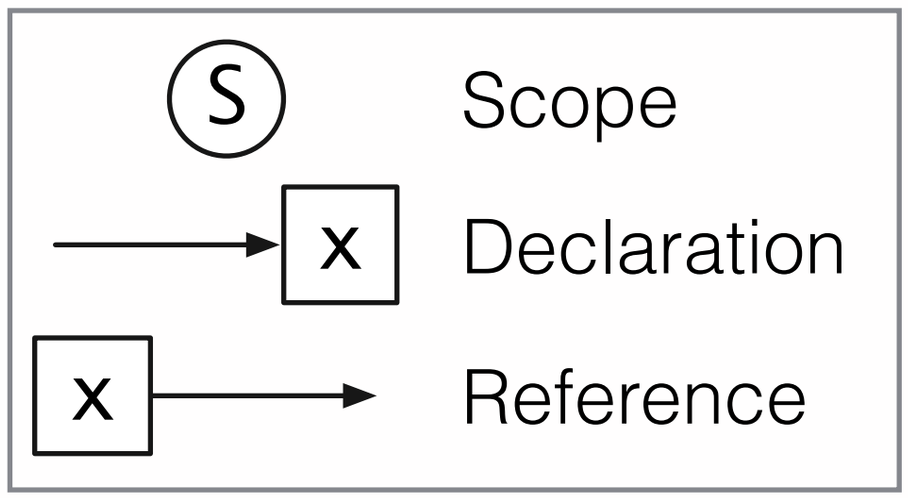


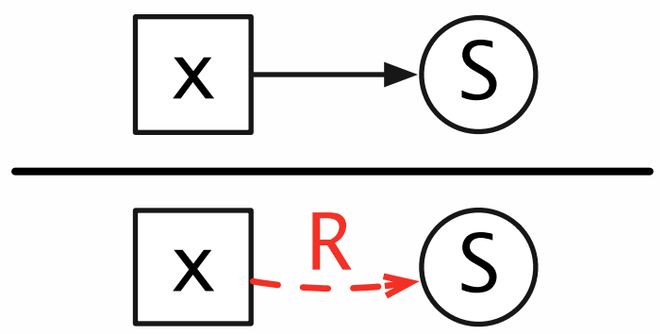
Reference Step

Simple Scopes

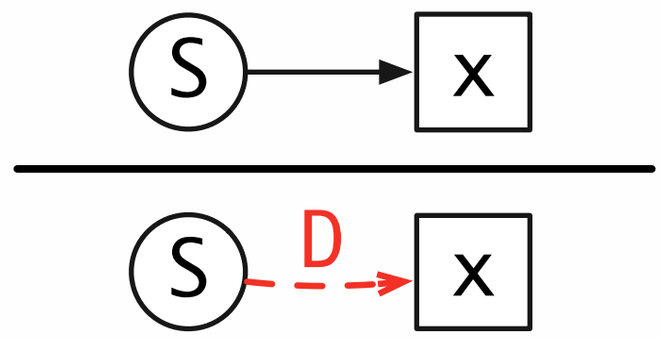






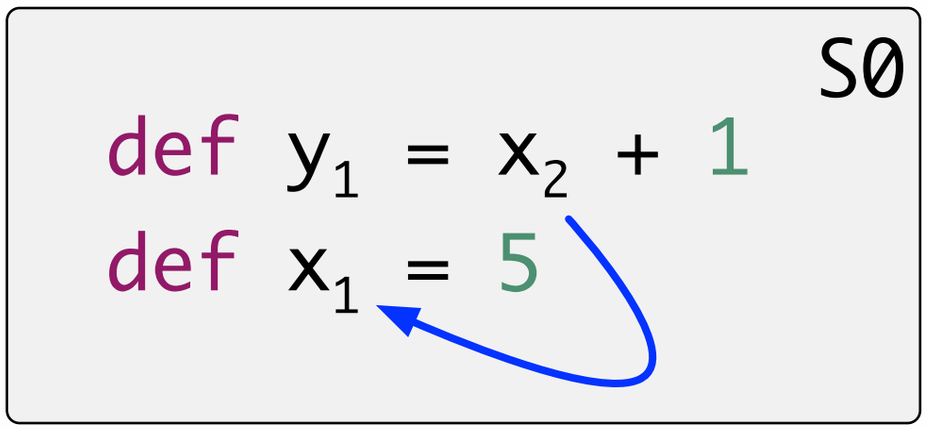


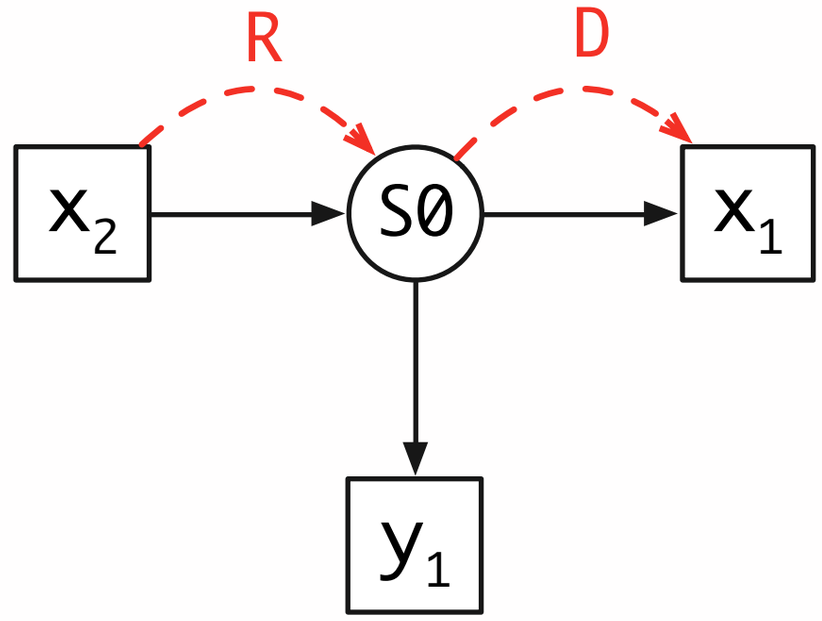
Reference Step

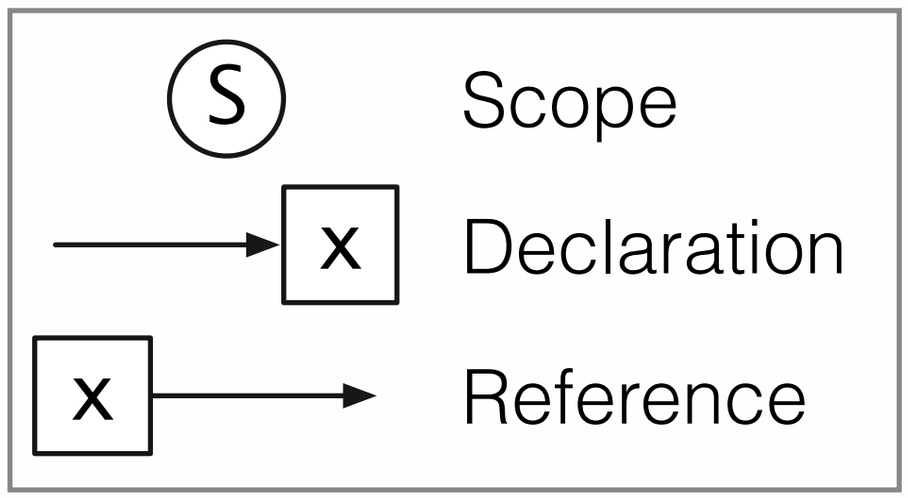


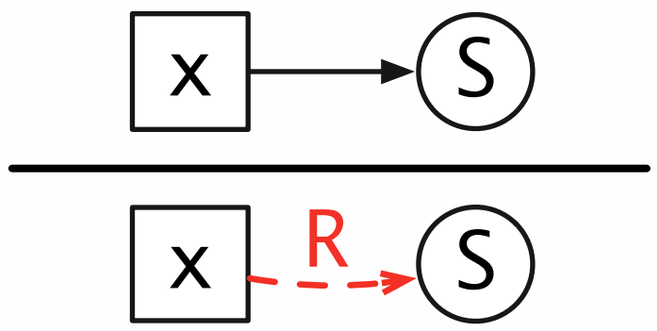
Declaration Step

Simple Scopes

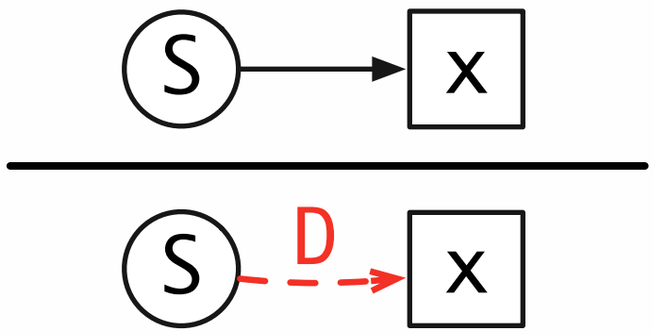








Reference Step



Declaration Step

Ambiguous Resolutions

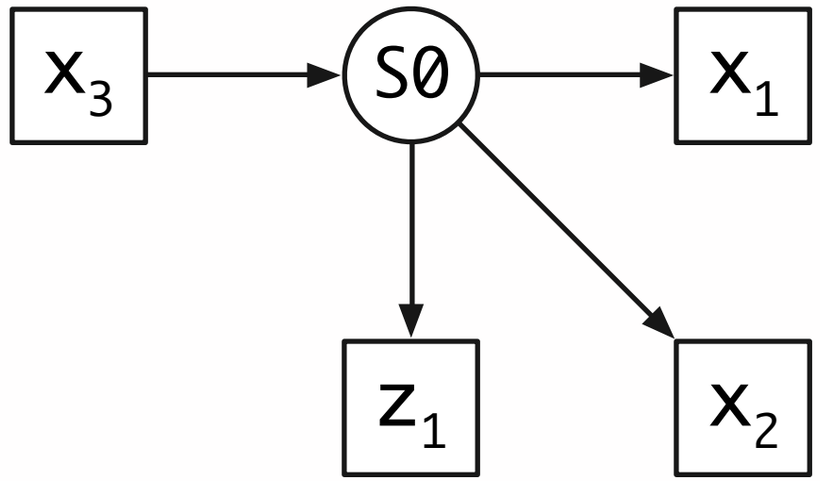
S0

```
def x₁ = 5

def x₂ = 3

def z₁ = x₃ + 1
```

Ambiguous Resolutions

S0

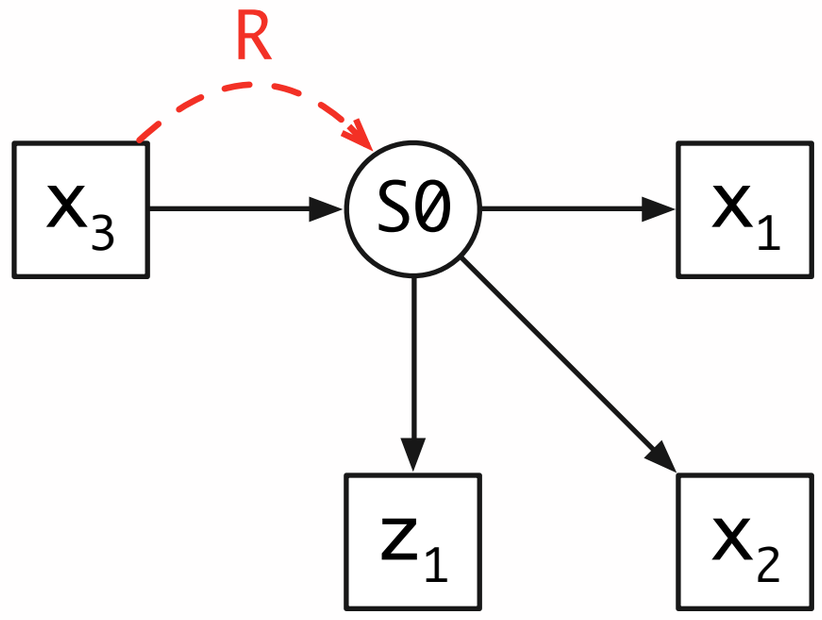
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

S0

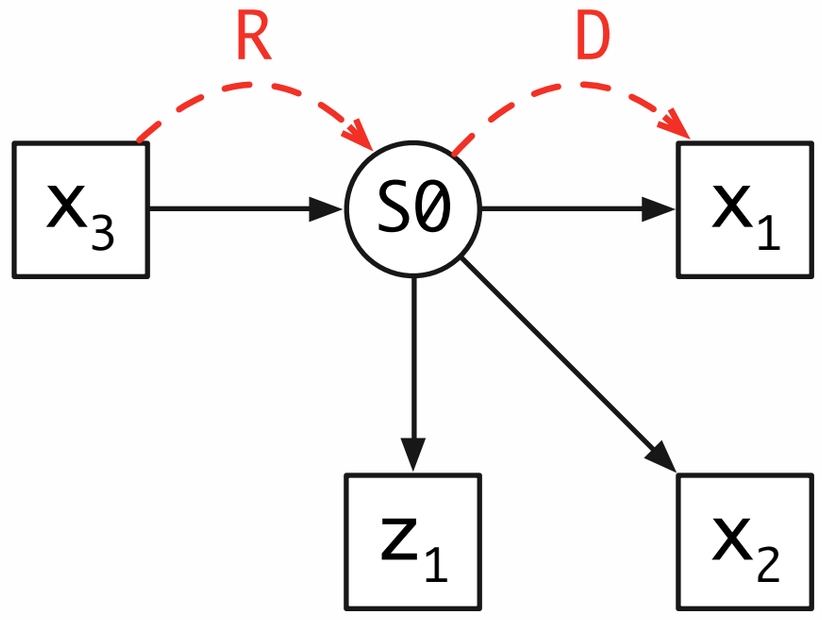
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

S0

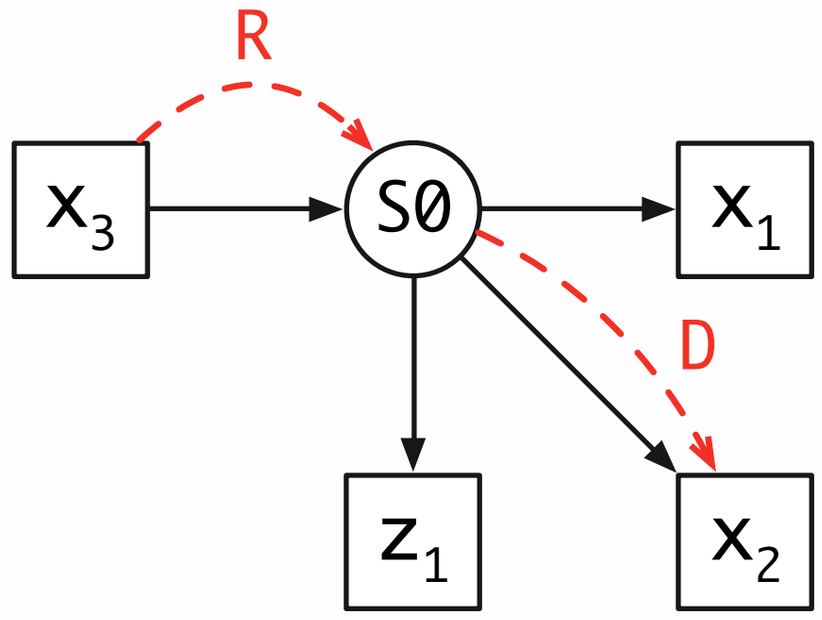
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

S0

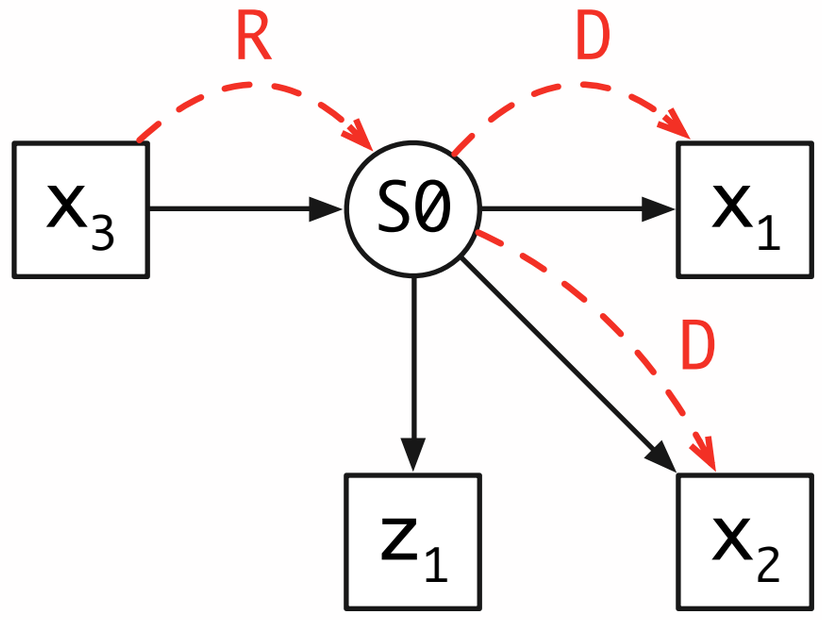
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

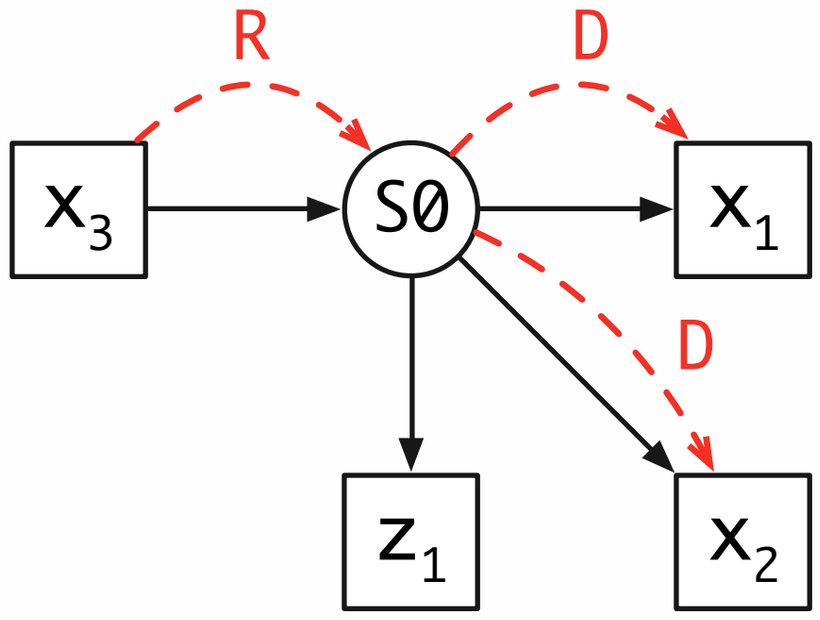
S0

```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

```
                                    S0
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```




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

Lexical Scoping

```
def x1 = z2 5

def z1 =
  fun y1 {
    x2 + y2
  }
```

Lexical Scoping

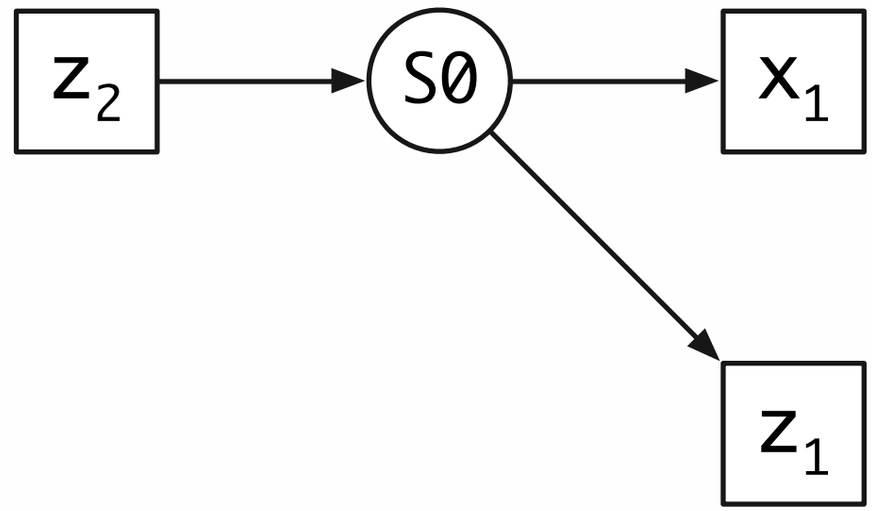
```
                                S0
def x1 = z2 5


def z1 =
  fun y1 {
    x2 + y2
  }
```

Lexical Scoping

```
                                S0
def x1 = z2 5

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

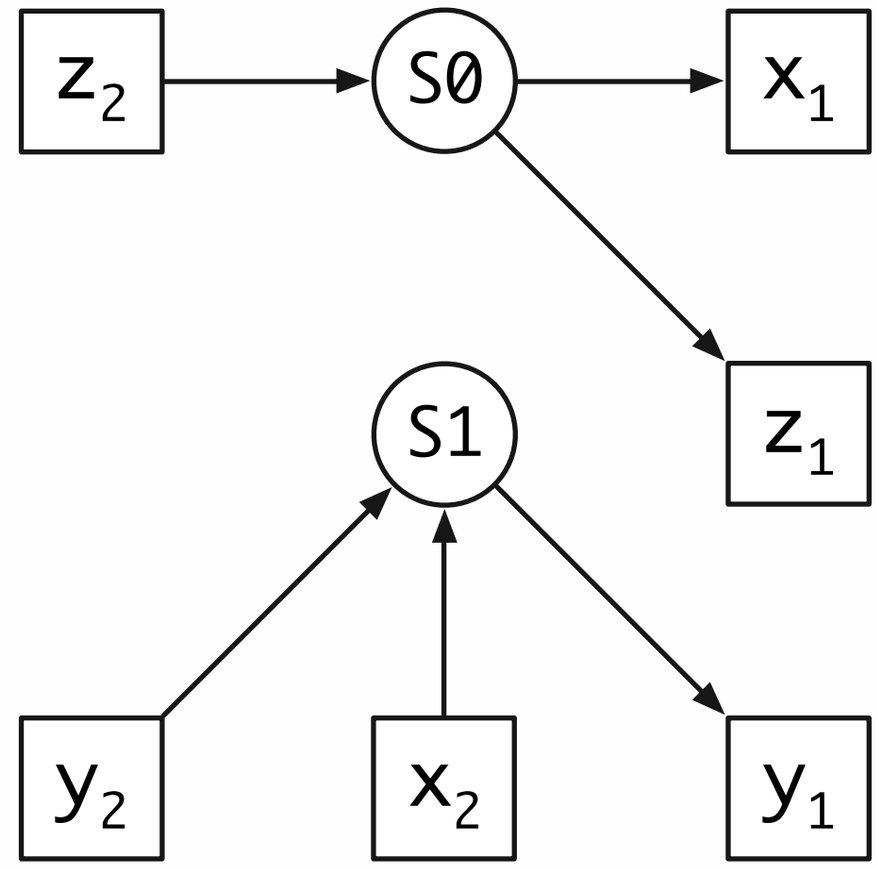
Lexical Scoping

```
                                    S0
def x1 = z2 5

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



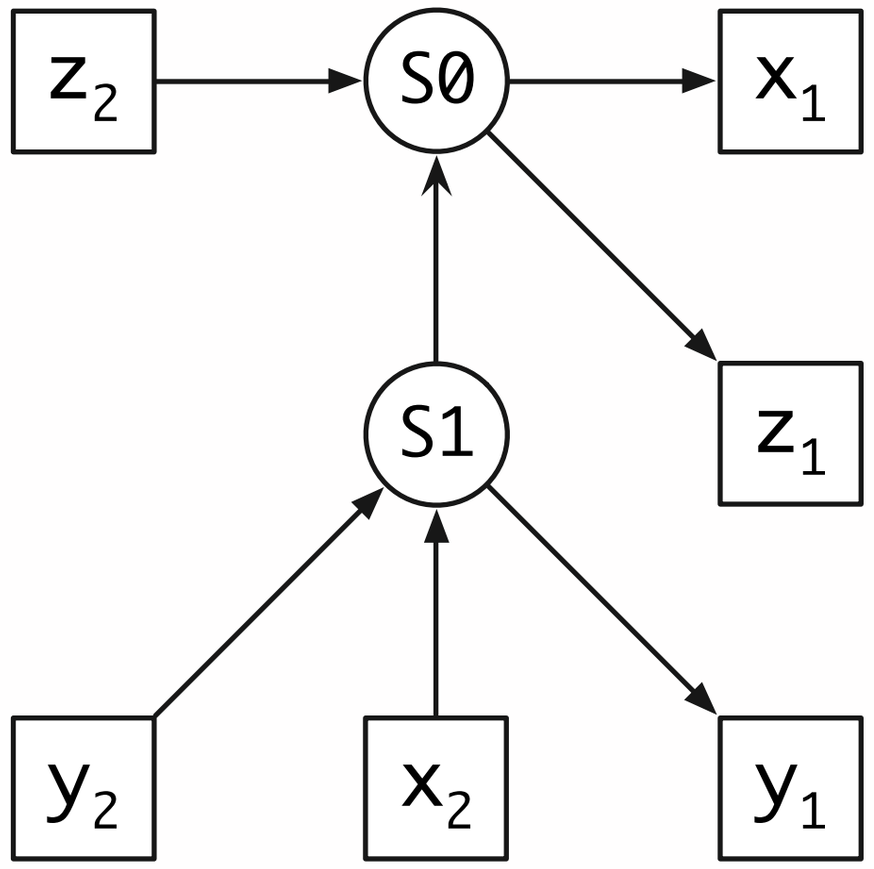
Lexical Scoping

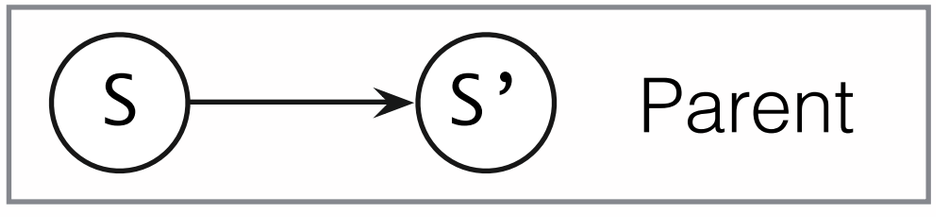
```
                                  S0
def x1 = z2 5

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



Lexical Scoping

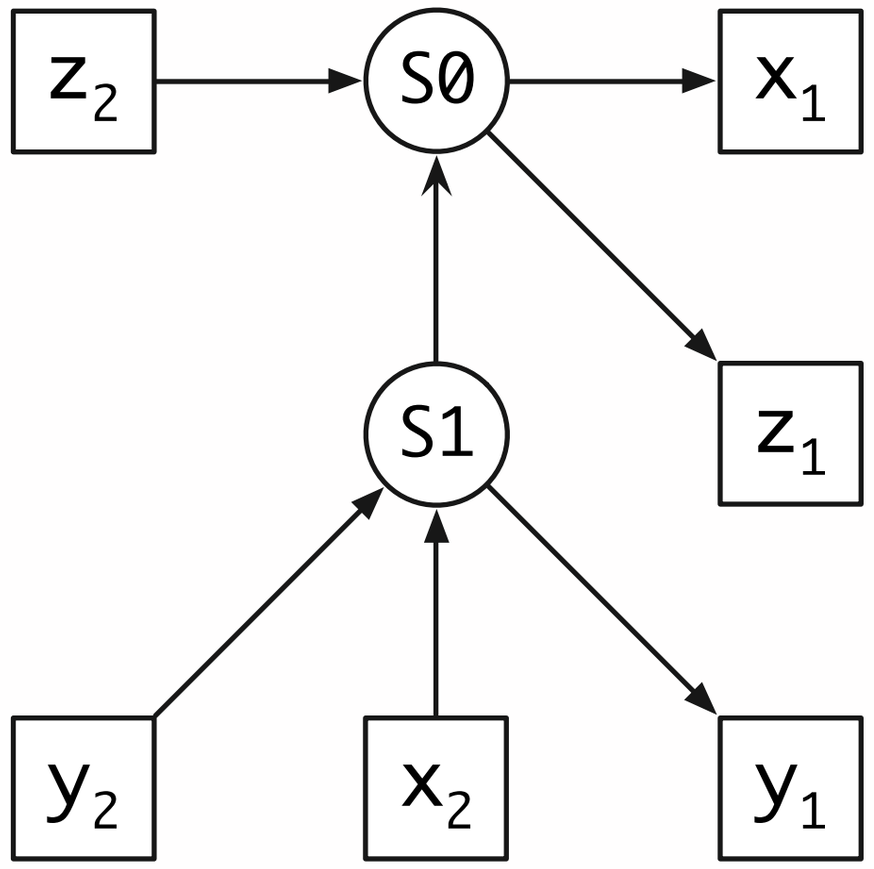
```
                                  S0
def x1 = z2 5

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

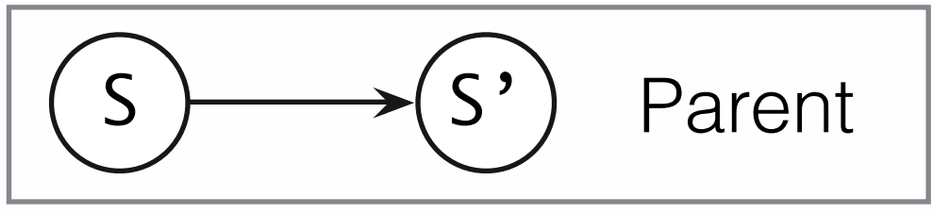




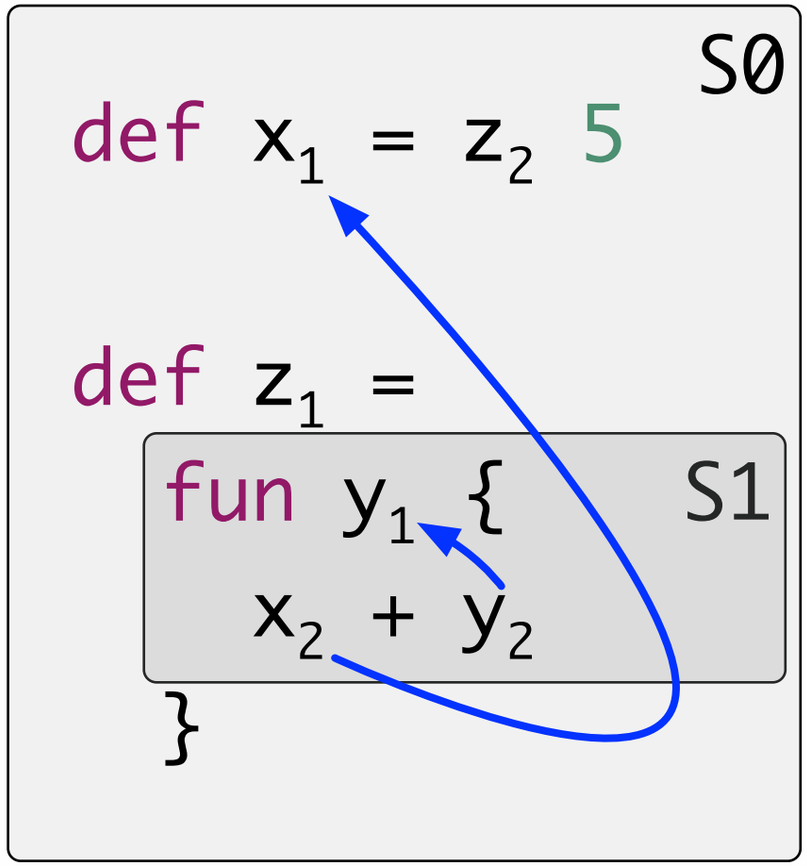
Lexical Scoping

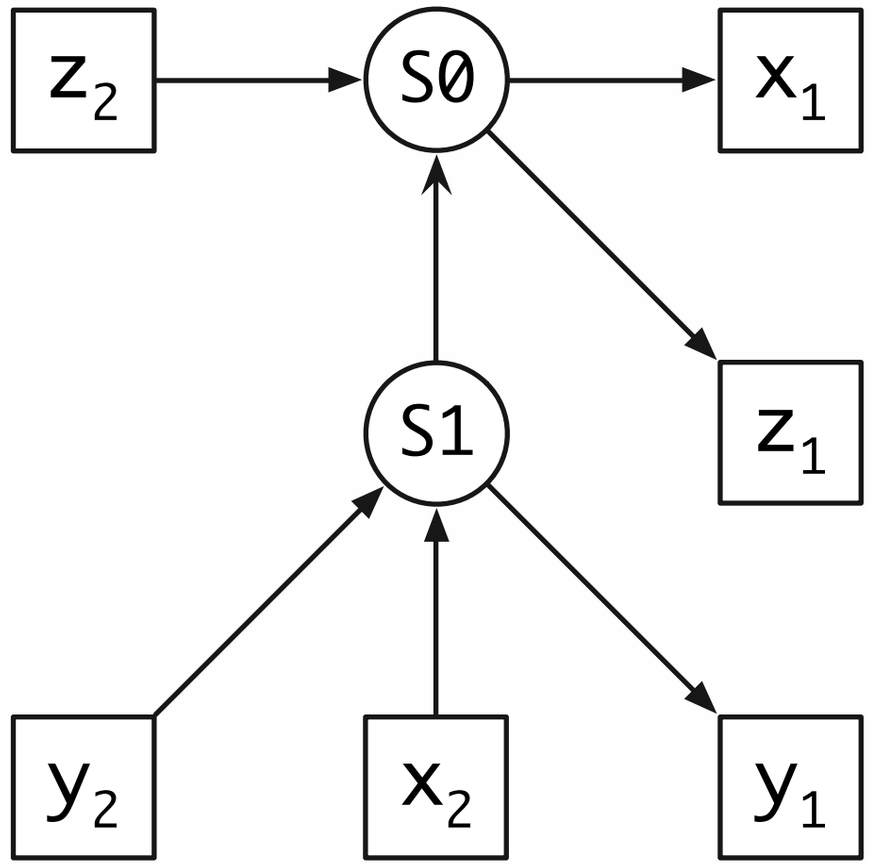
```
                                  S0
def x1 = z2 5

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

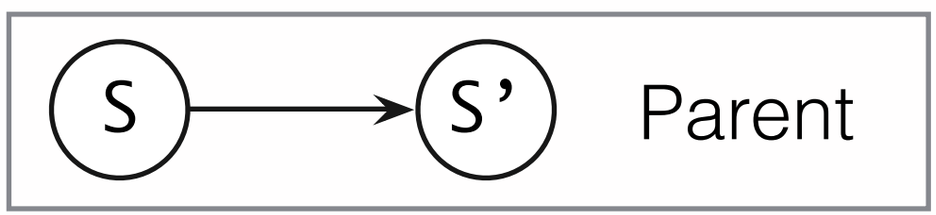




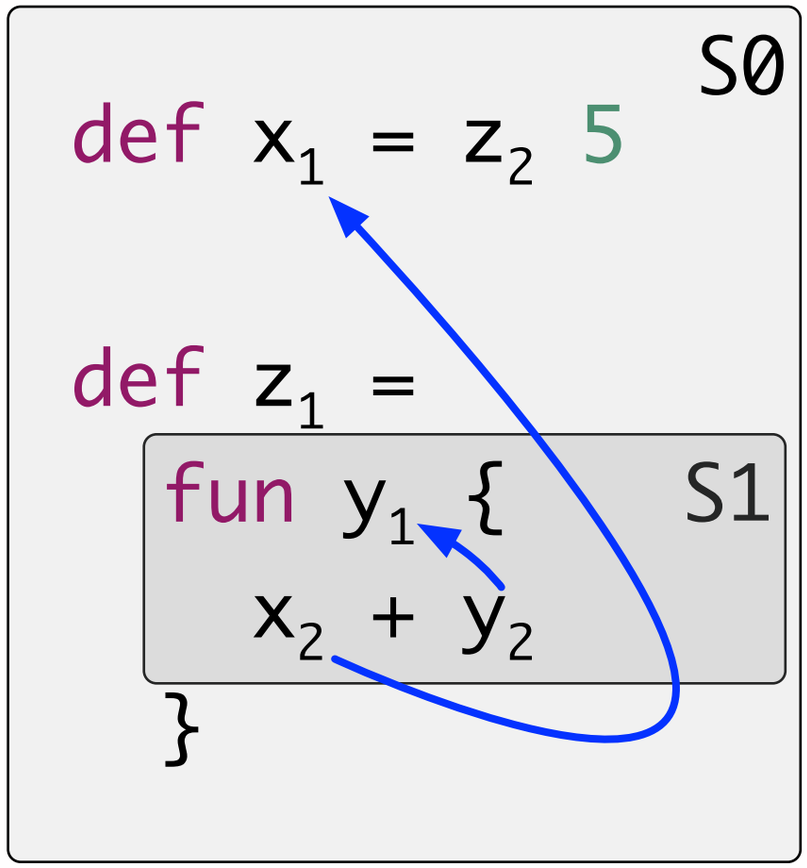
Lexical Scoping

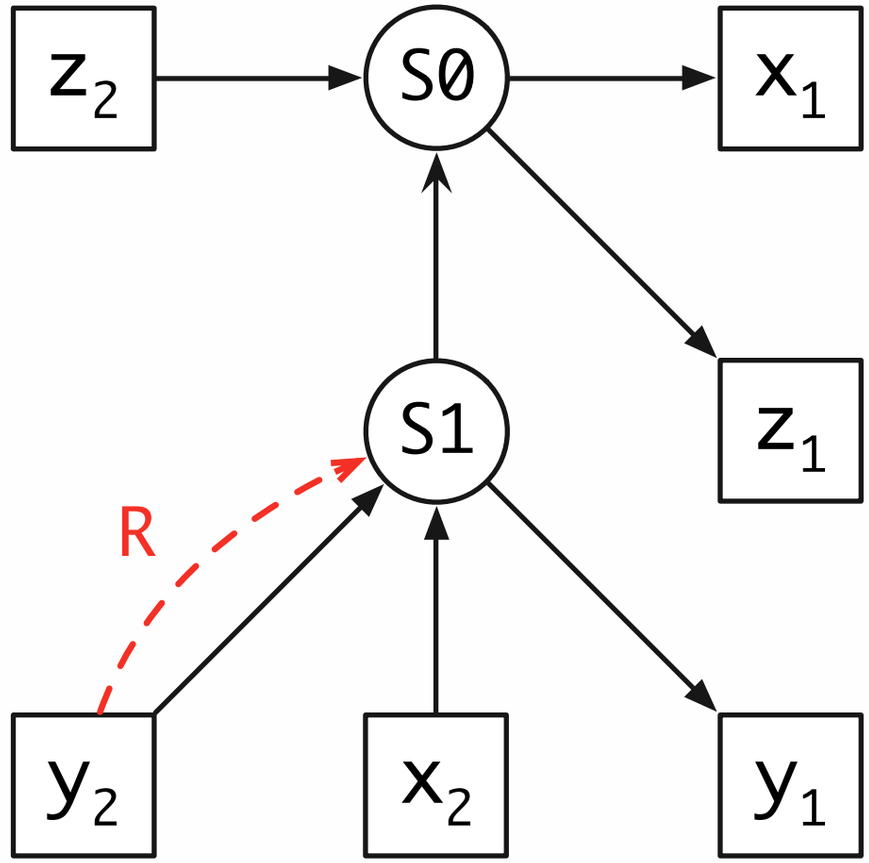


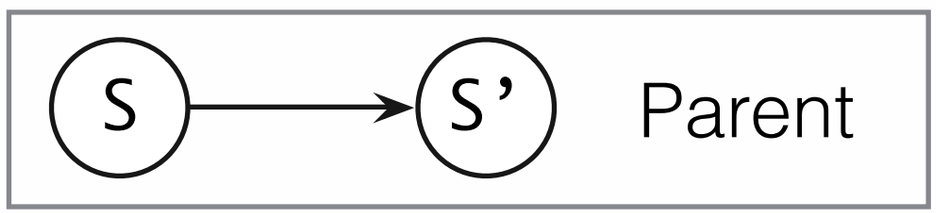




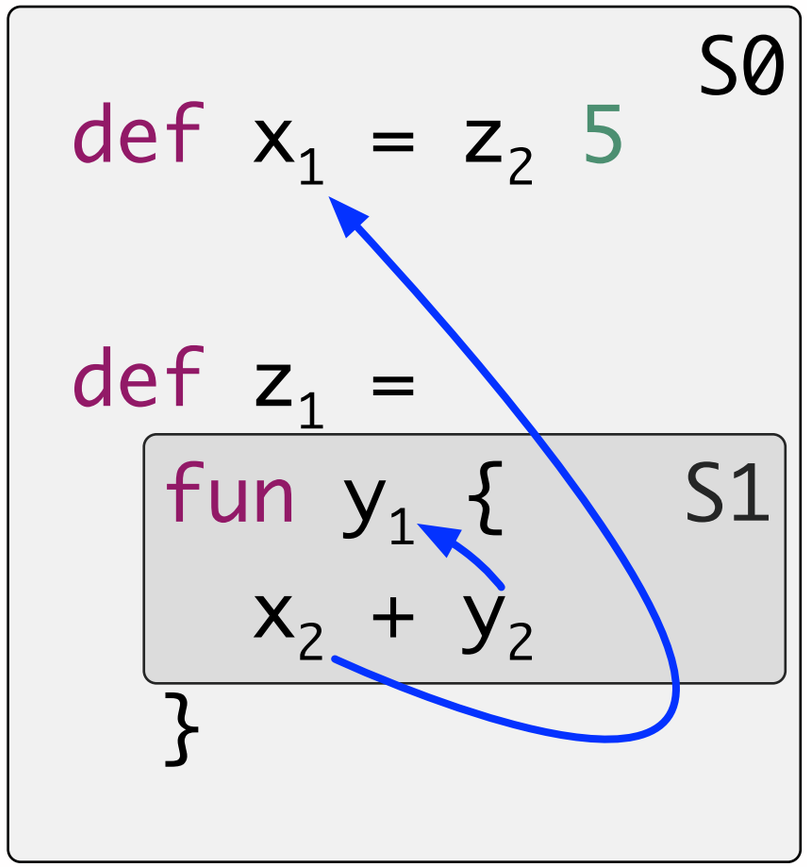
Lexical Scoping

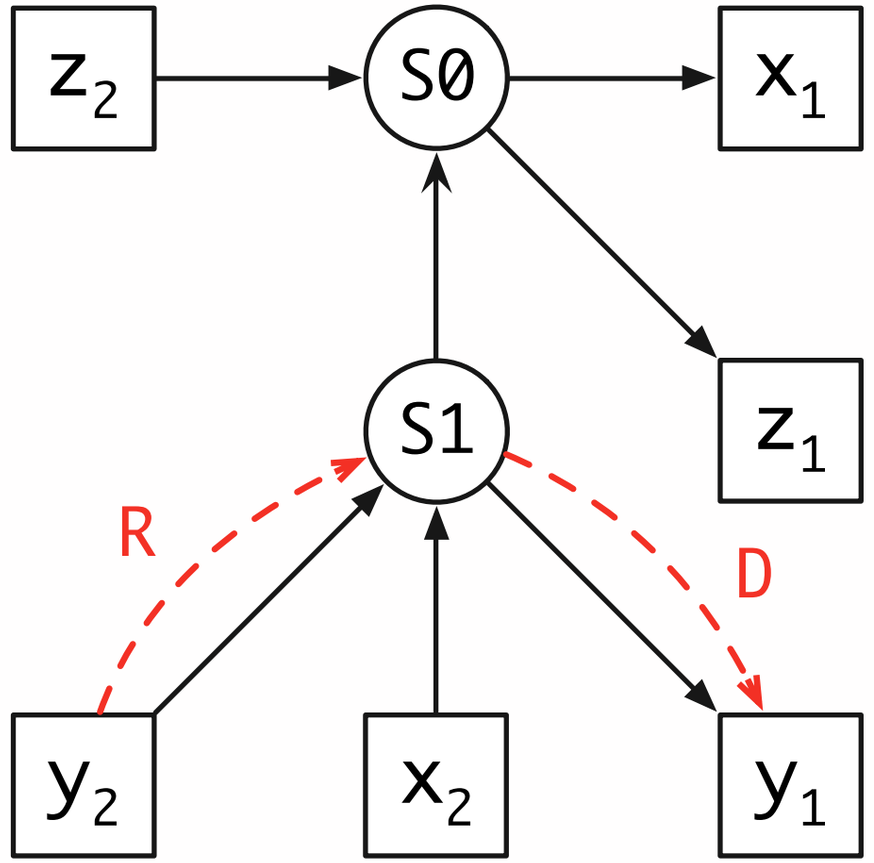


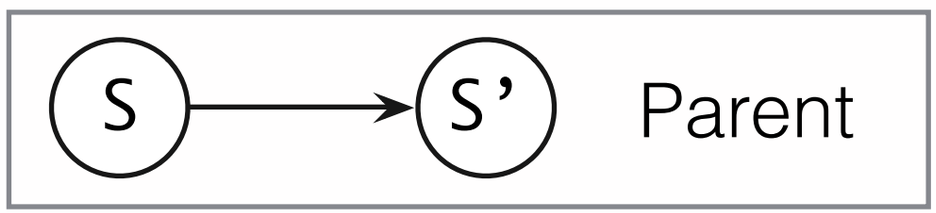




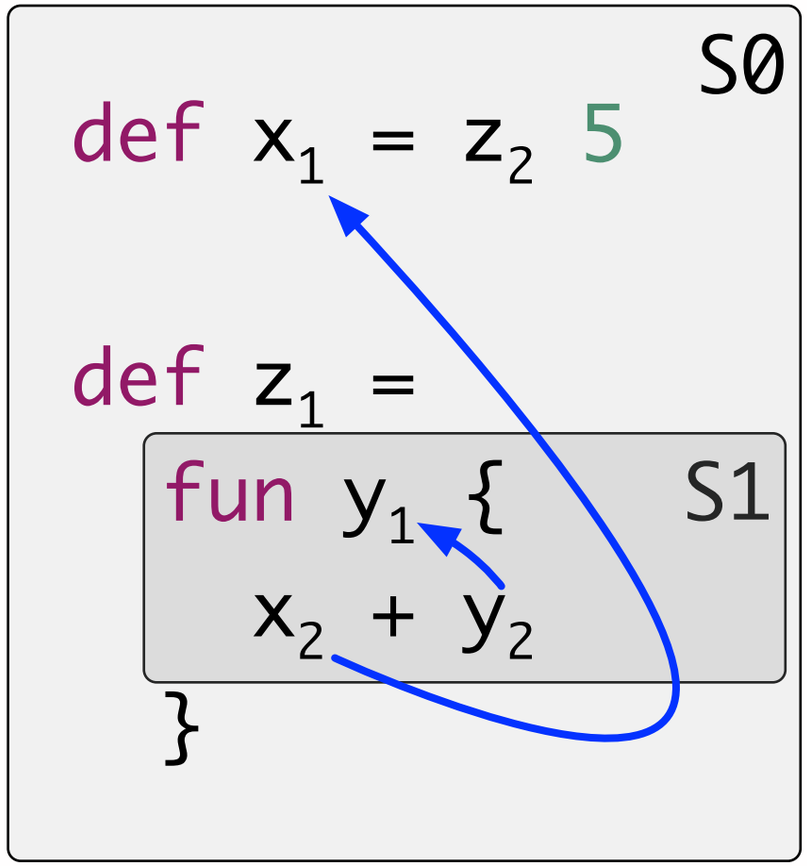
Lexical Scoping

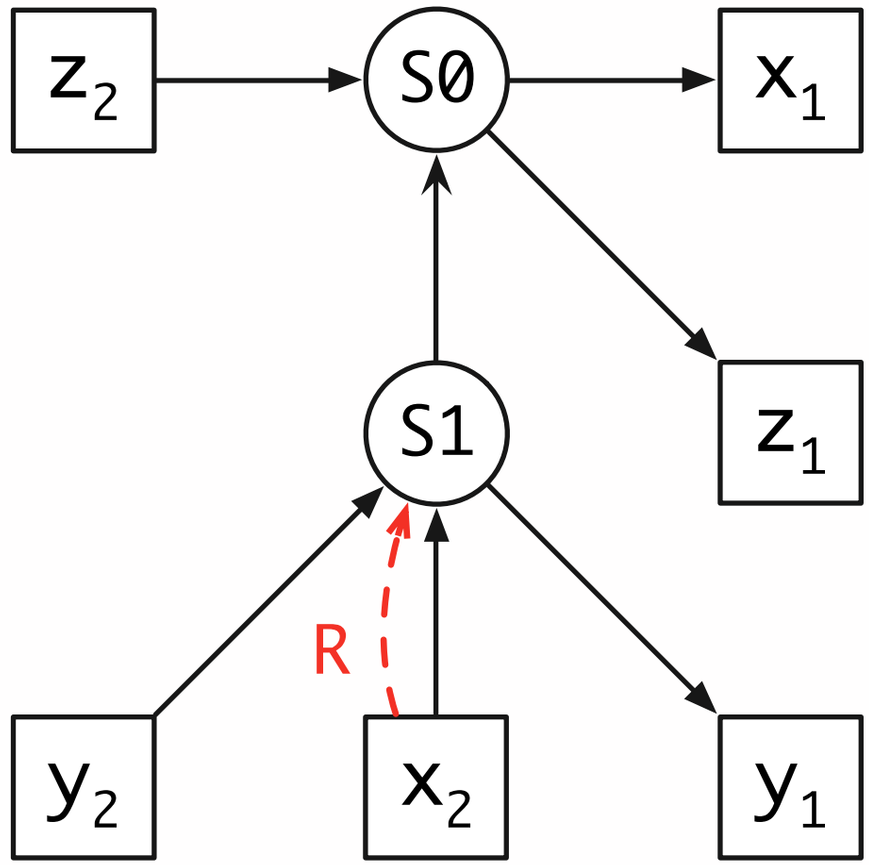


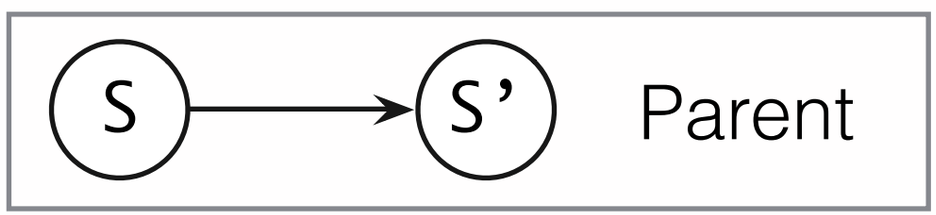




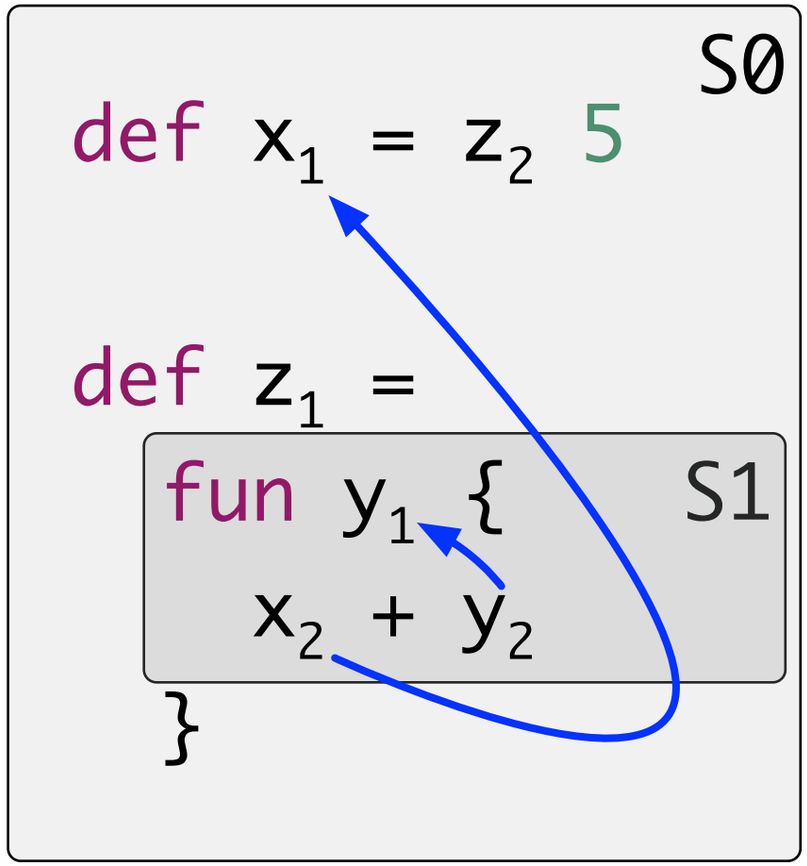
Lexical Scoping

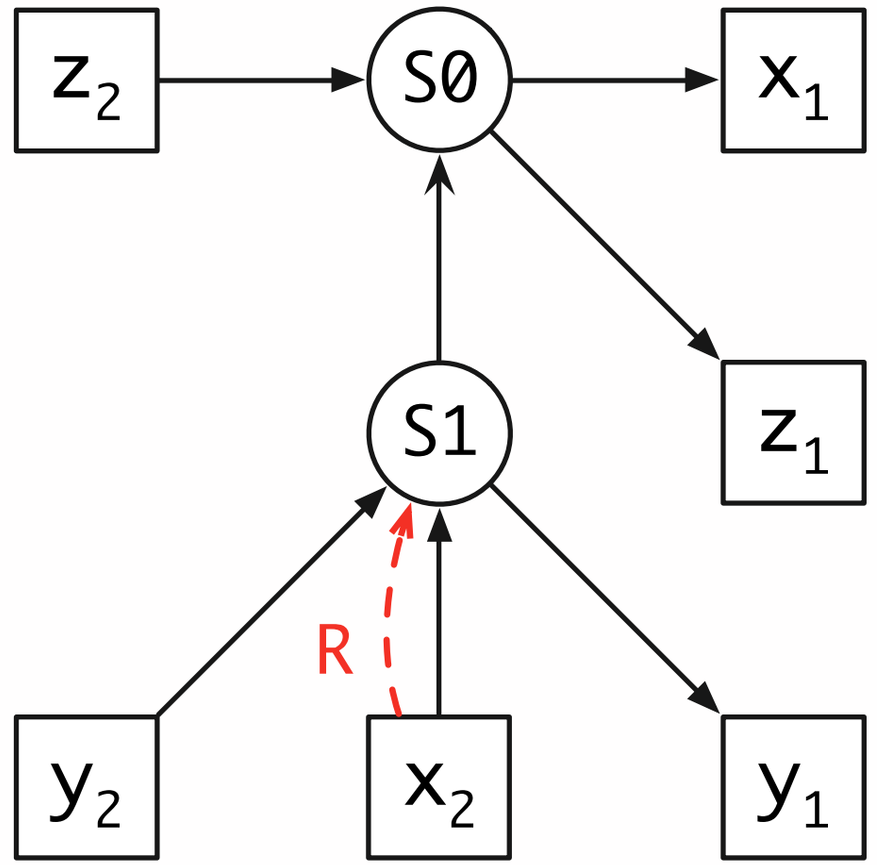


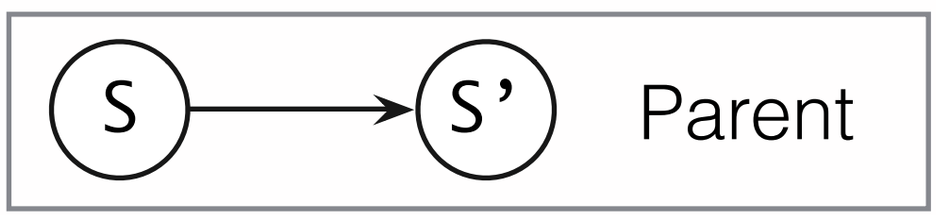


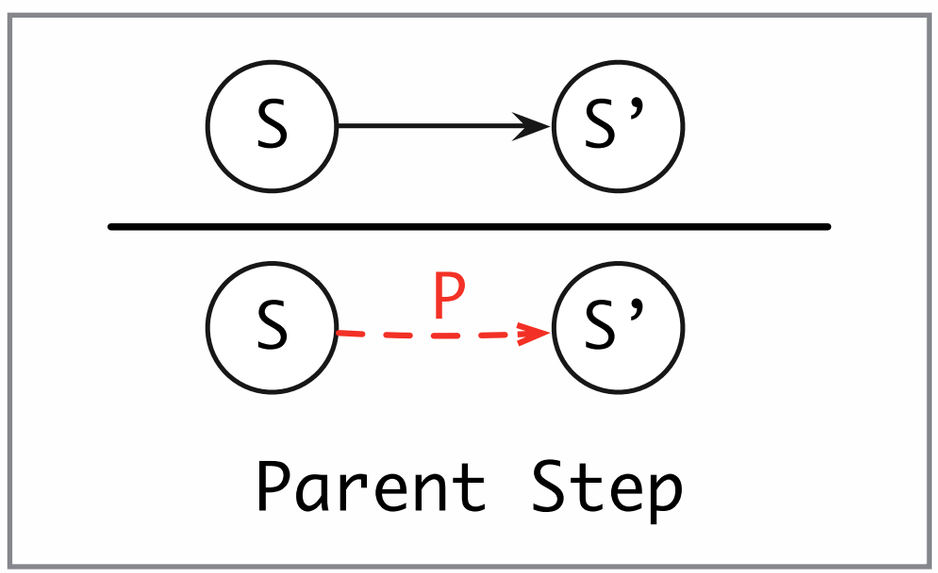


Lexical Scoping

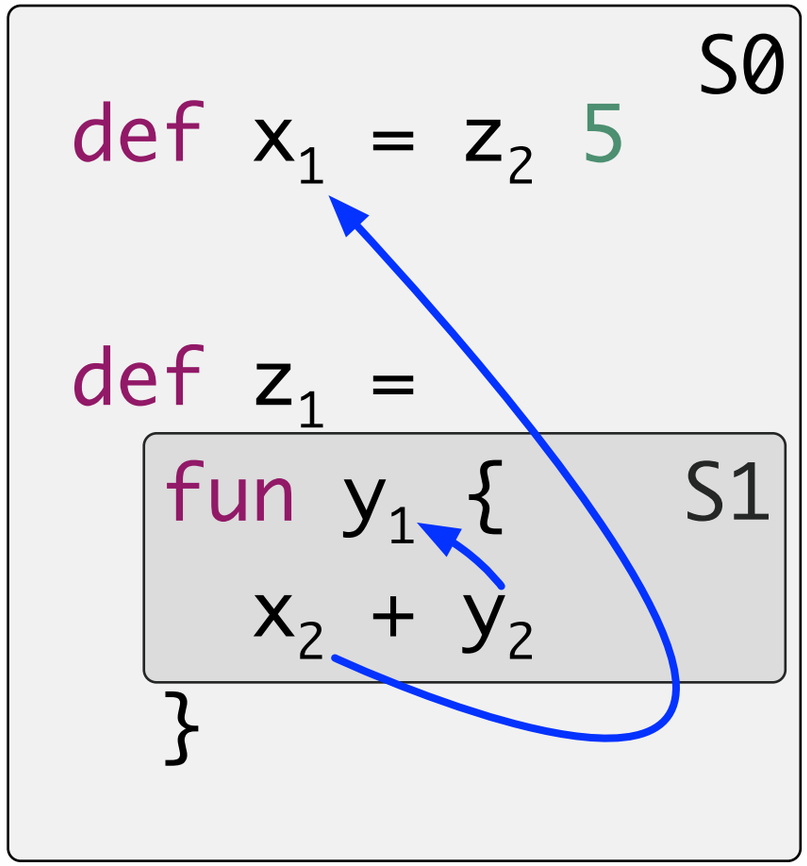


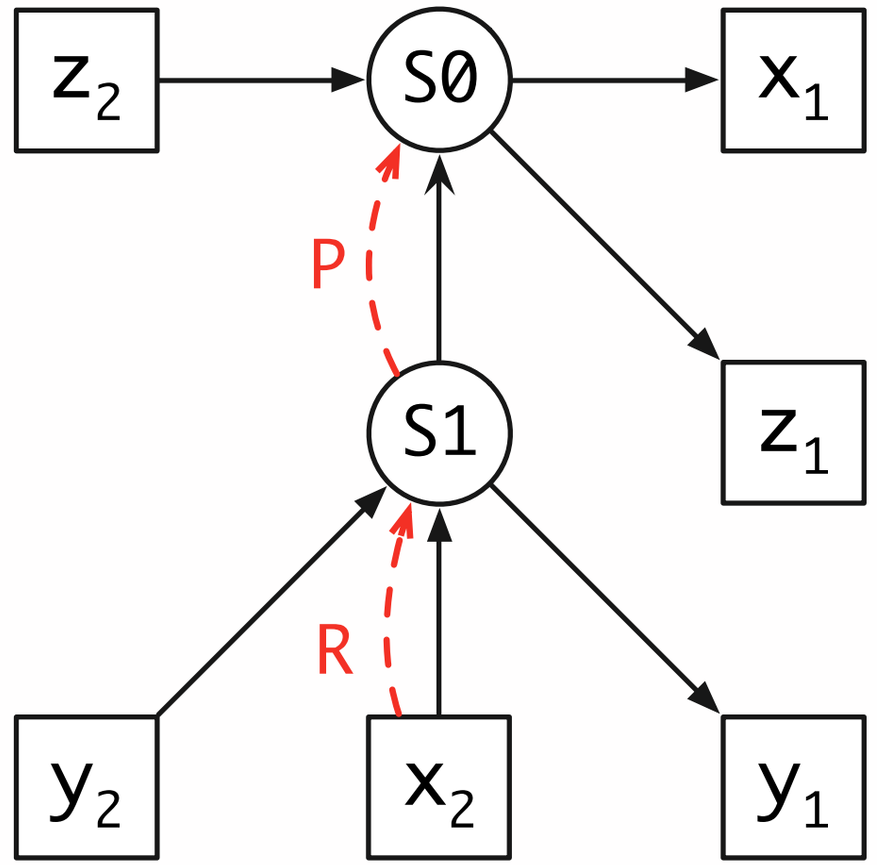


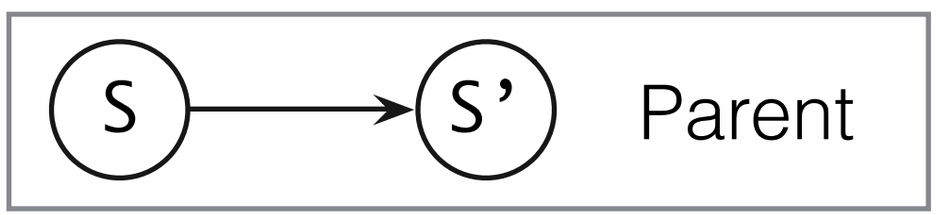


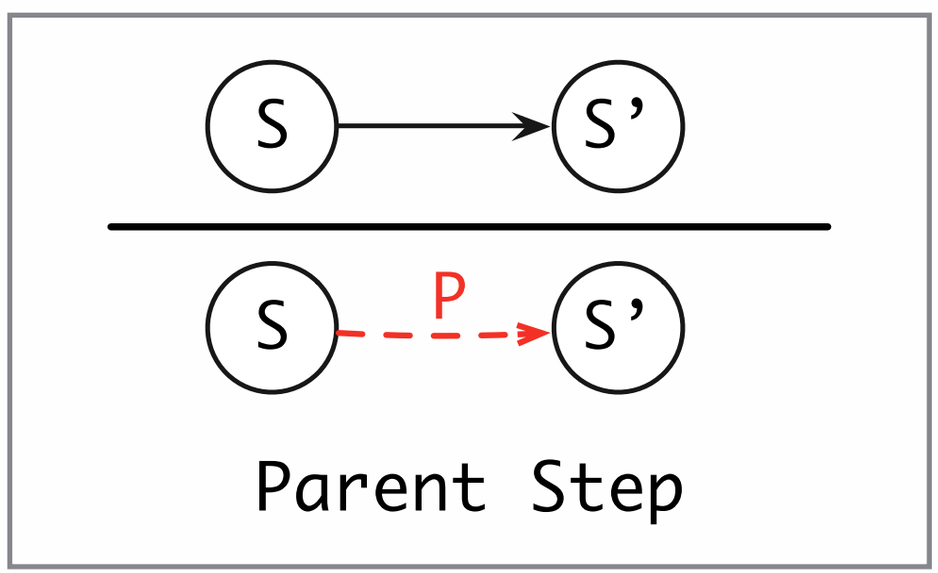


Lexical Scoping

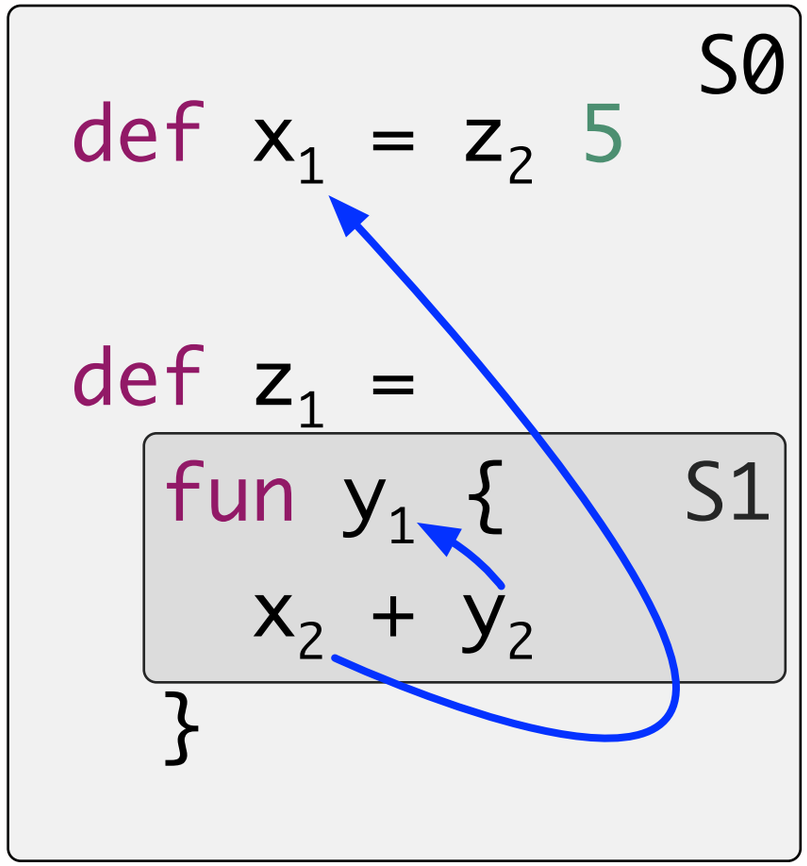


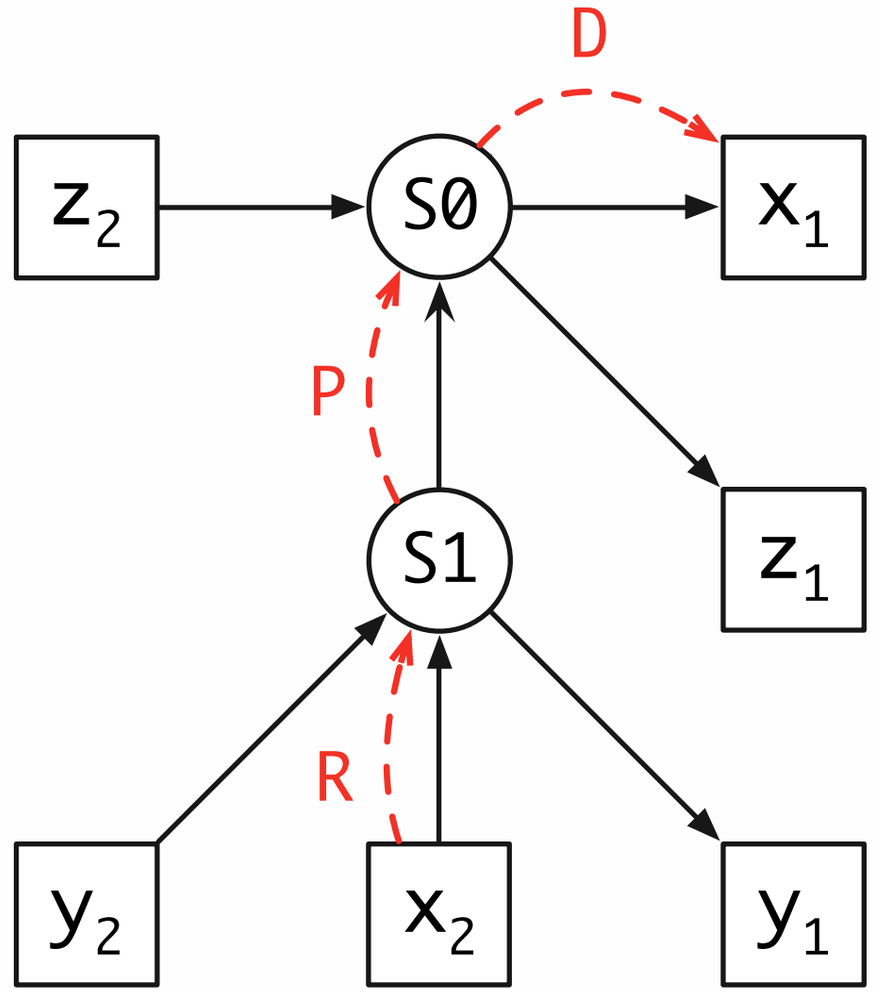


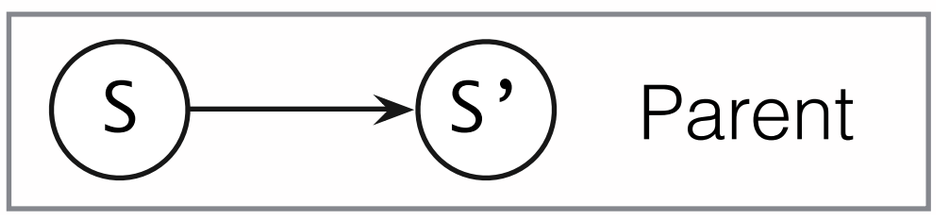


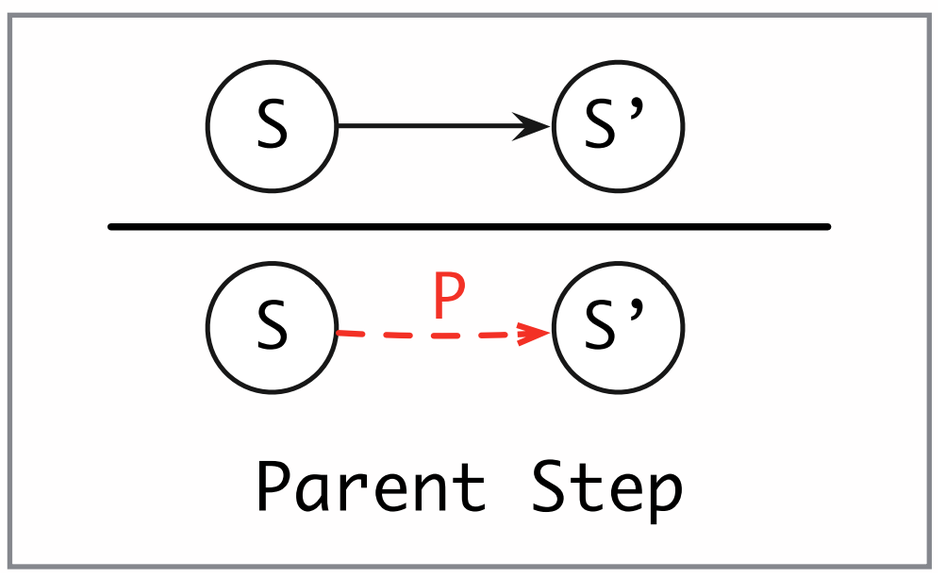


Lexical Scoping

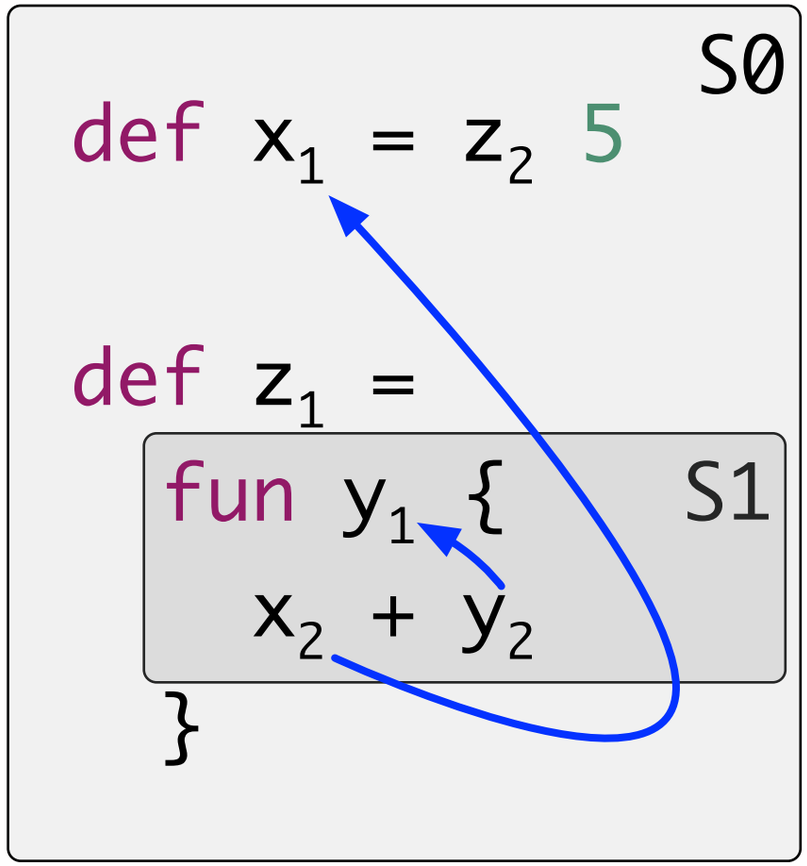


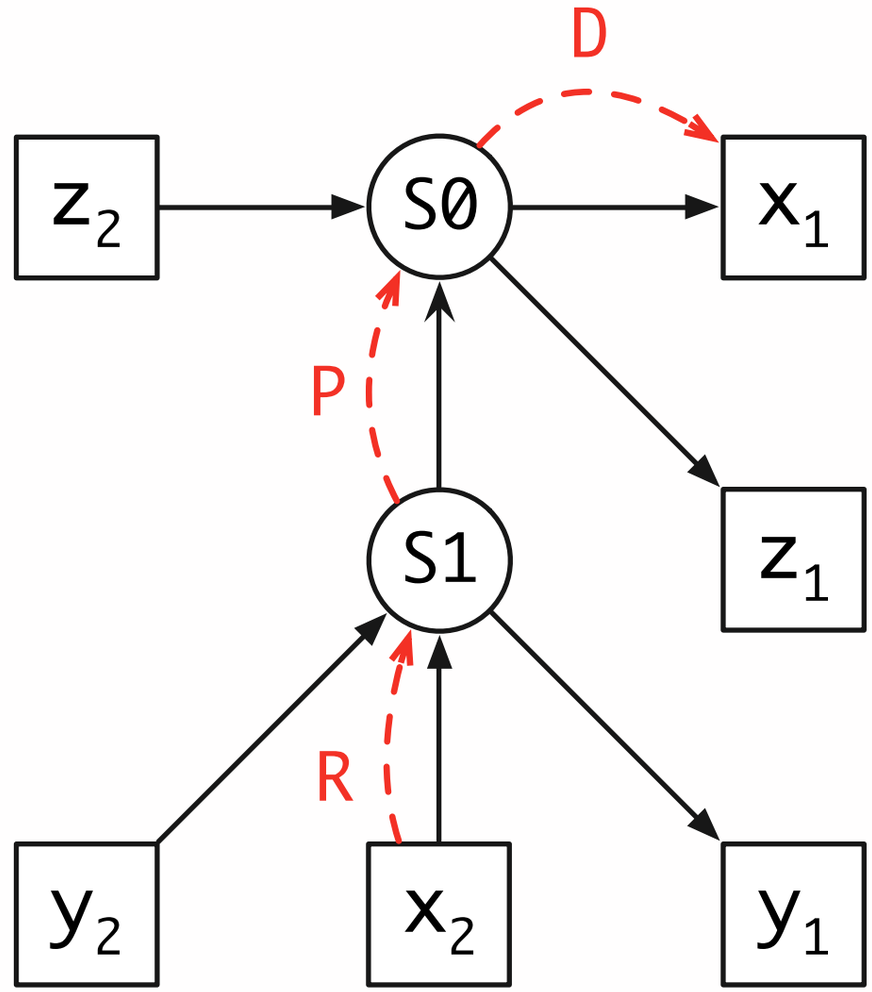


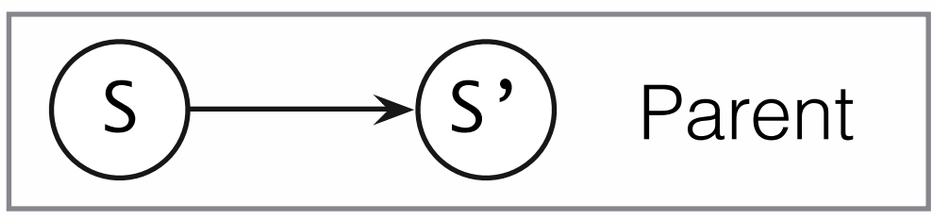




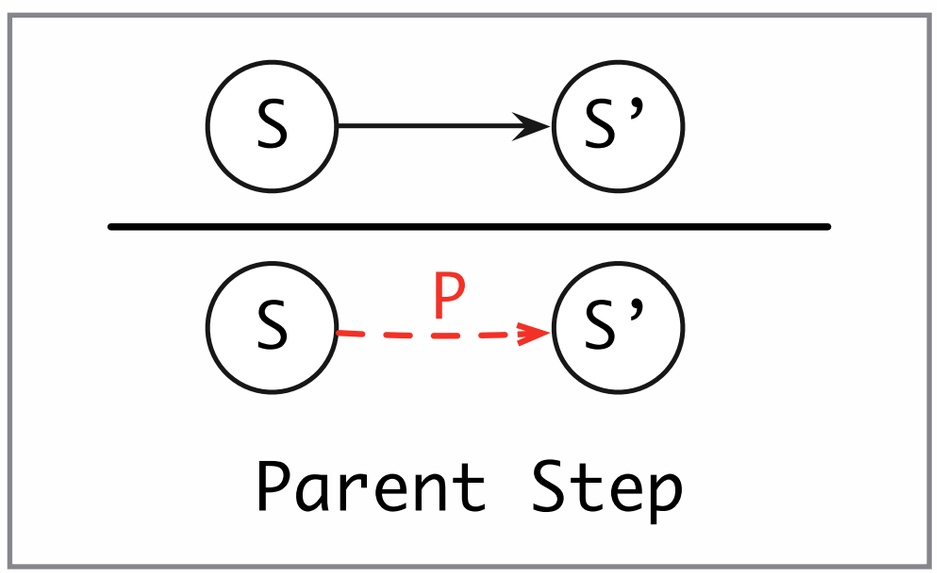
Lexical Scoping







Well formed path: R.P*.D



Shadowing

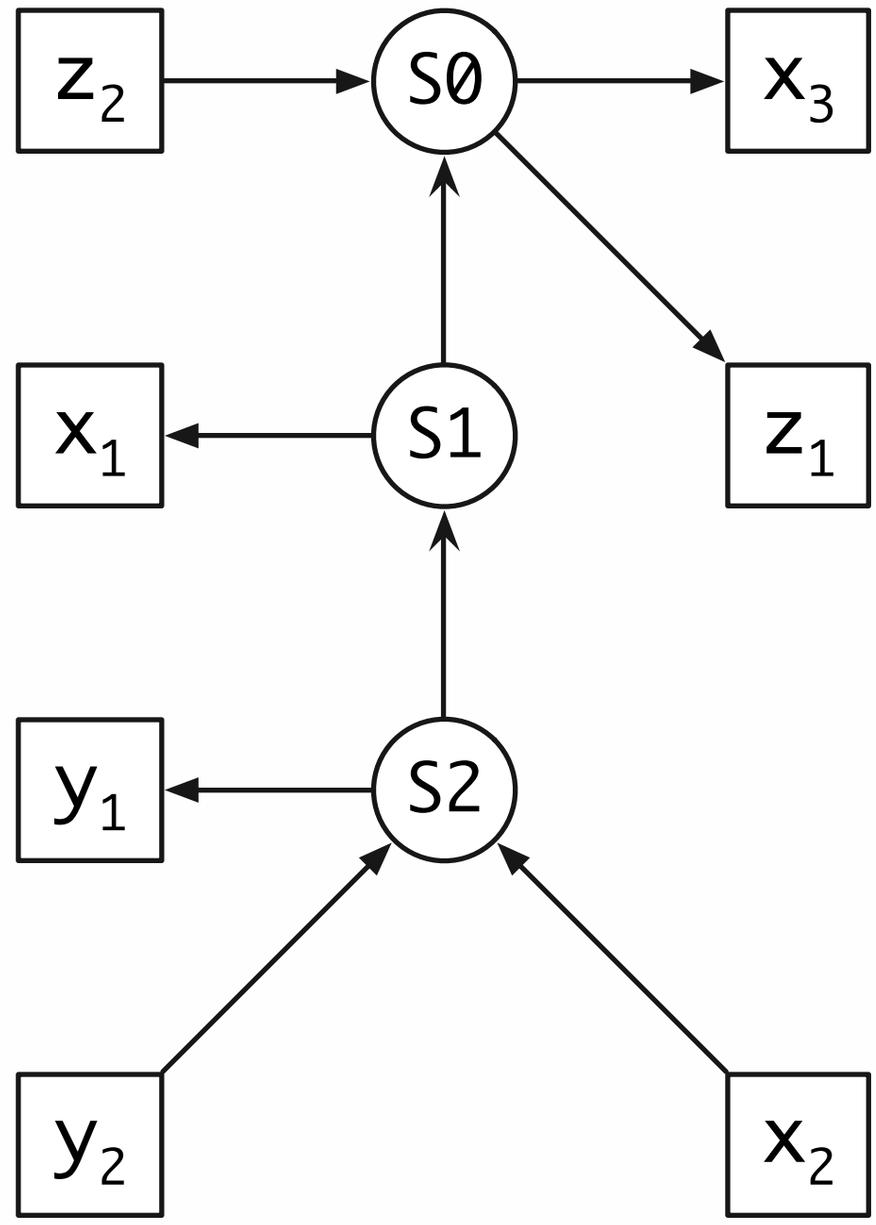
```
def x3 = z2 5 7

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

Shadowing

```
def x3 = z2 5 7                    S0


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

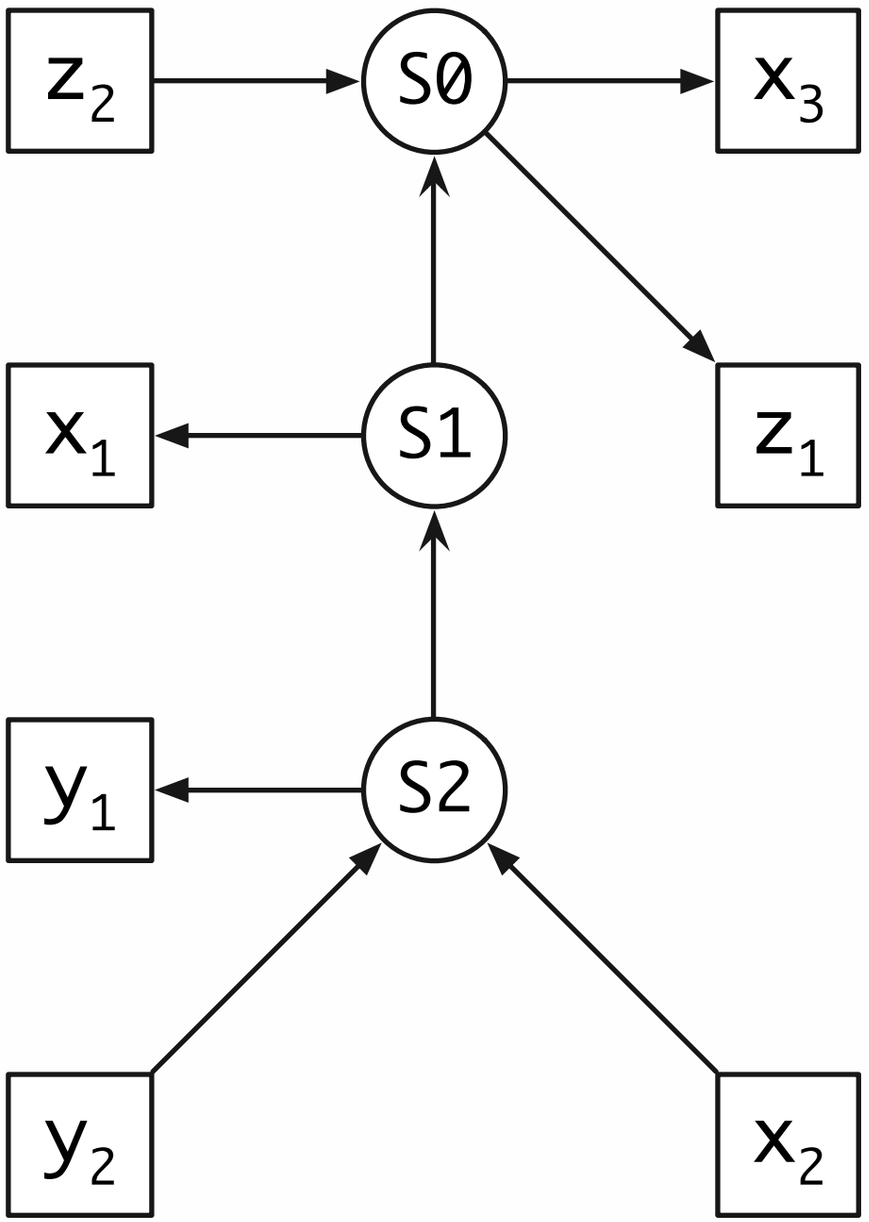
Shadowing

```
def x3 = z2 5 7                    S0

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



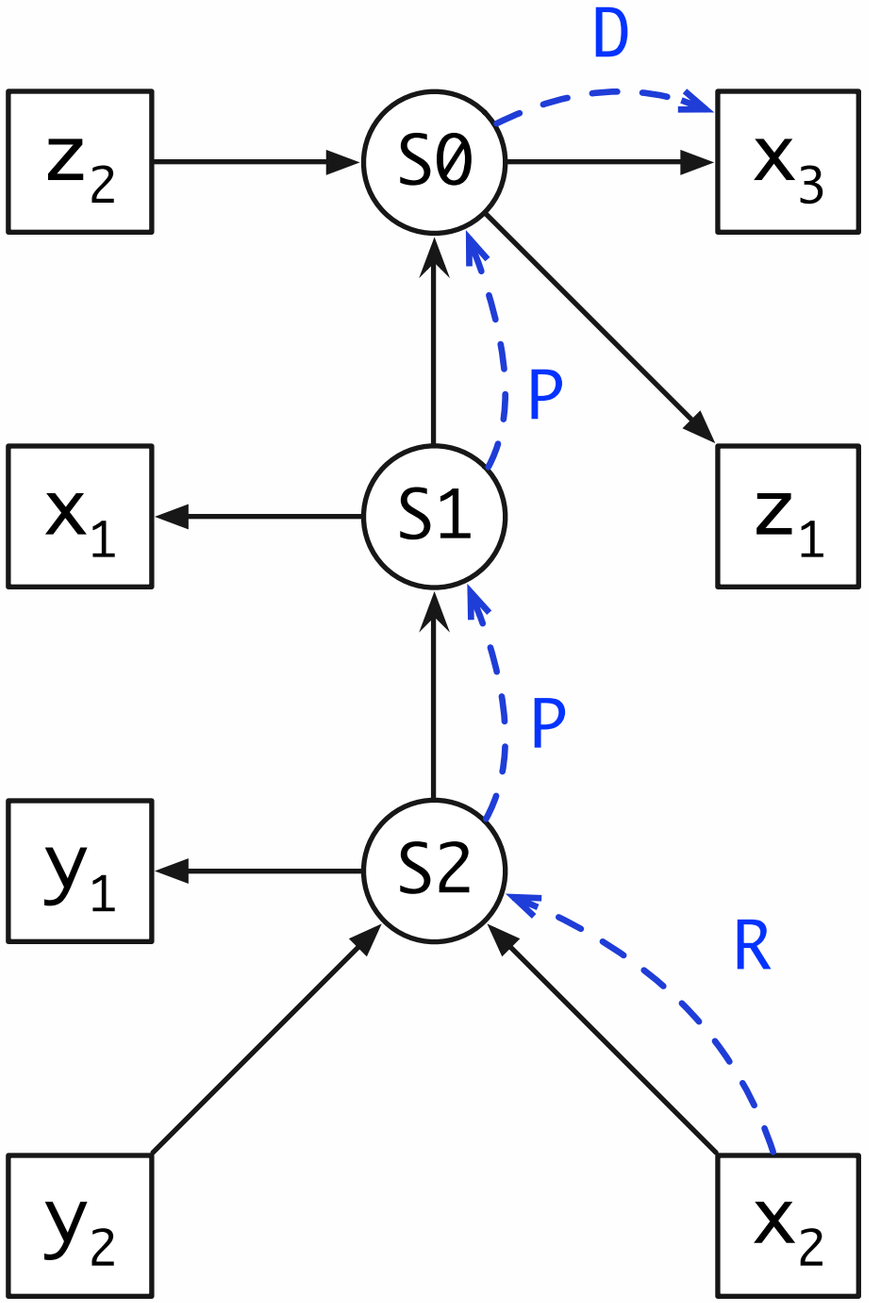
Shadowing

```
def x3 = z2 5 7                    S0

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



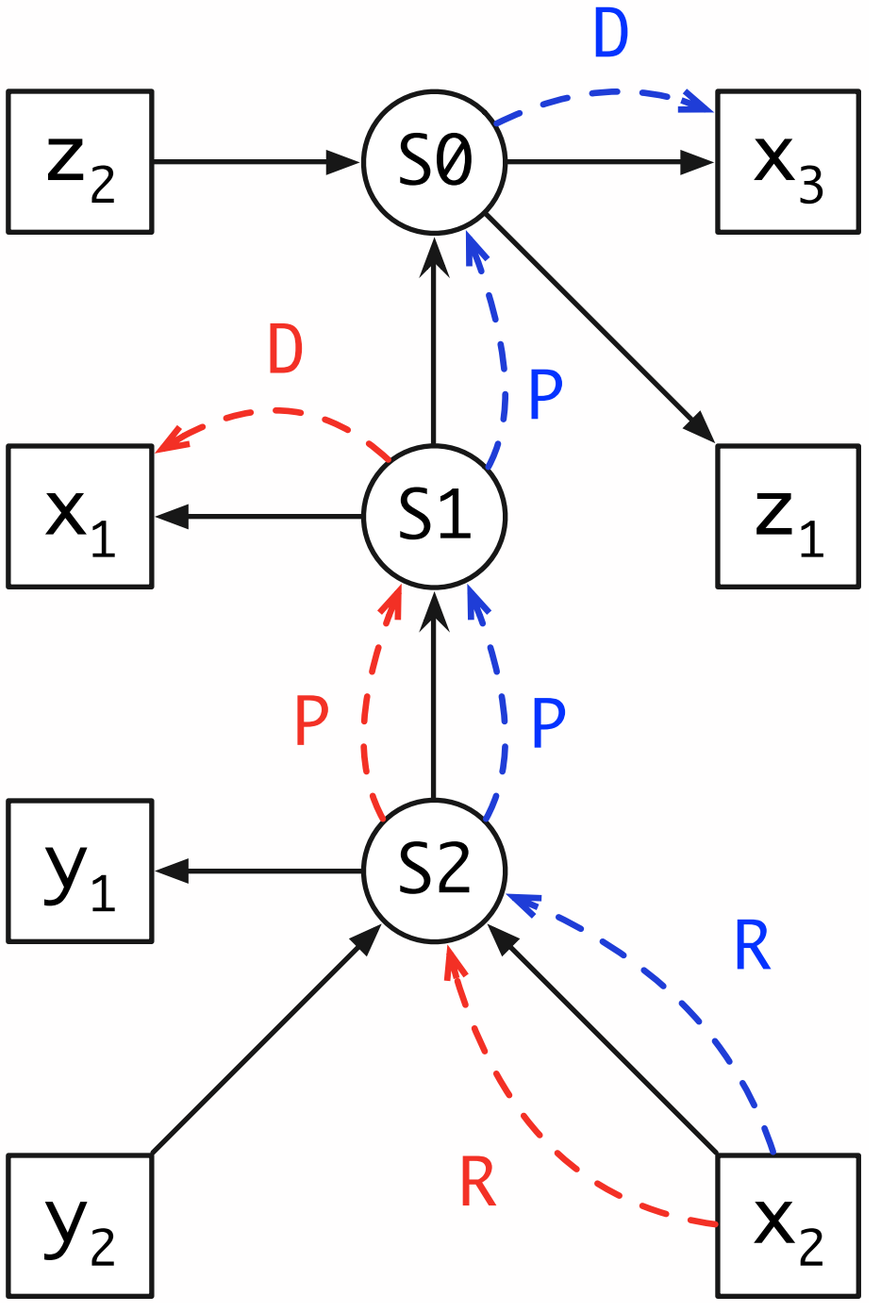
Shadowing

```
def x3 = z2 5 7                    S0

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



Shadowing

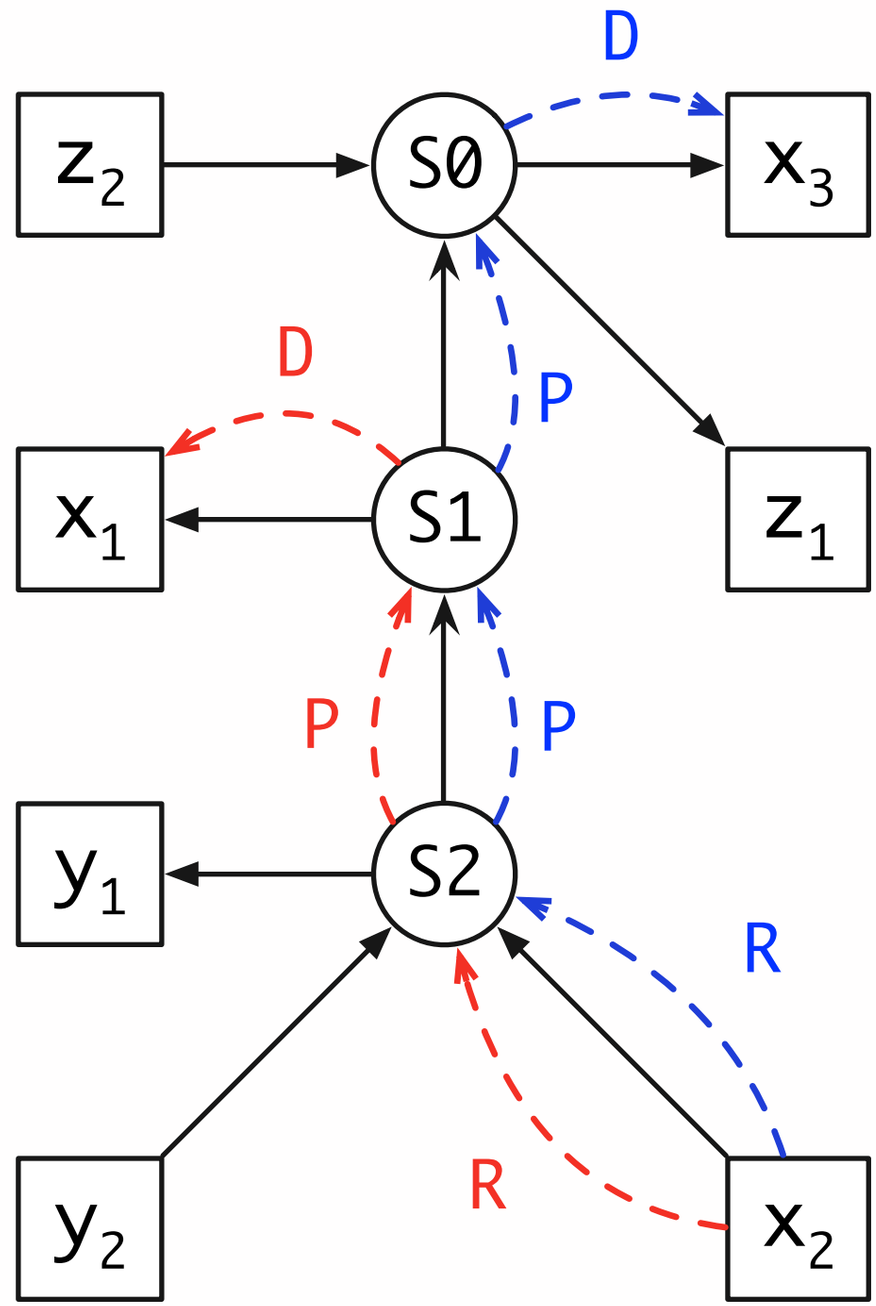
```
def x3 = z2 5 7                    S0

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



Shadowing

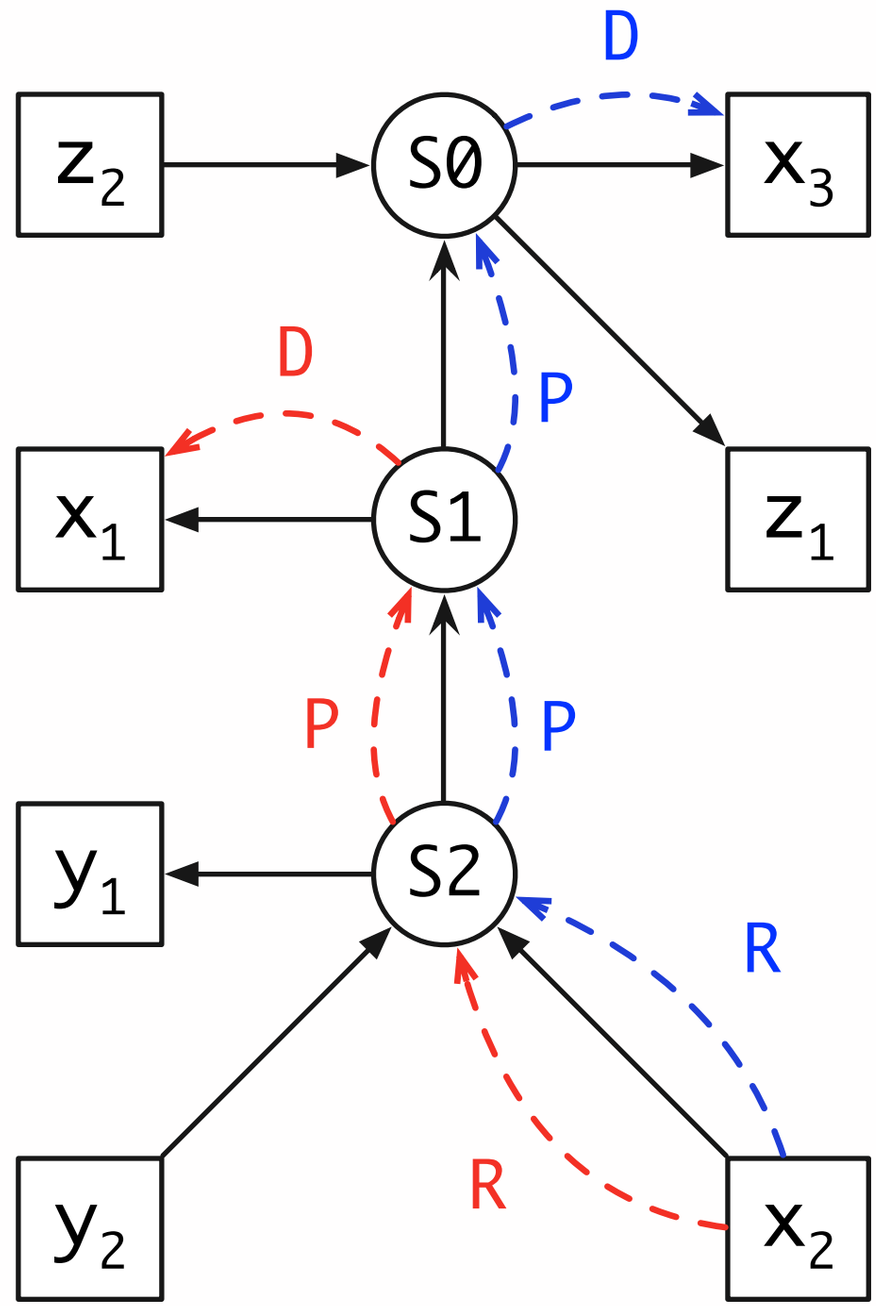
```
def x3 = z2 5 7                    S0

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

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



Shadowing

```
def x3 = z2 5 7                    S0

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

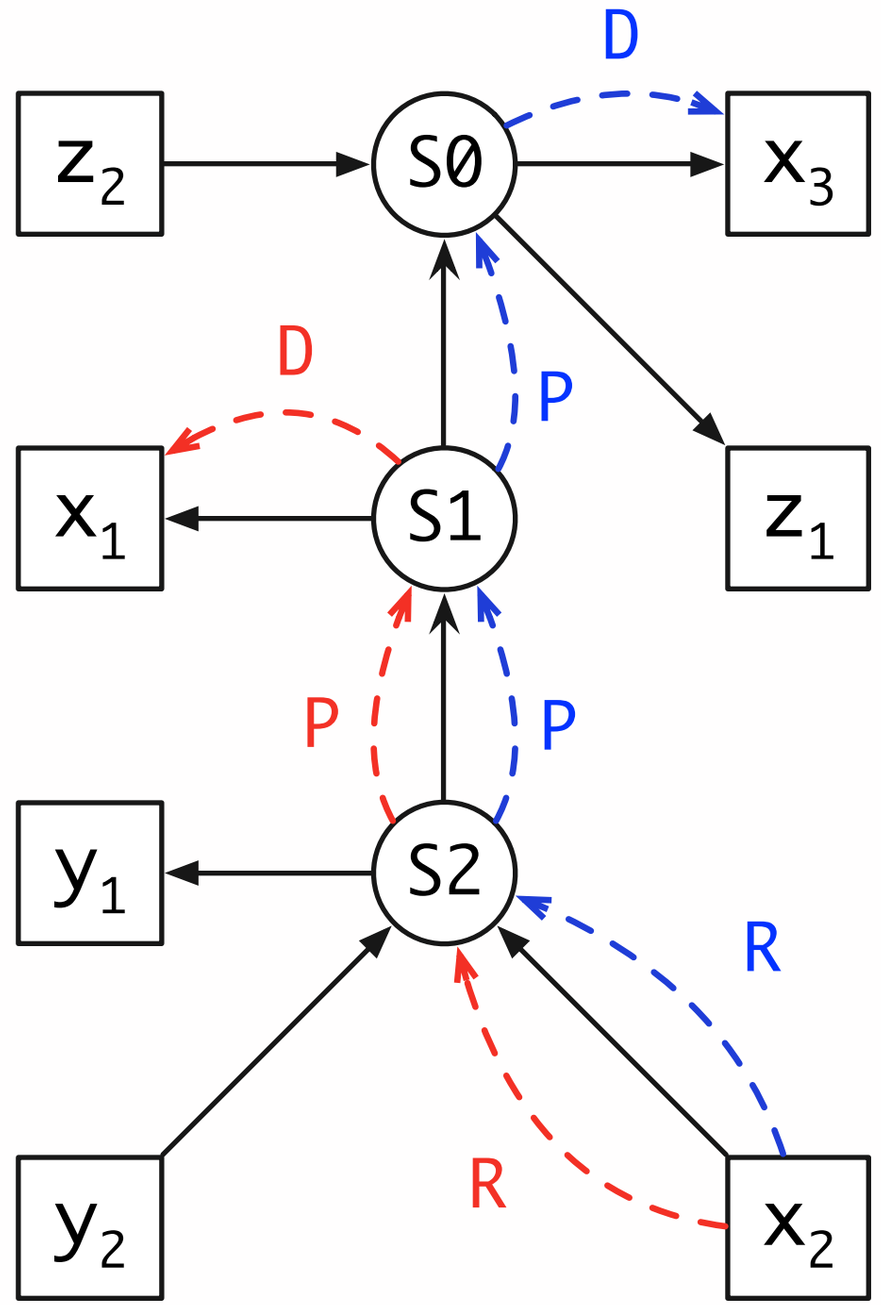


$$\frac{}{D < P.p}$$

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

Shadowing

```
def x3 = z2 5 7                                   S0

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



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

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

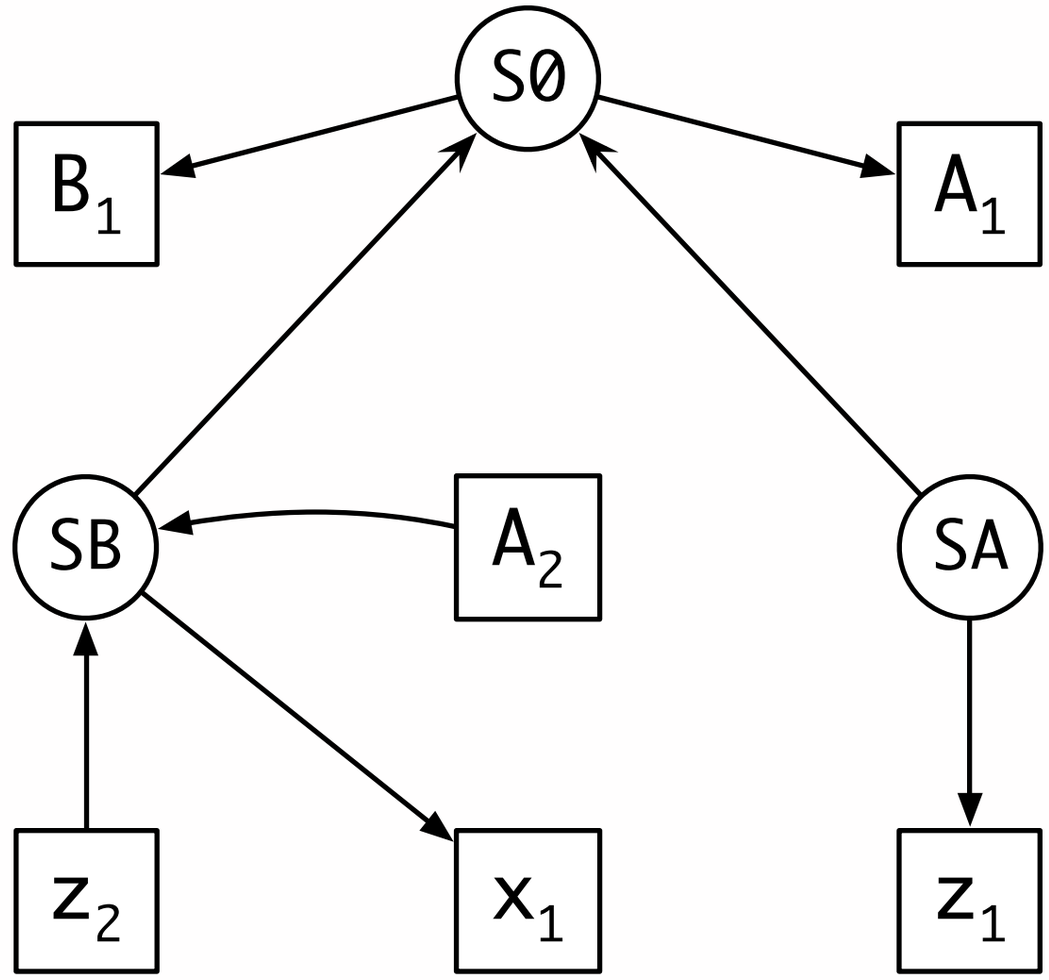
$$R.P.D < R.P.P.D$$

Imports

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

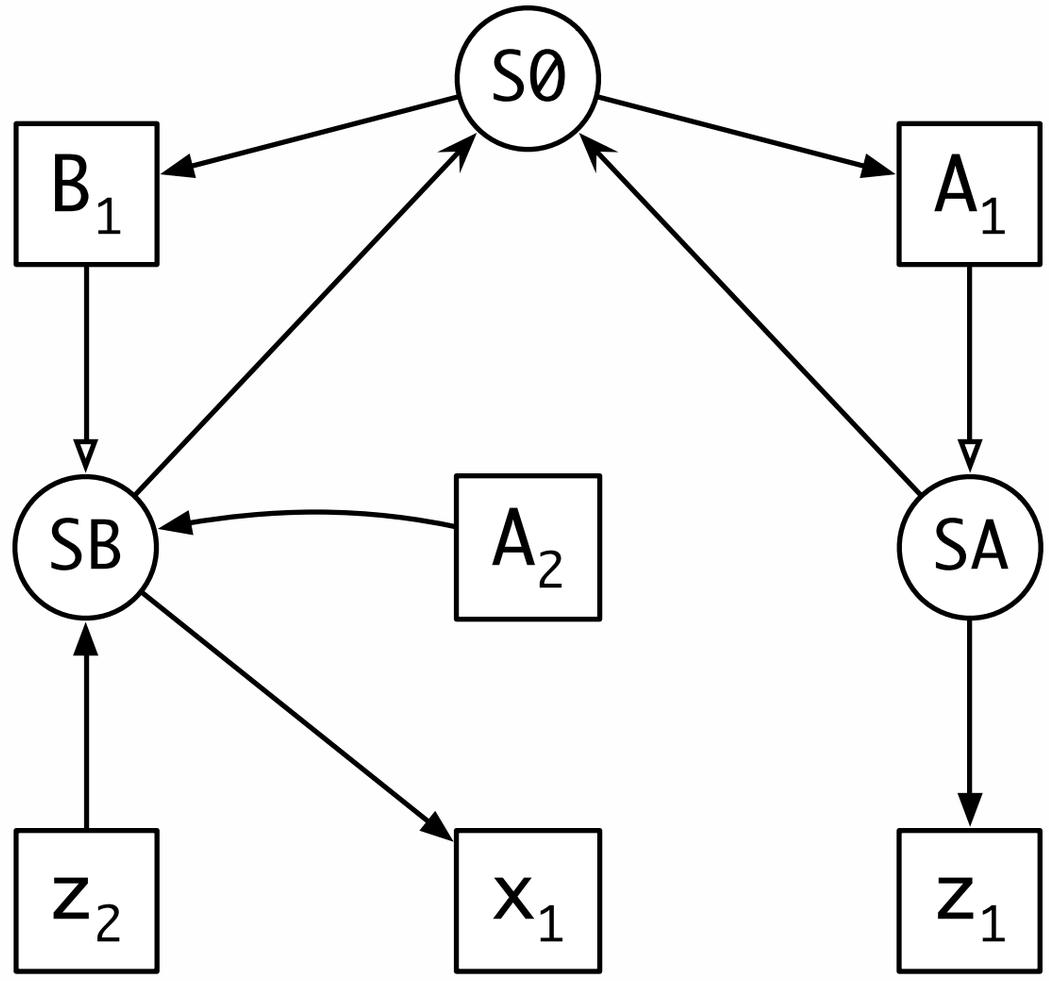
Imports

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



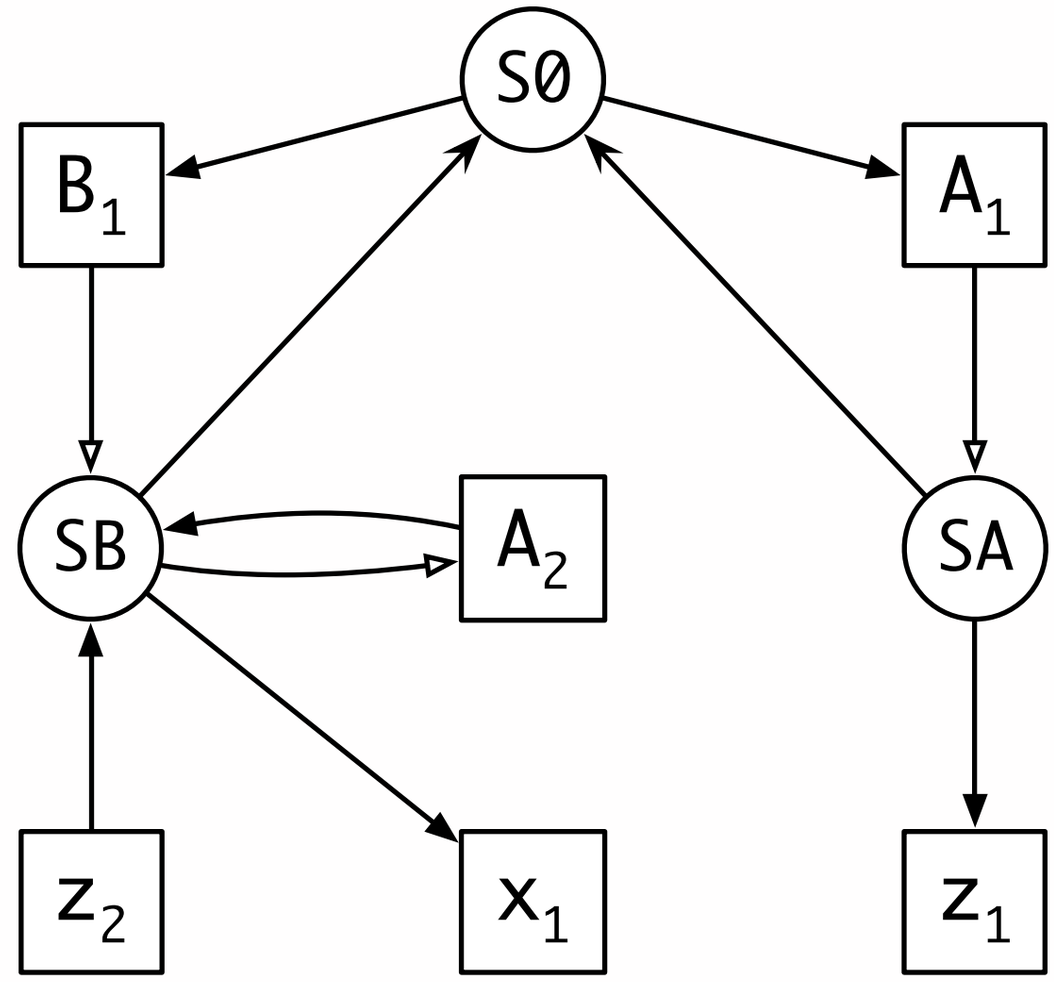
Imports

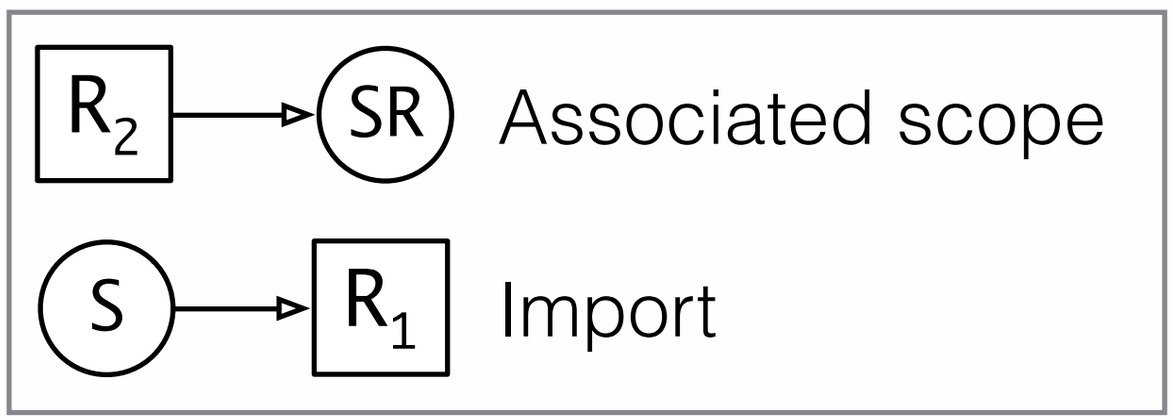
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```



Imports

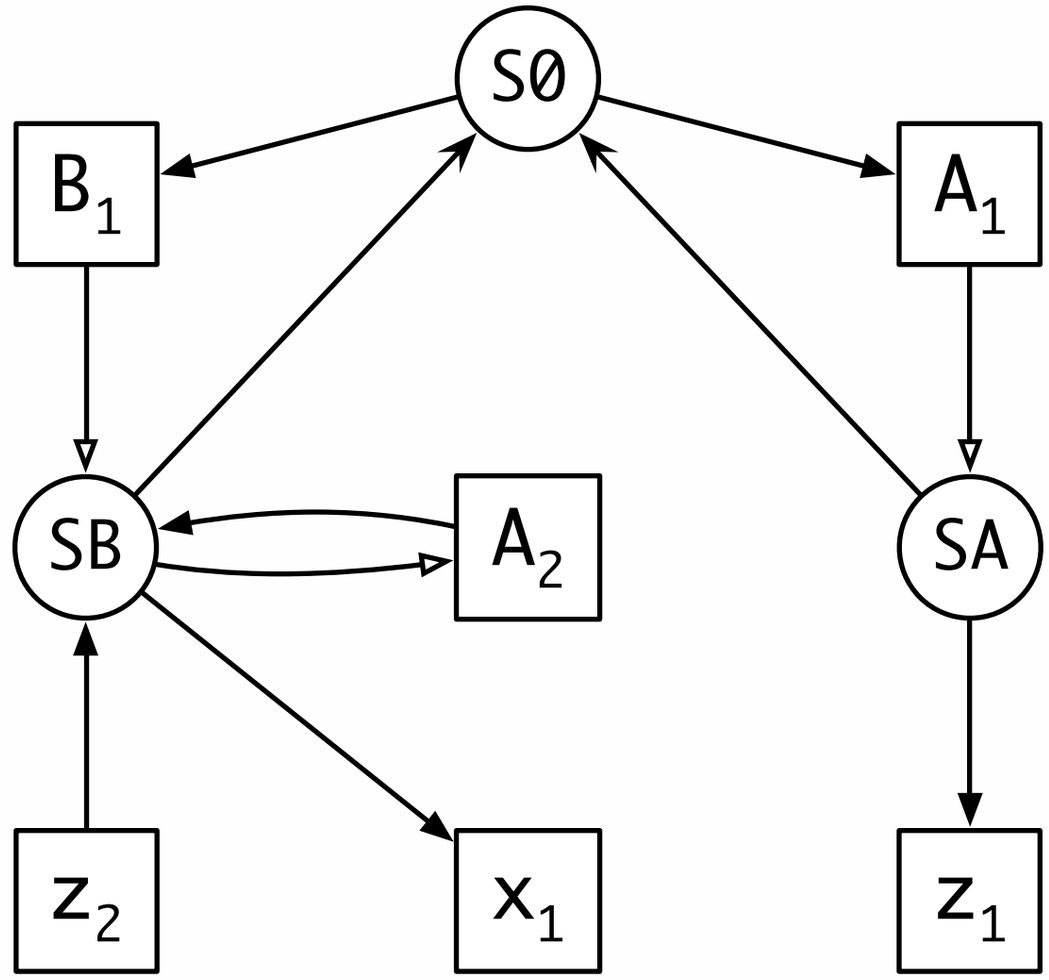
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

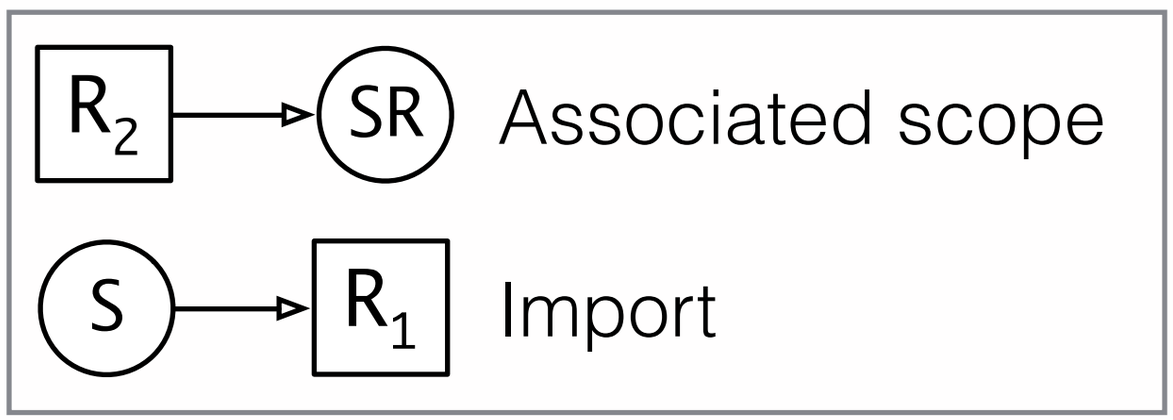




Imports

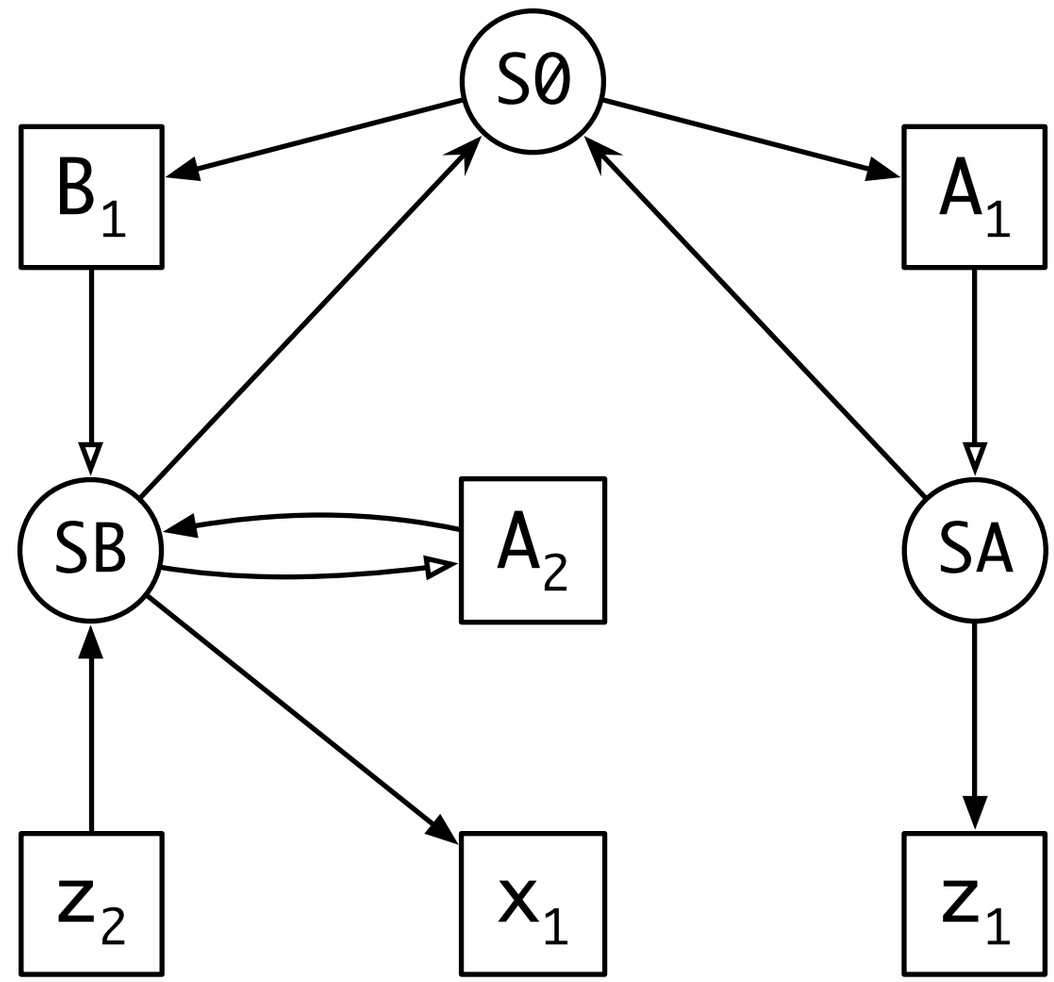
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

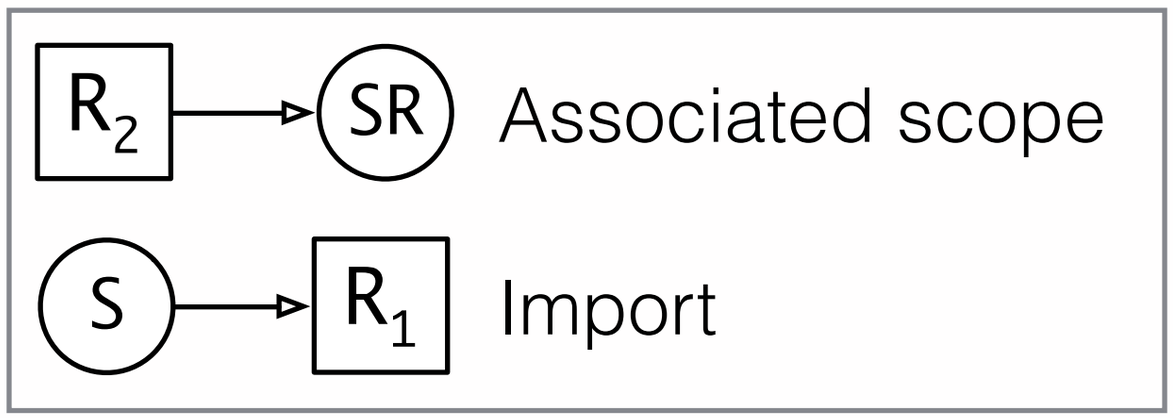




Imports

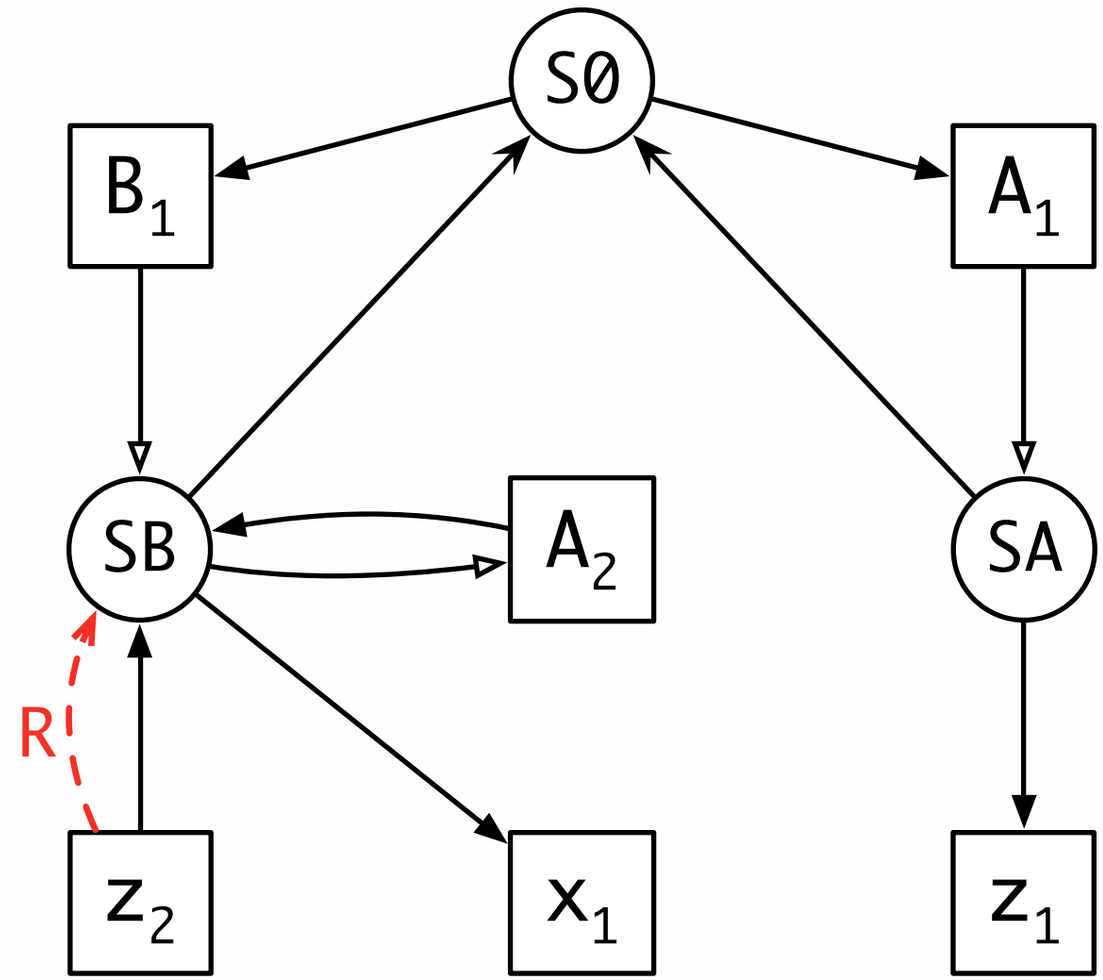
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

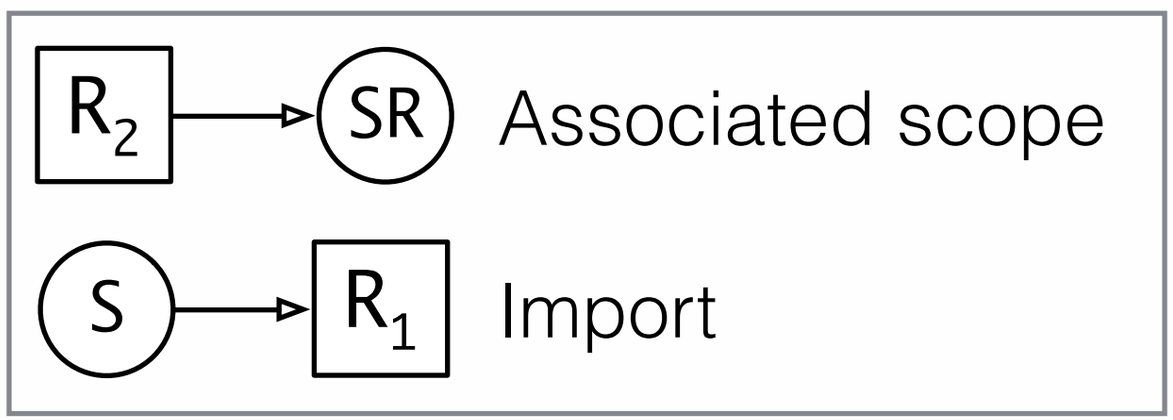




Imports

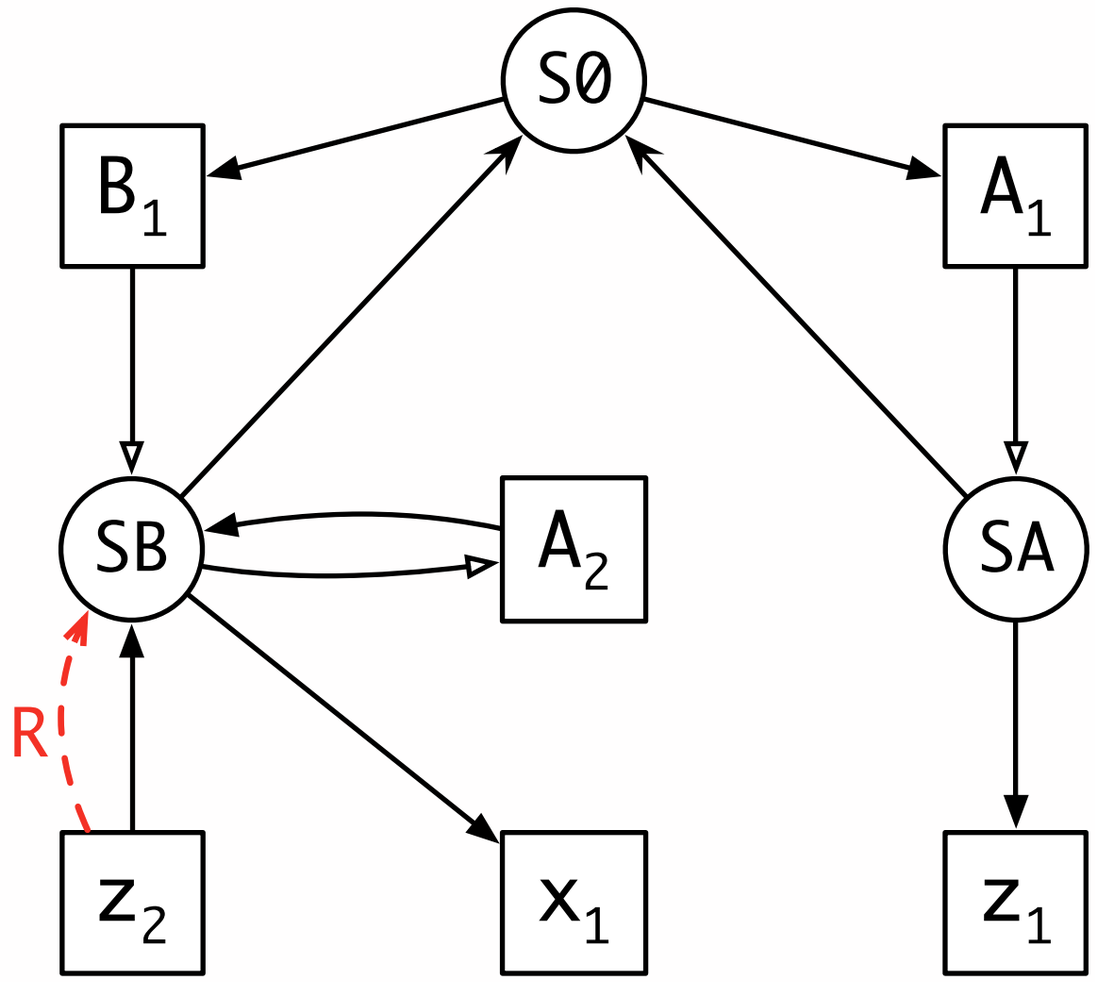
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

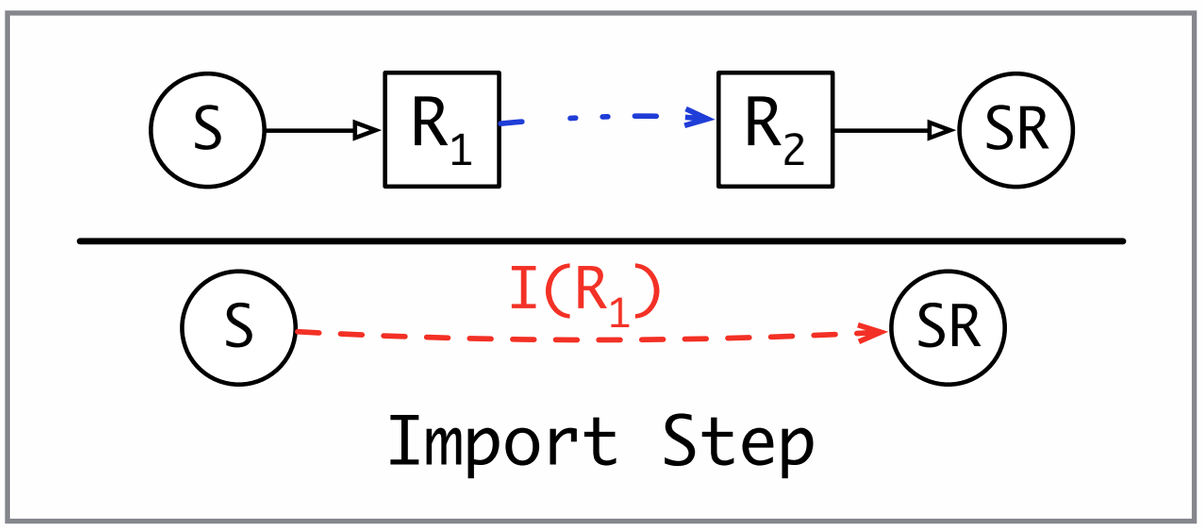




Imports

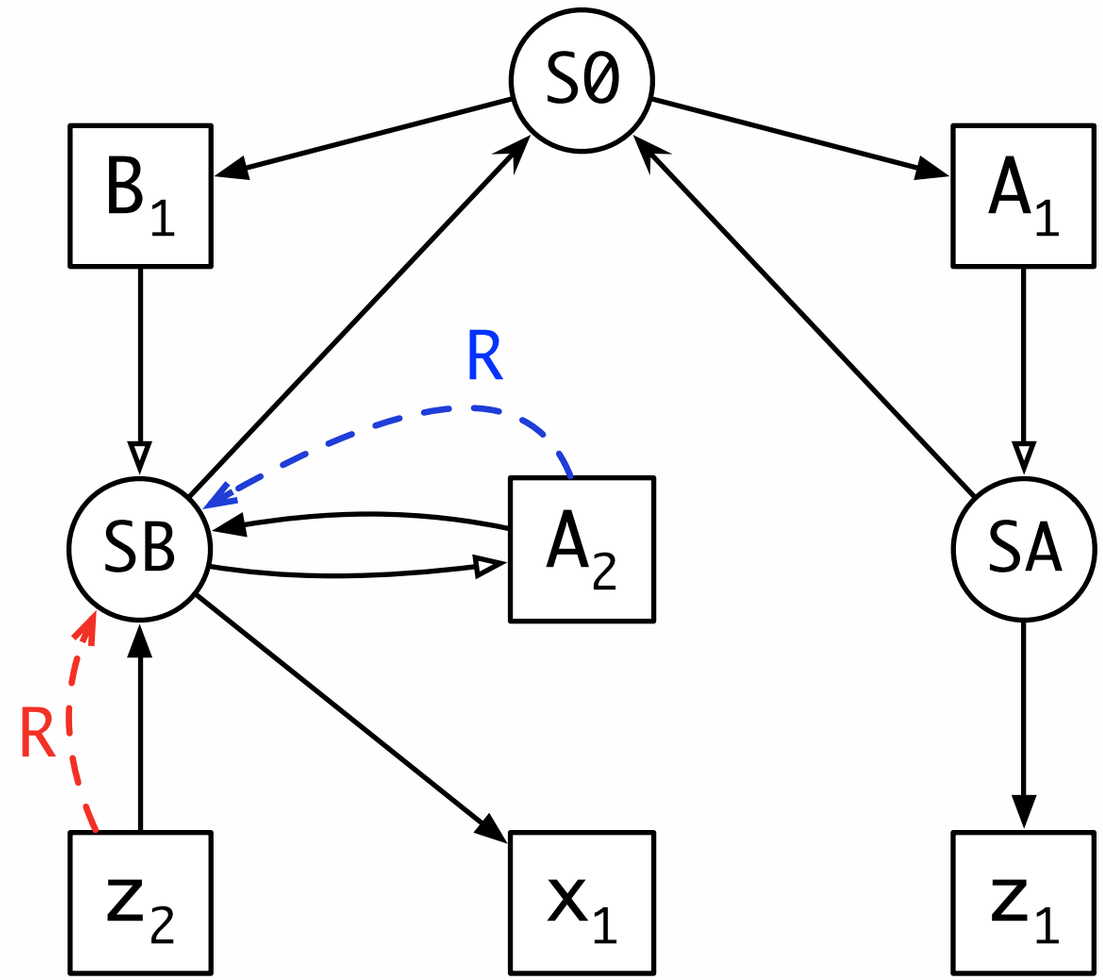
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

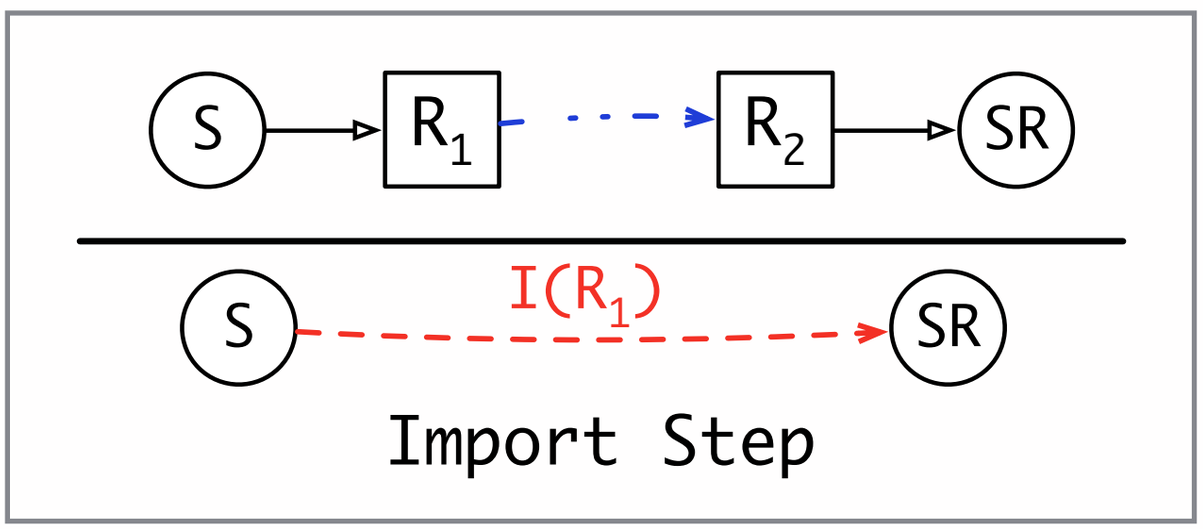




Imports

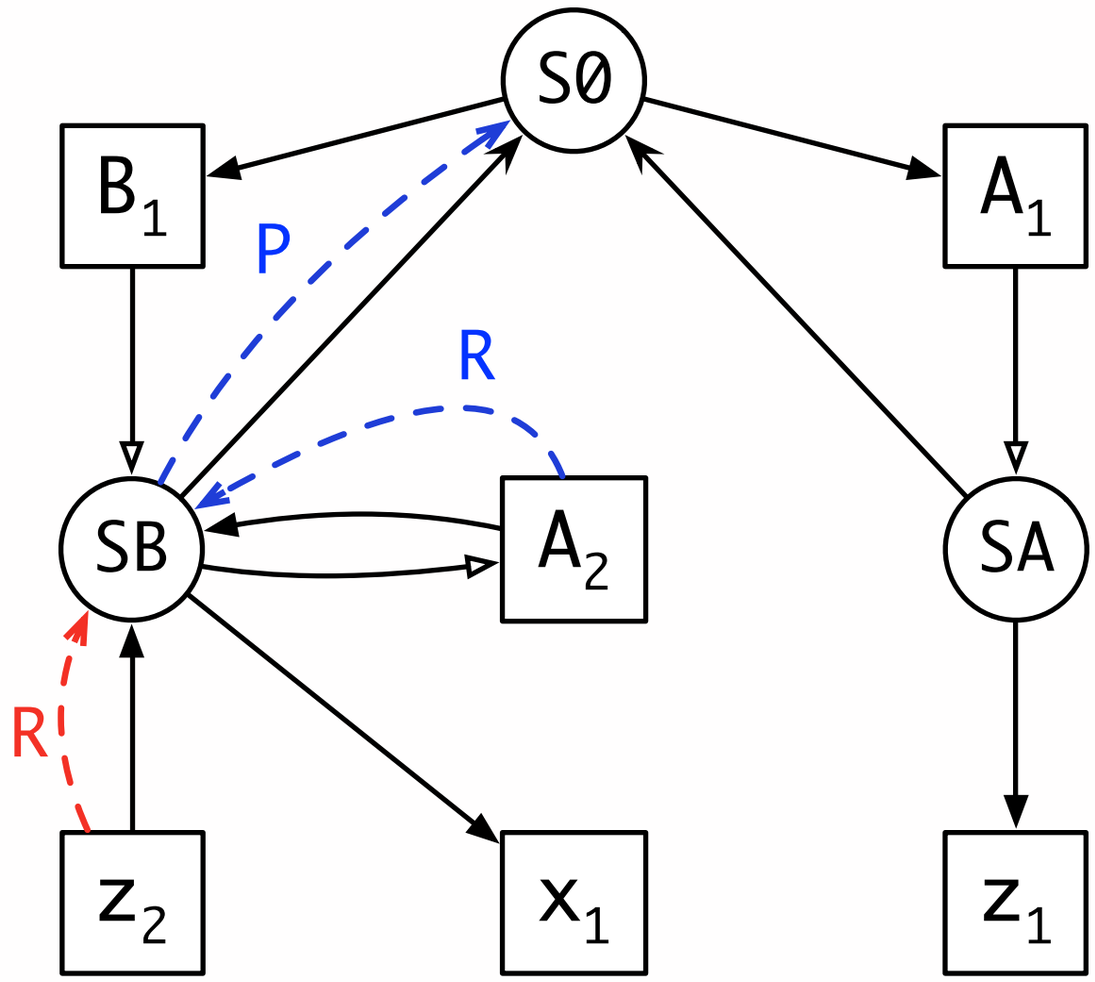
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

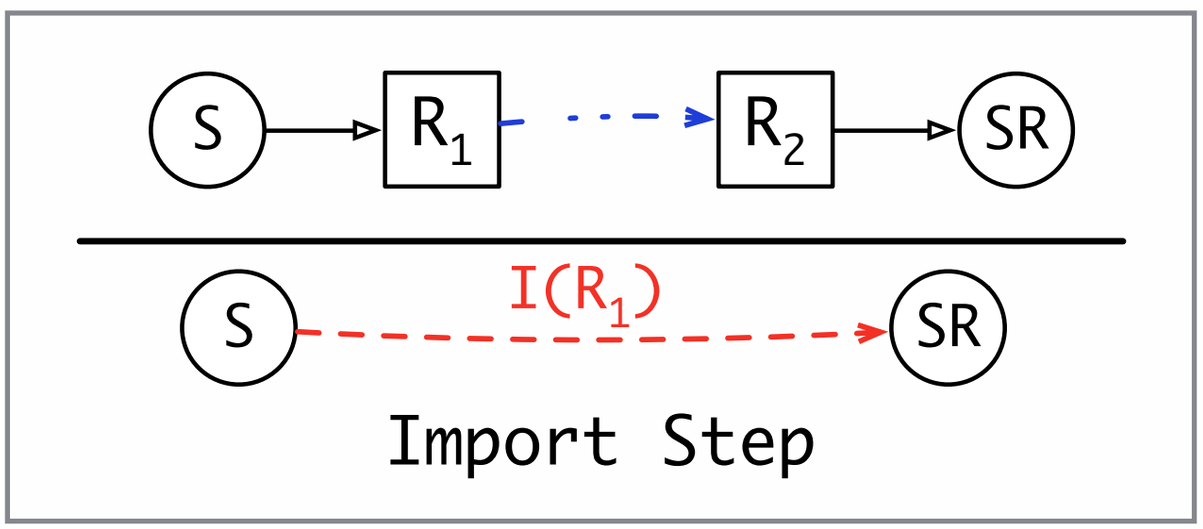




Imports

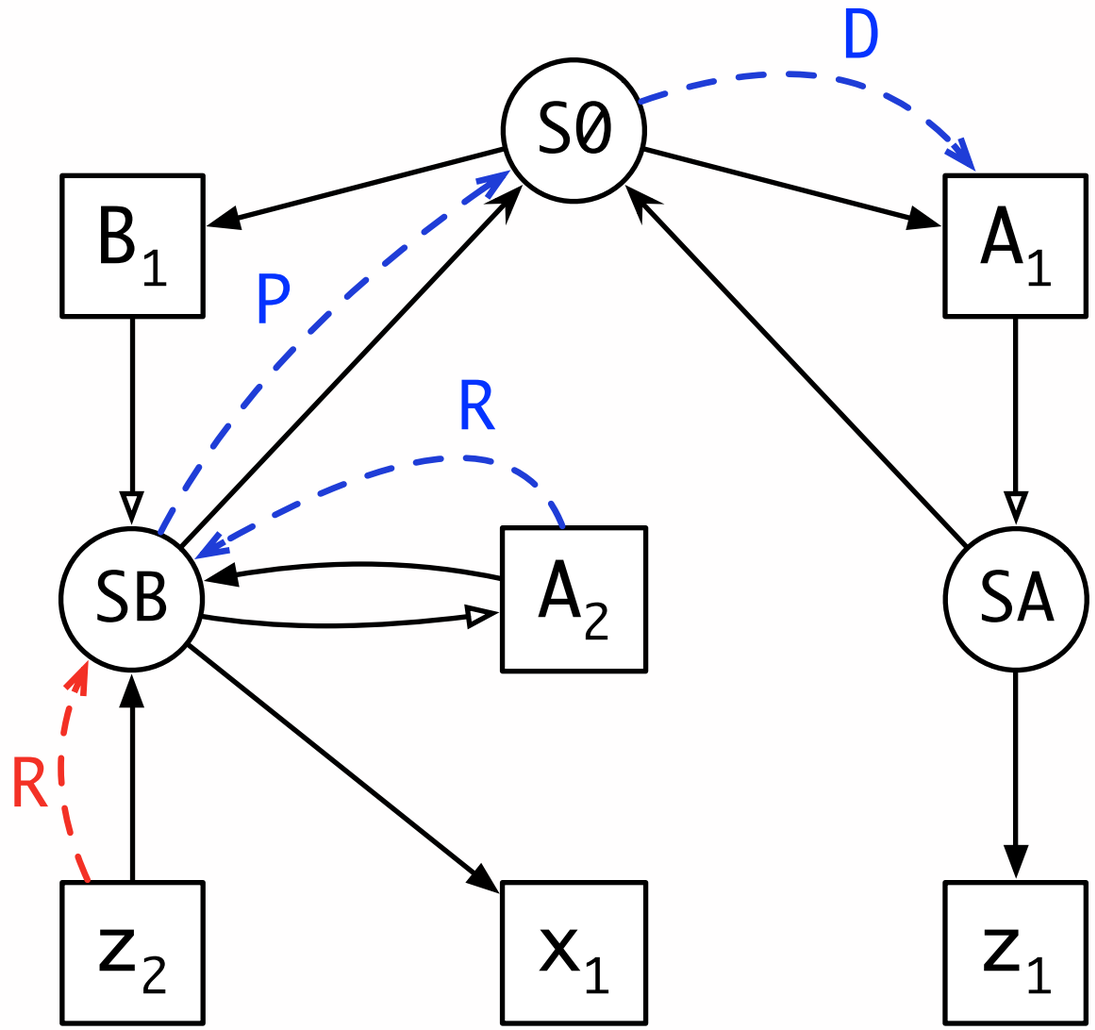
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

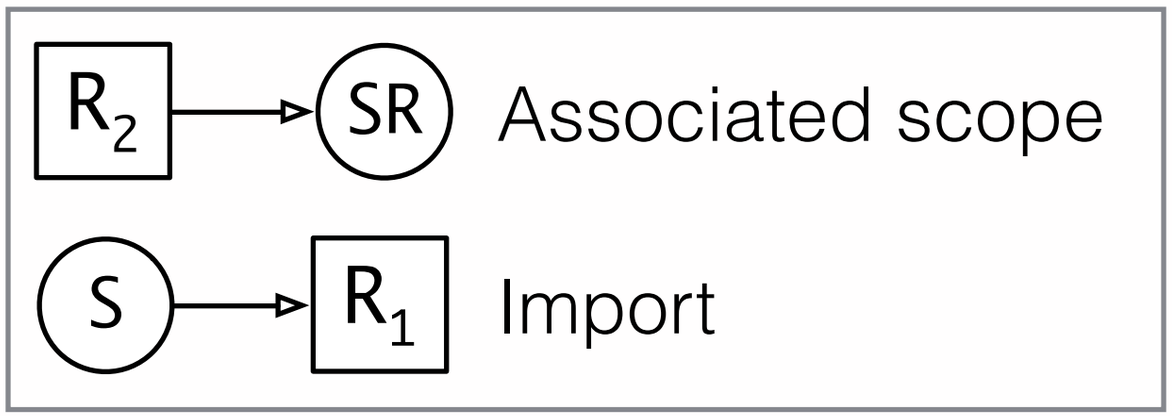


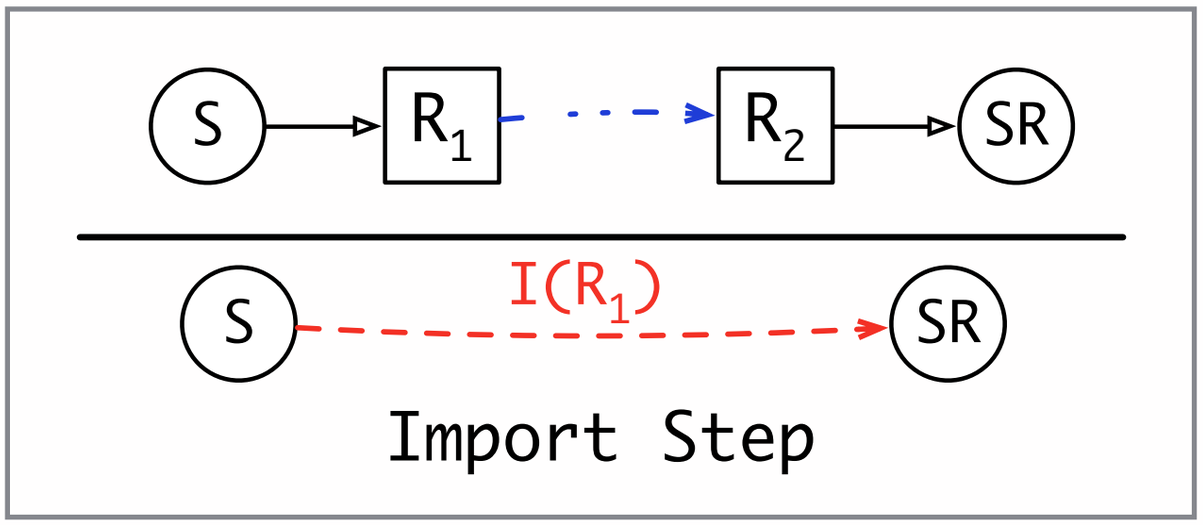


Imports

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

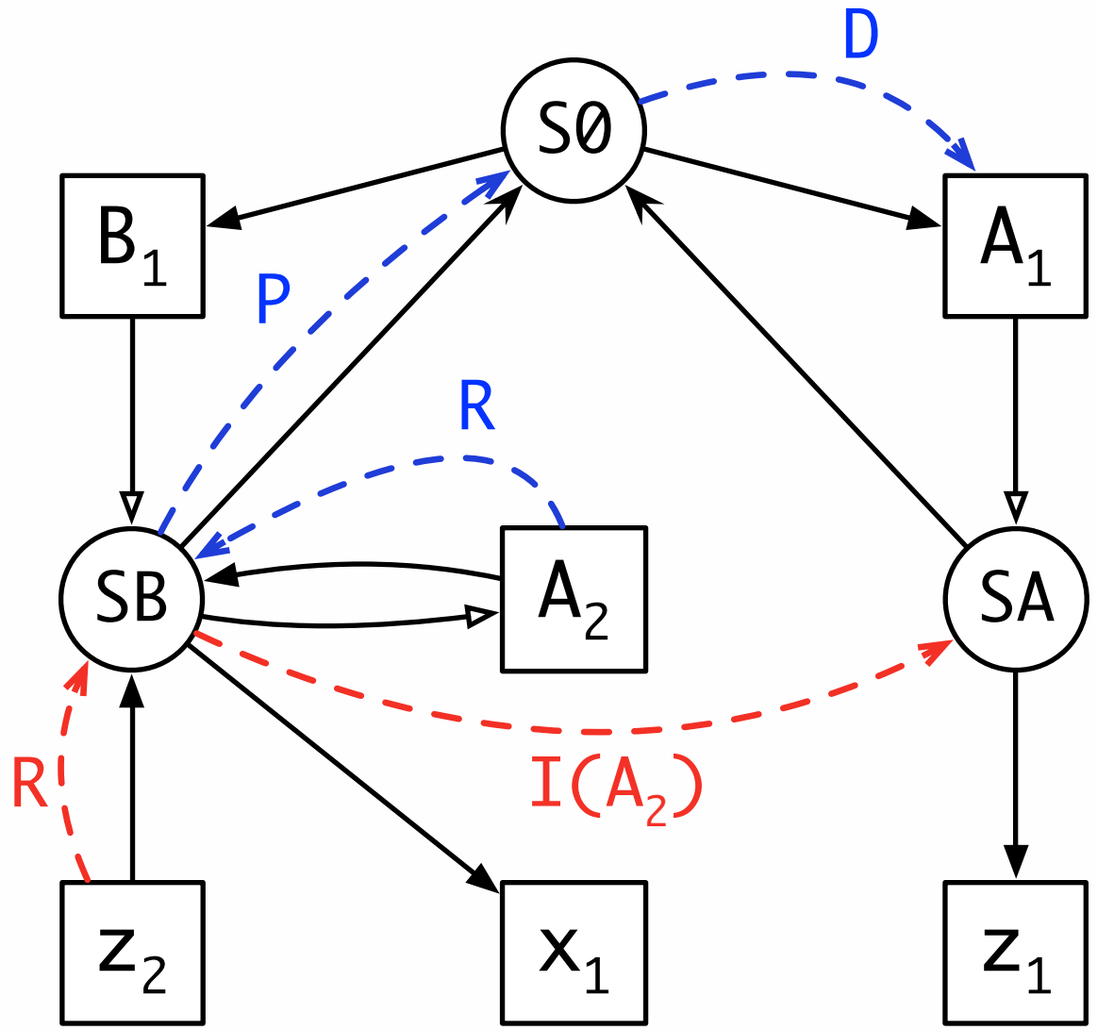


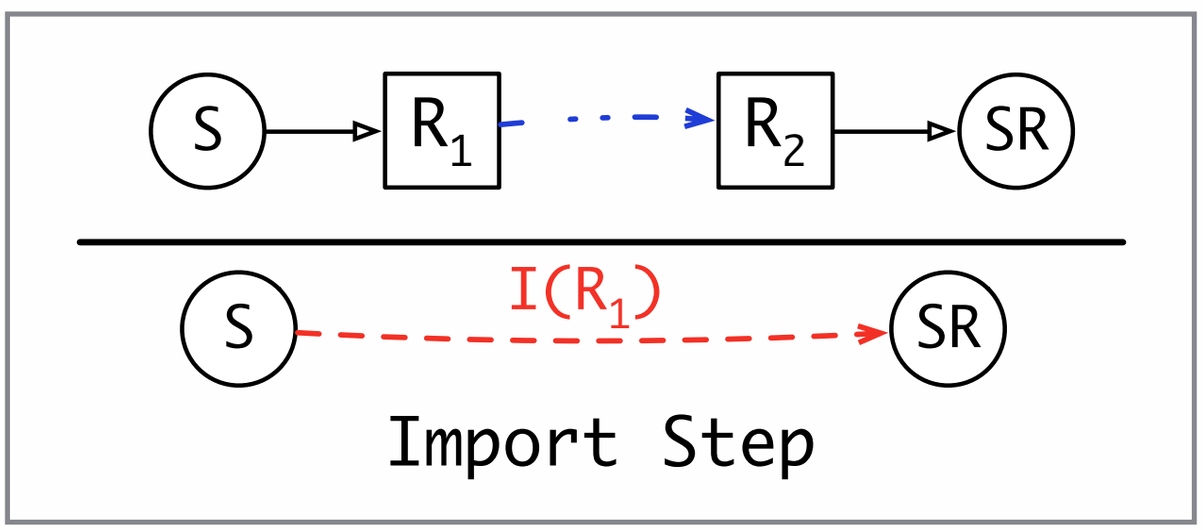




Imports

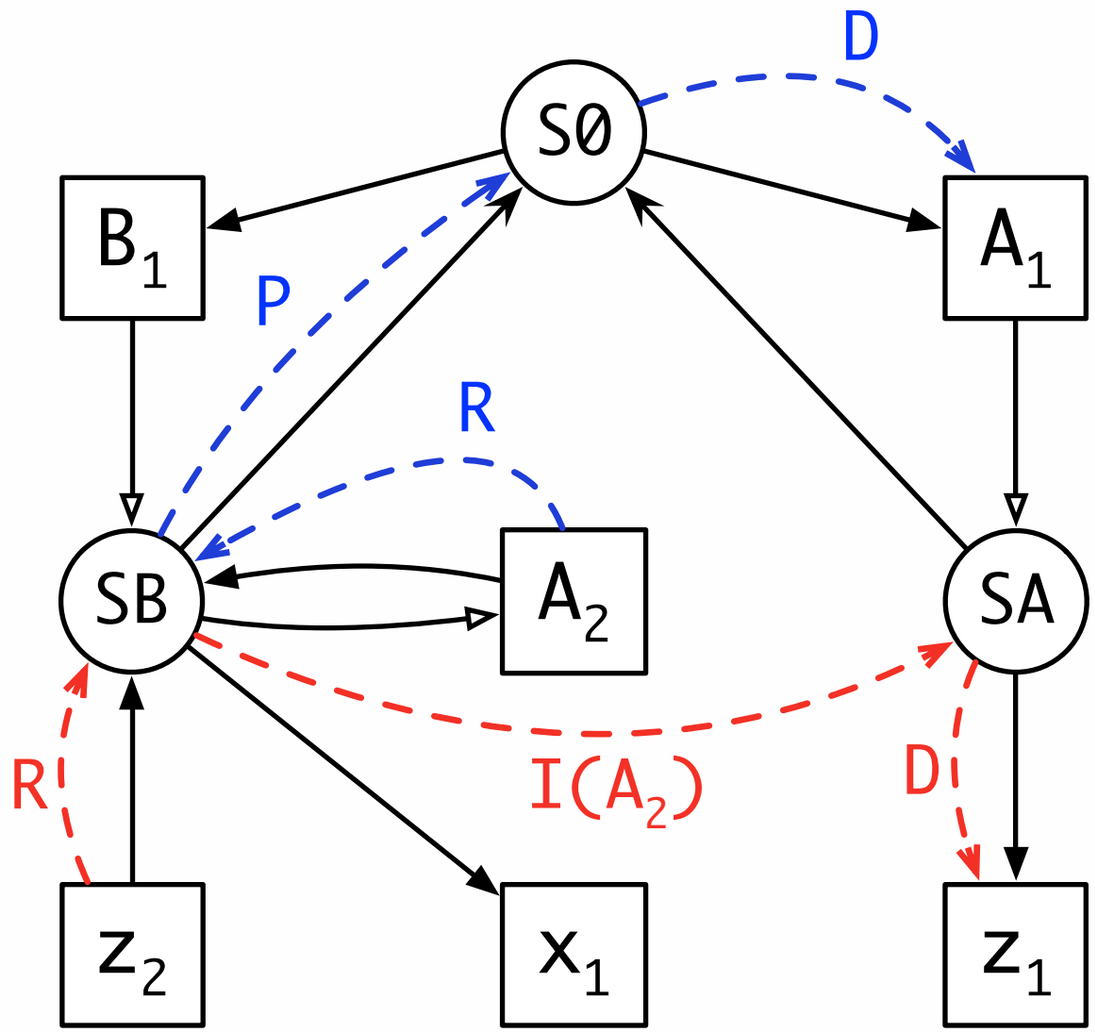
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

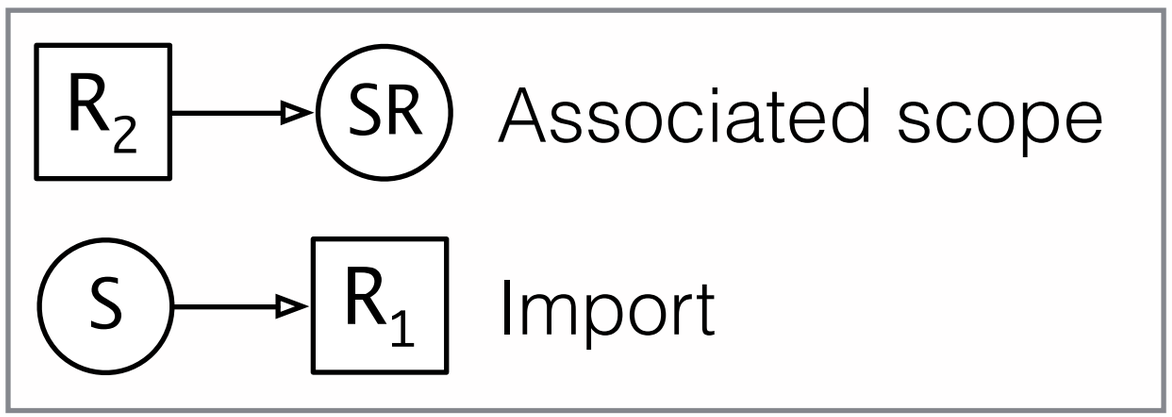


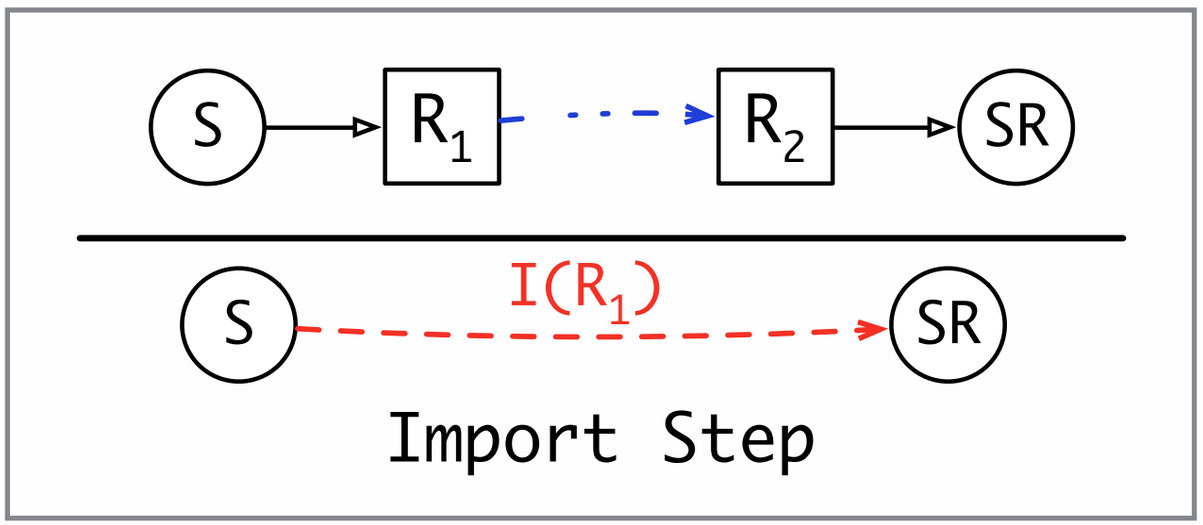


Imports

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





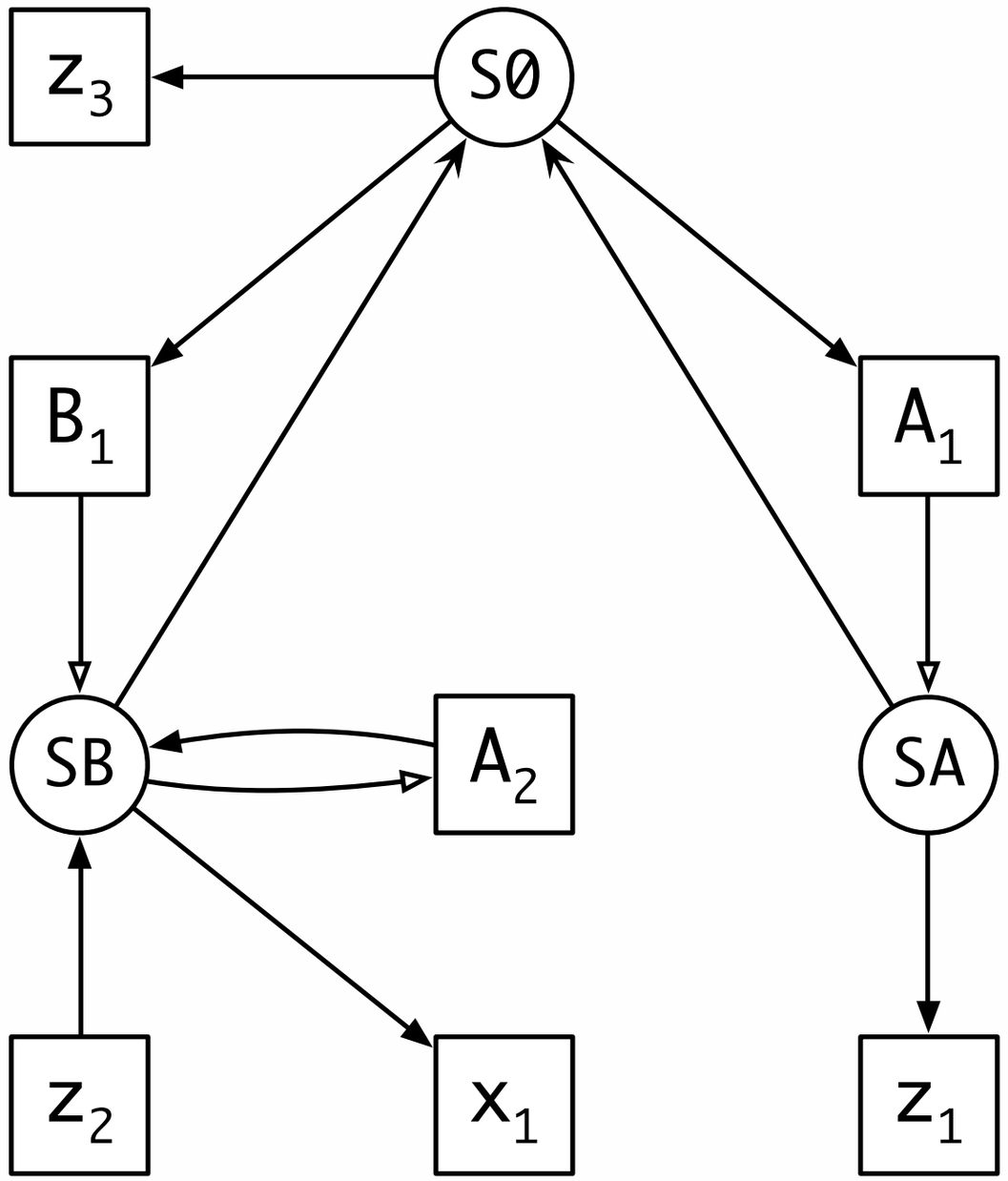


Imports shadow Parents

```
def z3 = 2                                  S0

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

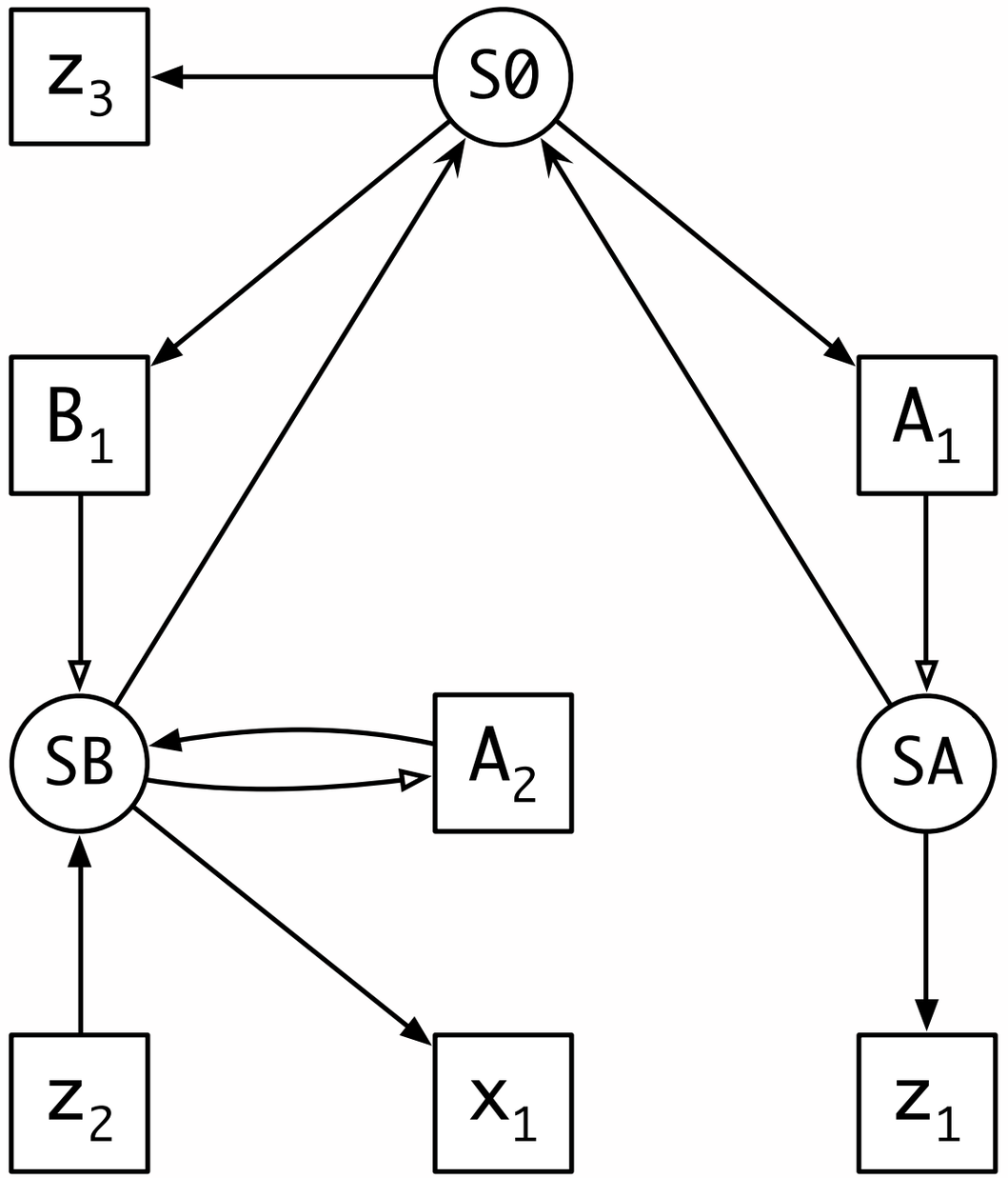
Imports shadow Parents

```
def z₃ = 2                                  S0

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



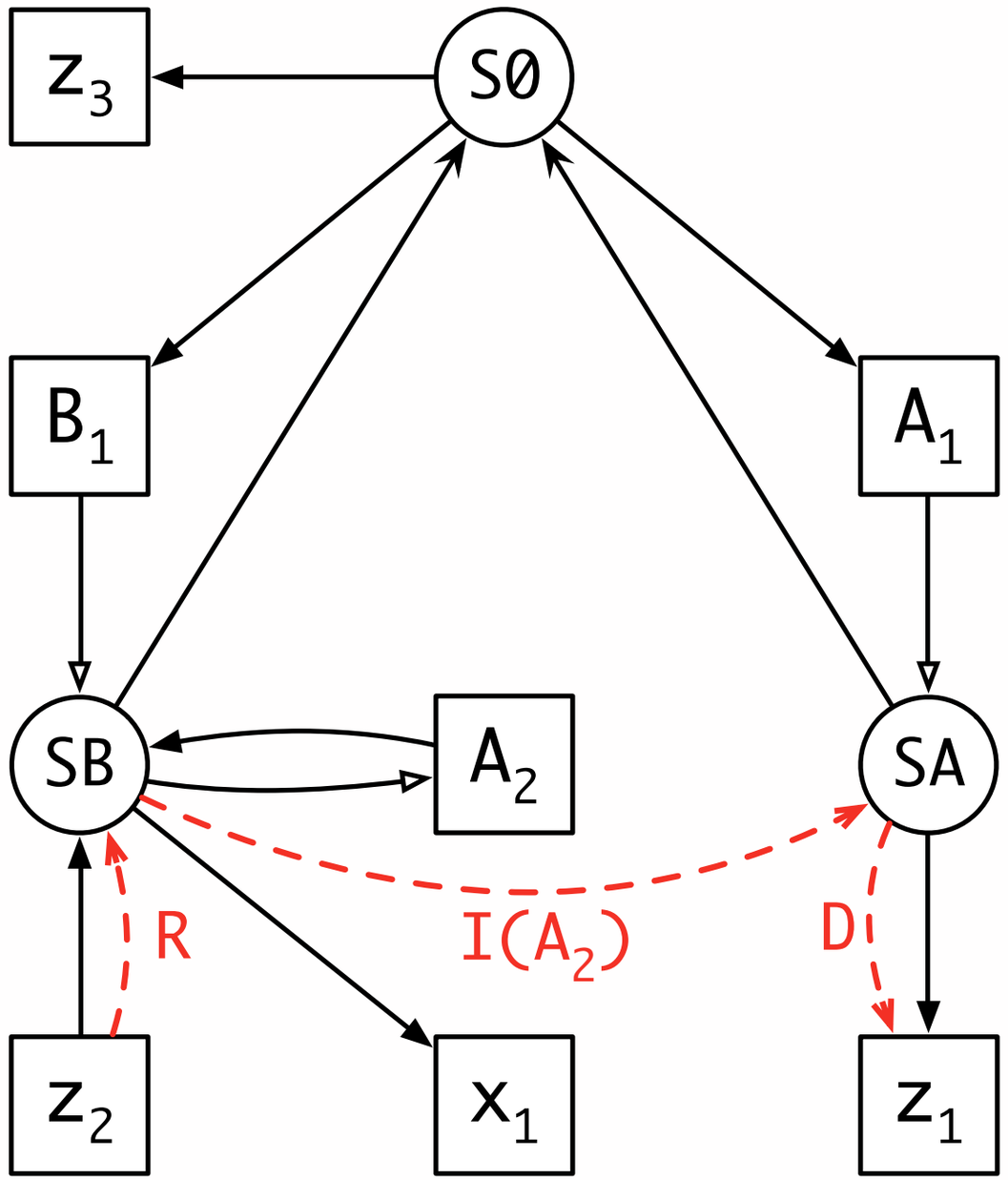
Imports shadow Parents

```
def z₃ = 2                                  S0

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



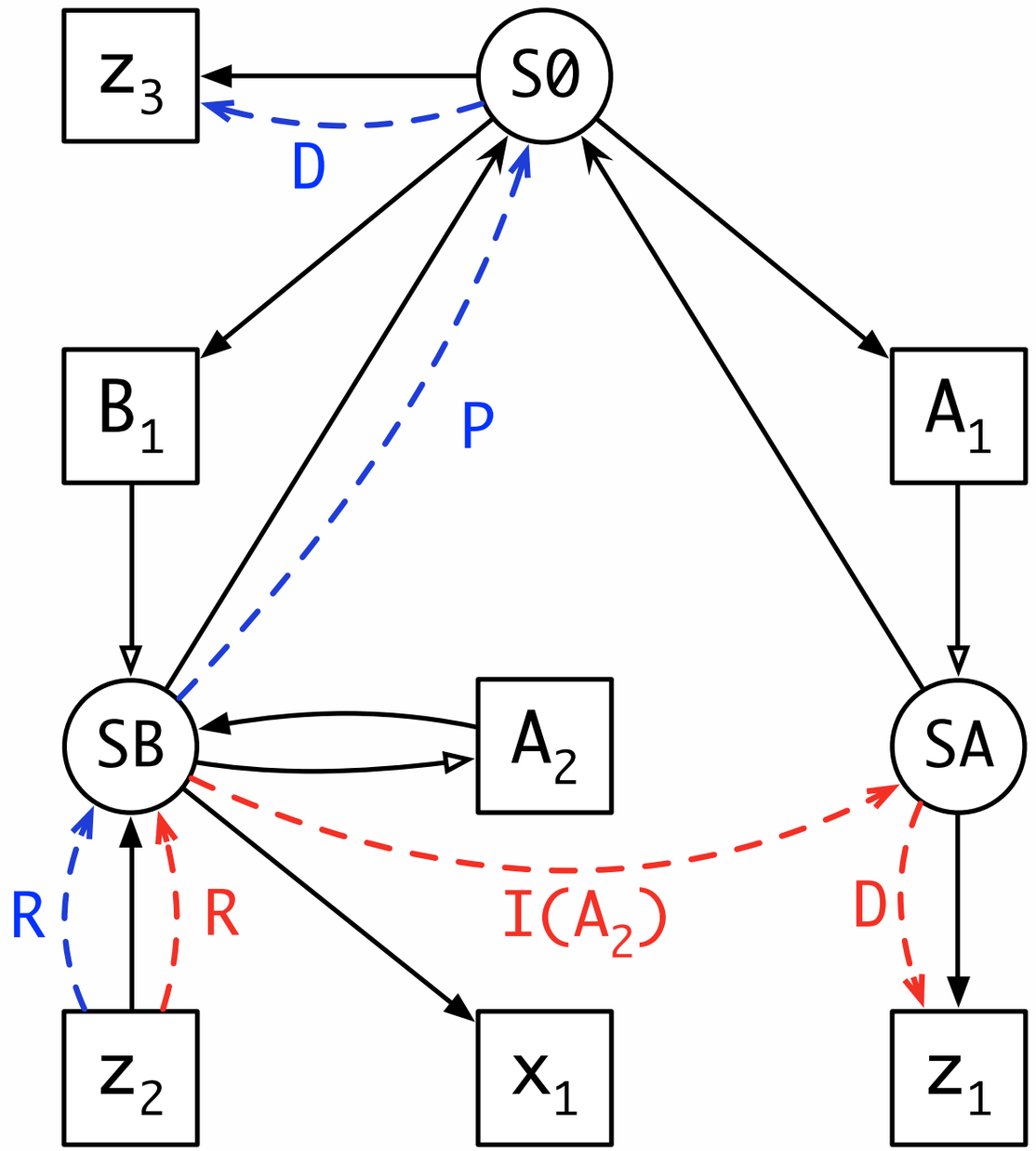
Imports shadow Parents

```
def z3 = 2                                  S0

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



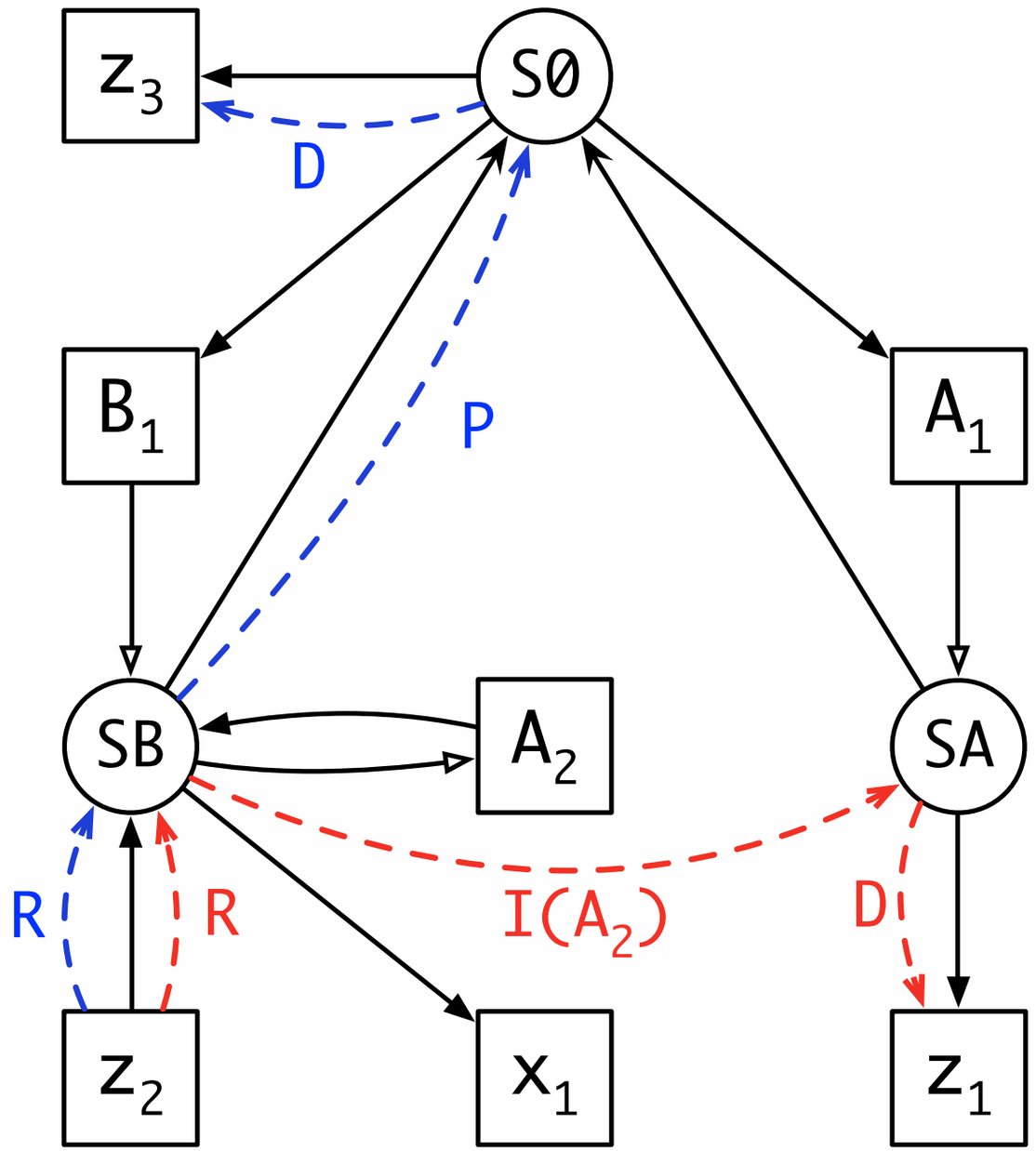
Imports shadow Parents

```
def z₃ = 2                                  S0

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



Imports shadow Parents

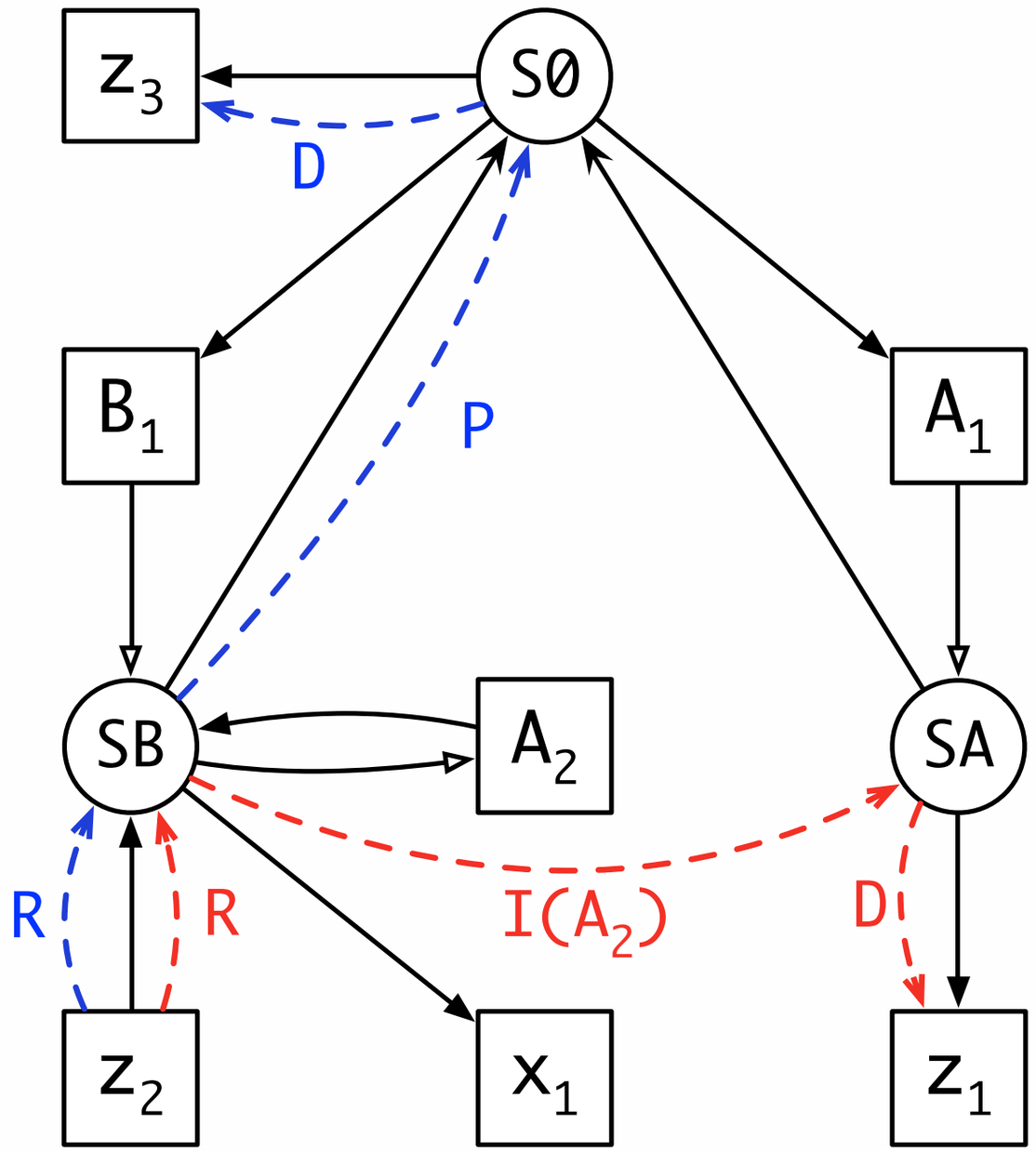
```
def z3 = 2                                  S0

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



I(_).p' < P.p

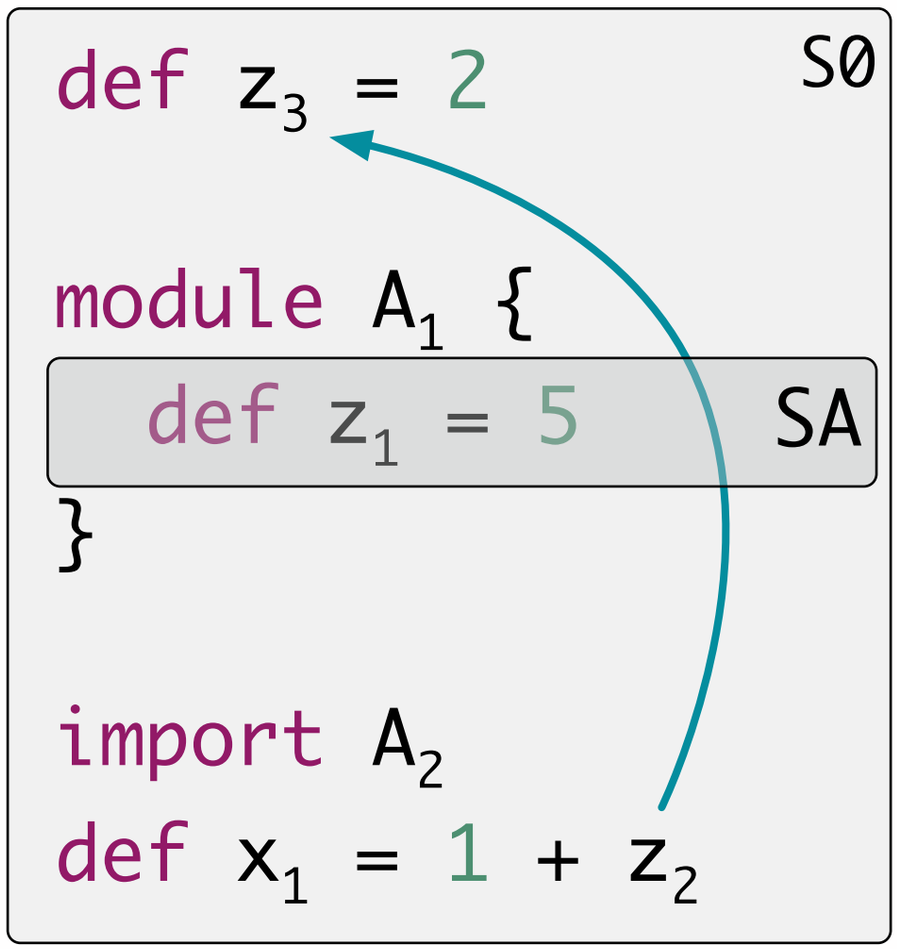
Imports shadow Parents

```
def z3 = 2                                  S0

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



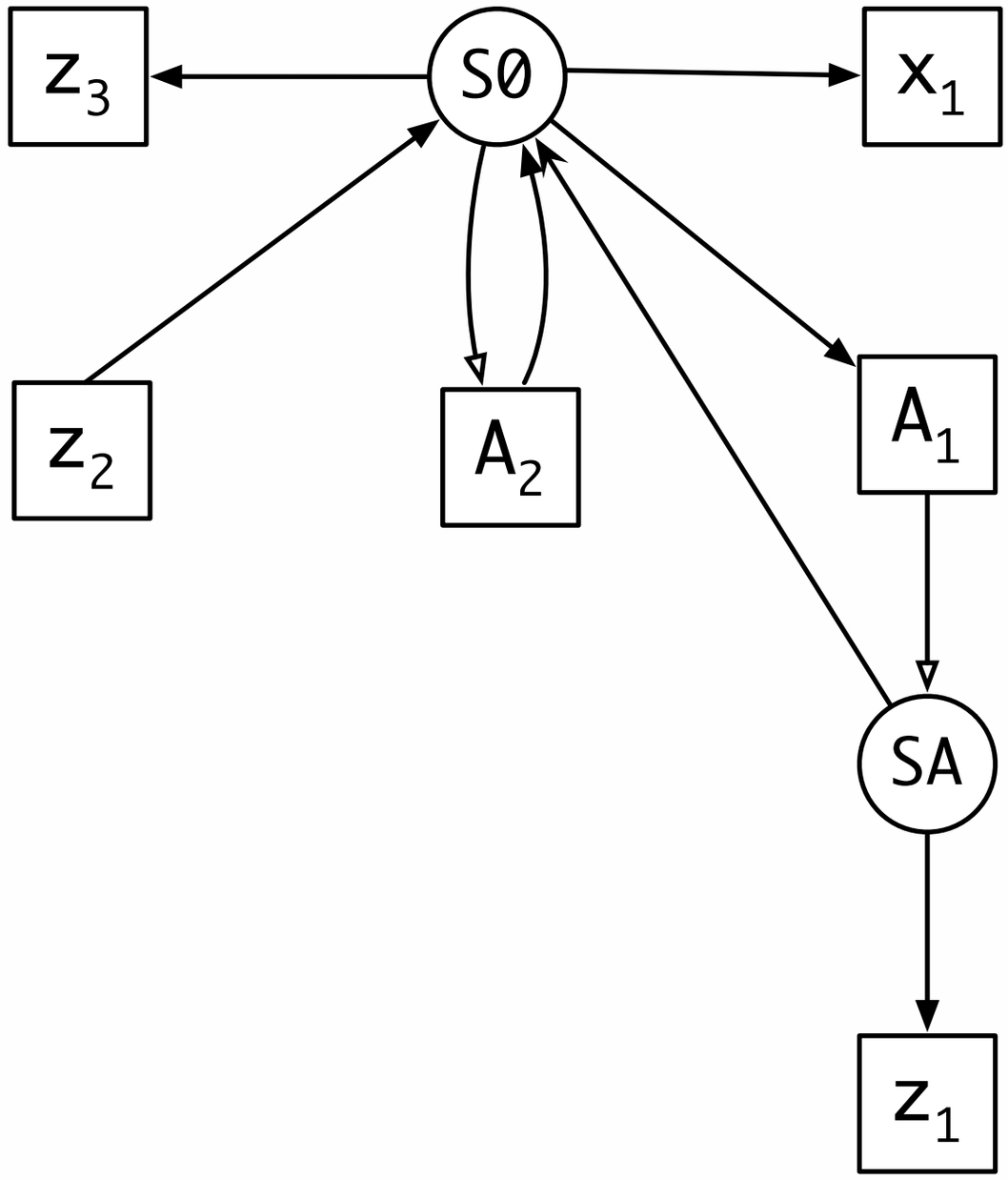
$$\frac{}{I(\_).p' < P.p} \quad \Rightarrow \quad R.I(A_2).D < R.P.D$$

Imports vs. Includes



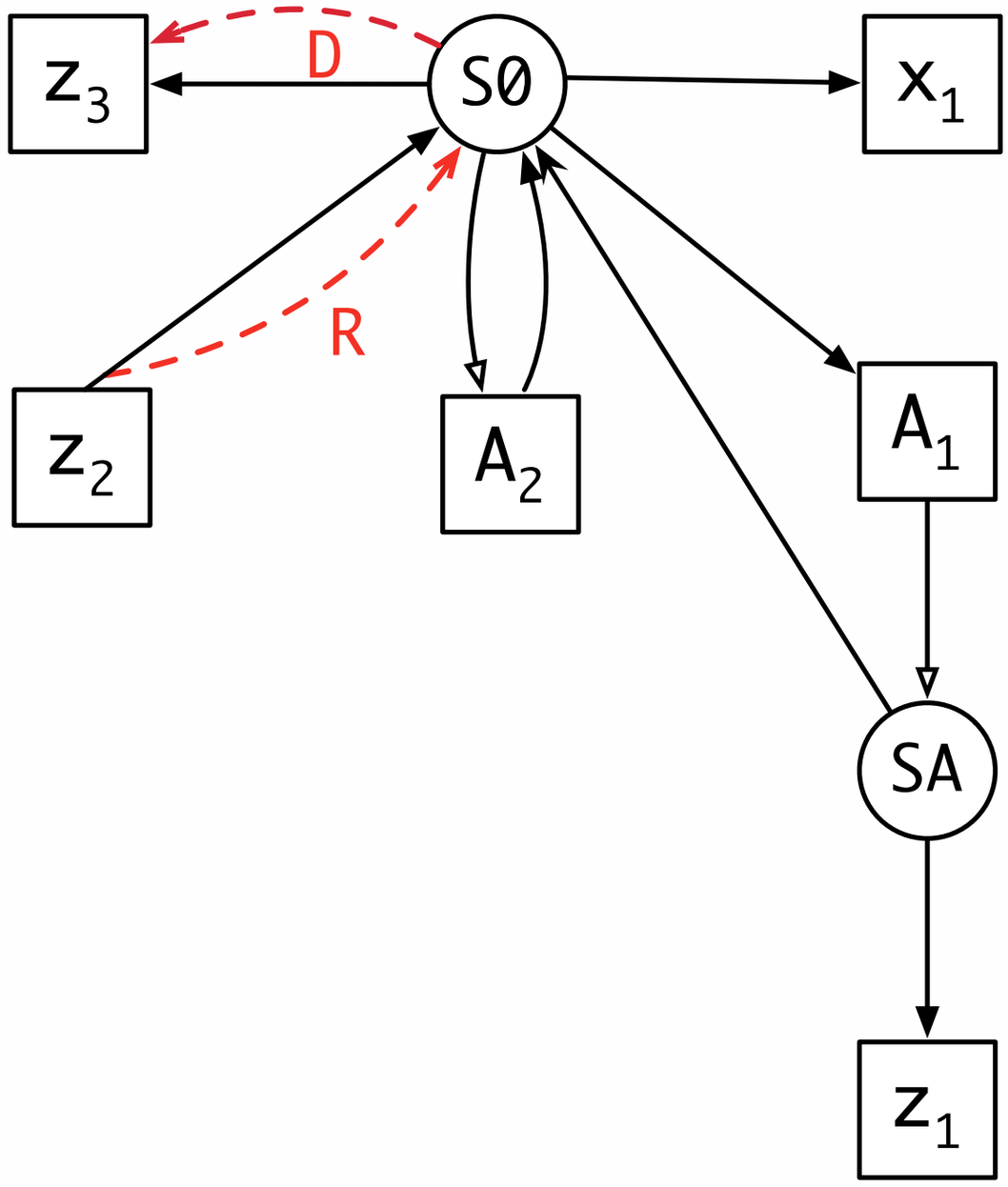
Imports vs. Includes

```
                                    S0
def z3 = 2
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



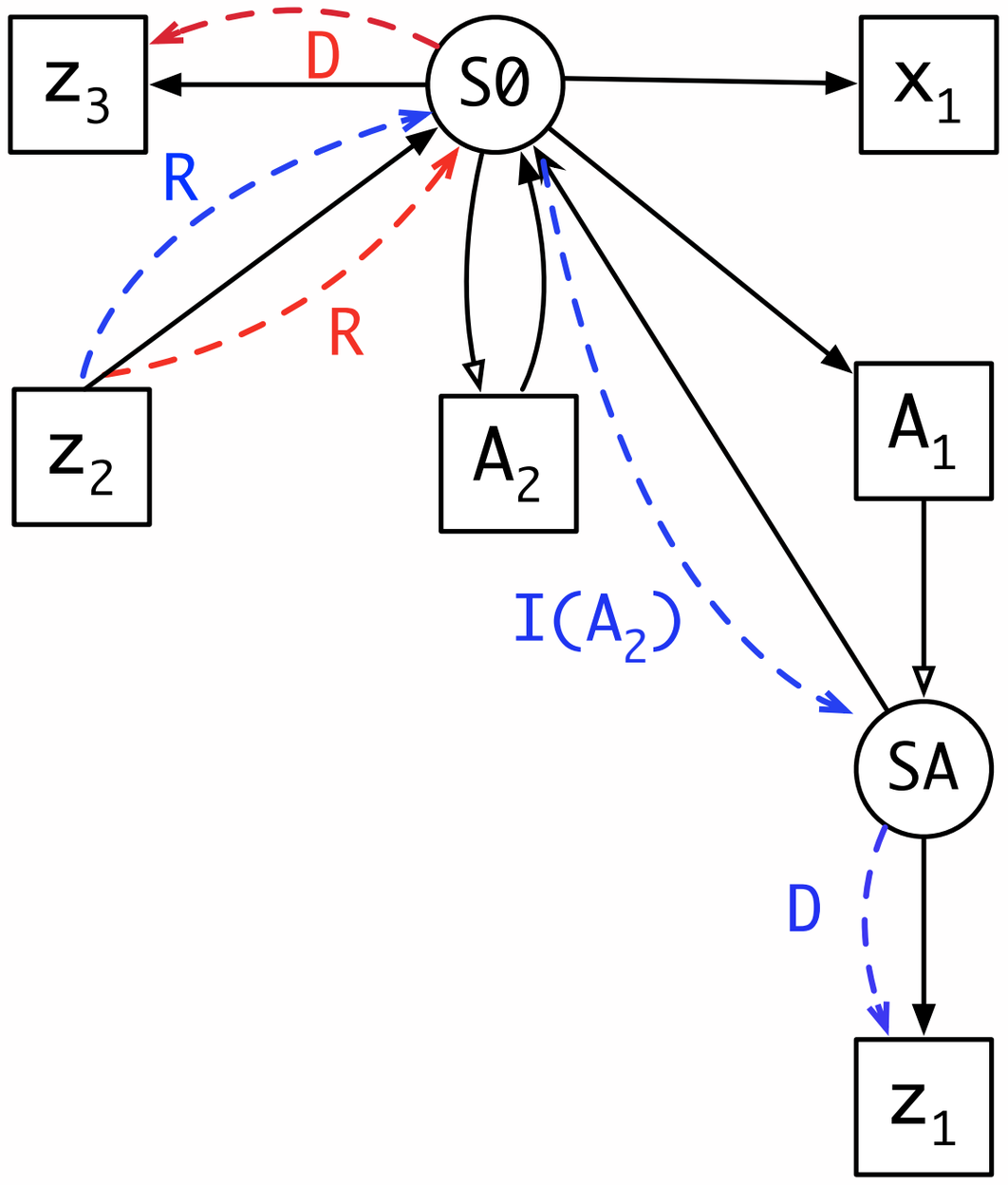
Imports vs. Includes

```
                                  S0
def z3 = 2
module A1 {
  def z1 = 5                      SA
}

import A2
def x1 = 1 + z2
```



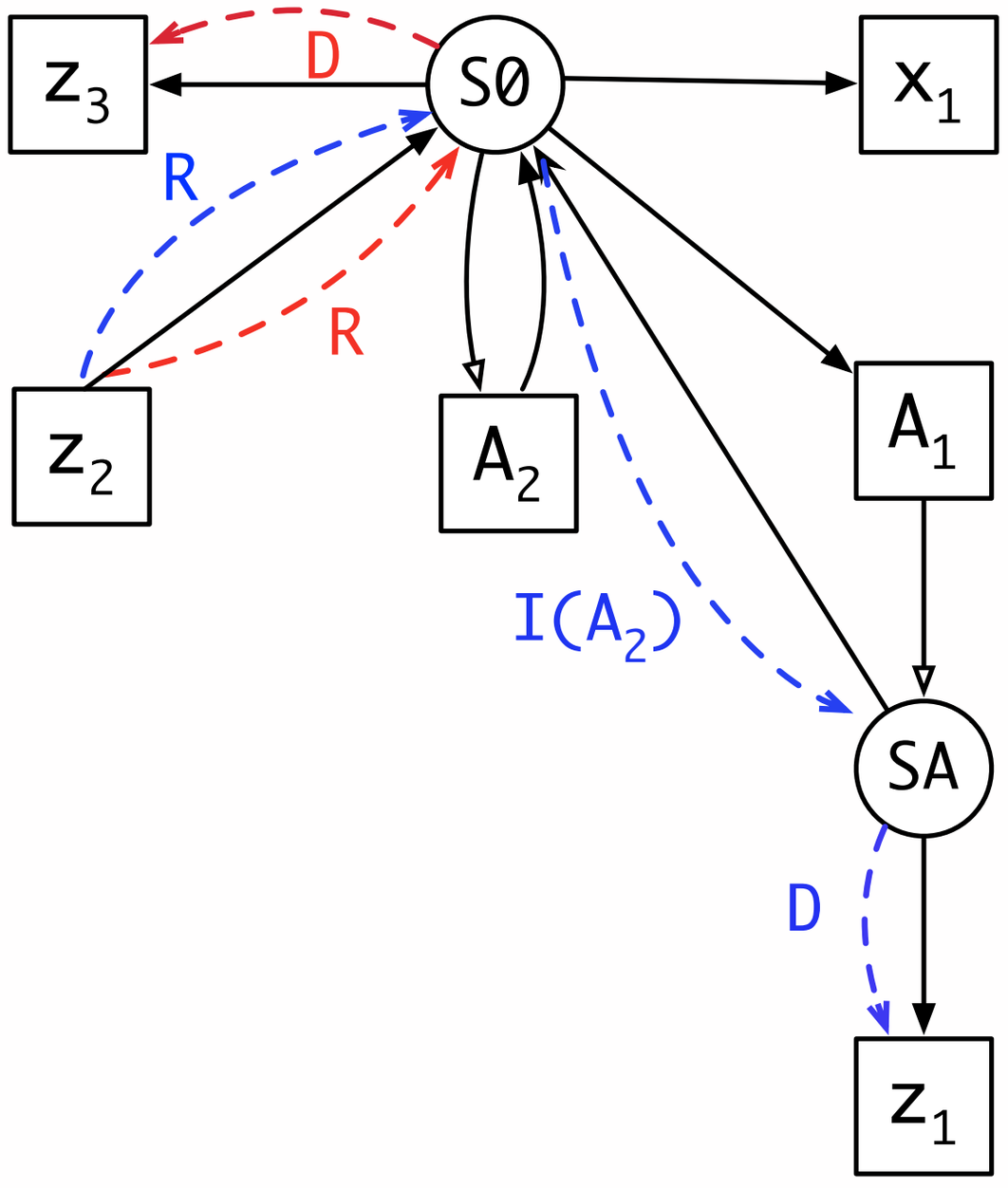
Imports vs. Includes

```
def z3 = 2                          S0

module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



Imports vs. Includes

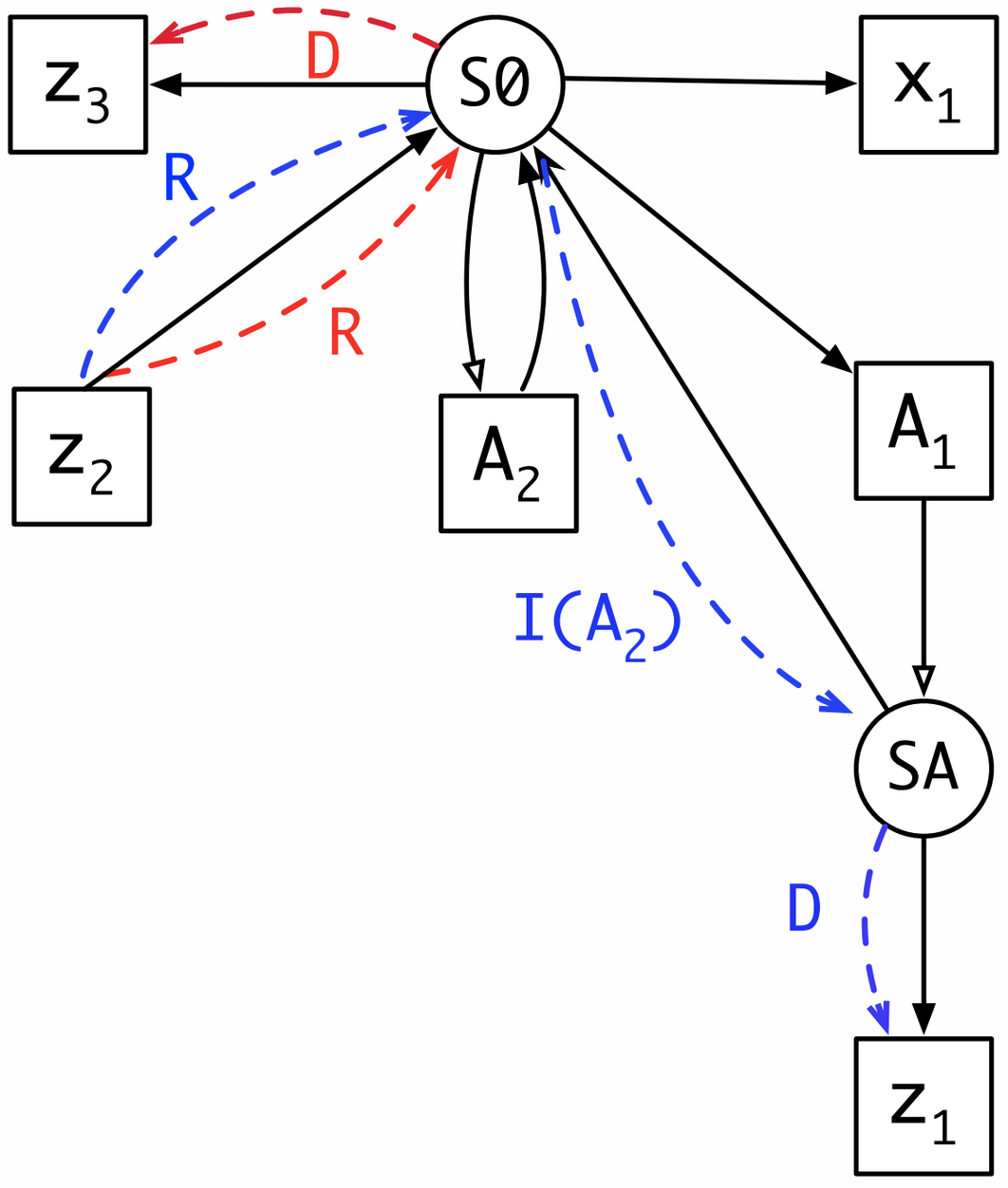
```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



D < I(_).p’

Imports vs. Includes

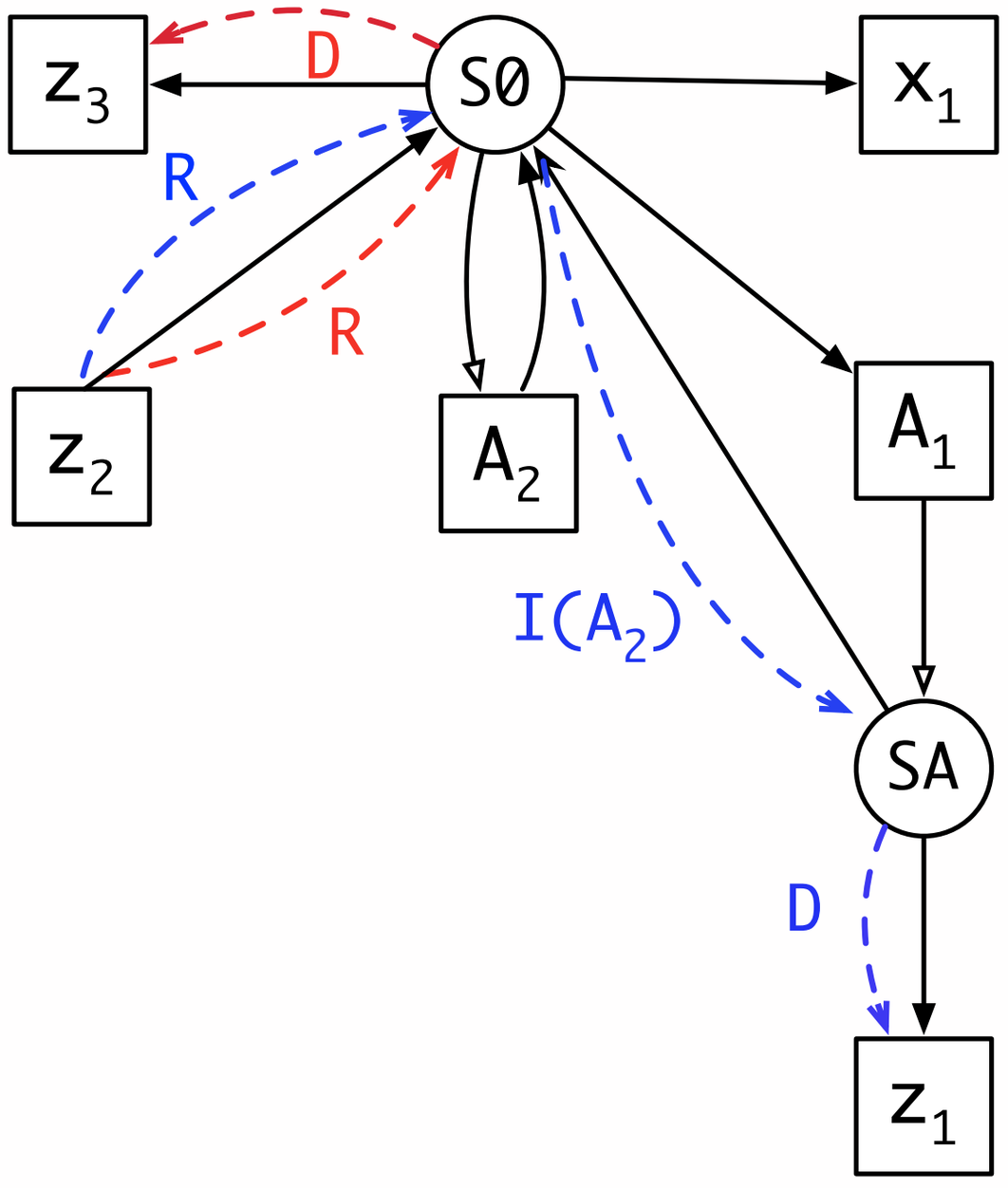
```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



$$\frac{}{\texttt{D} < \texttt{I(\_).p'}} \quad \Rightarrow \quad \texttt{R.D} < \texttt{R.I}(A_2)\texttt{.D}$$

Imports vs. Includes

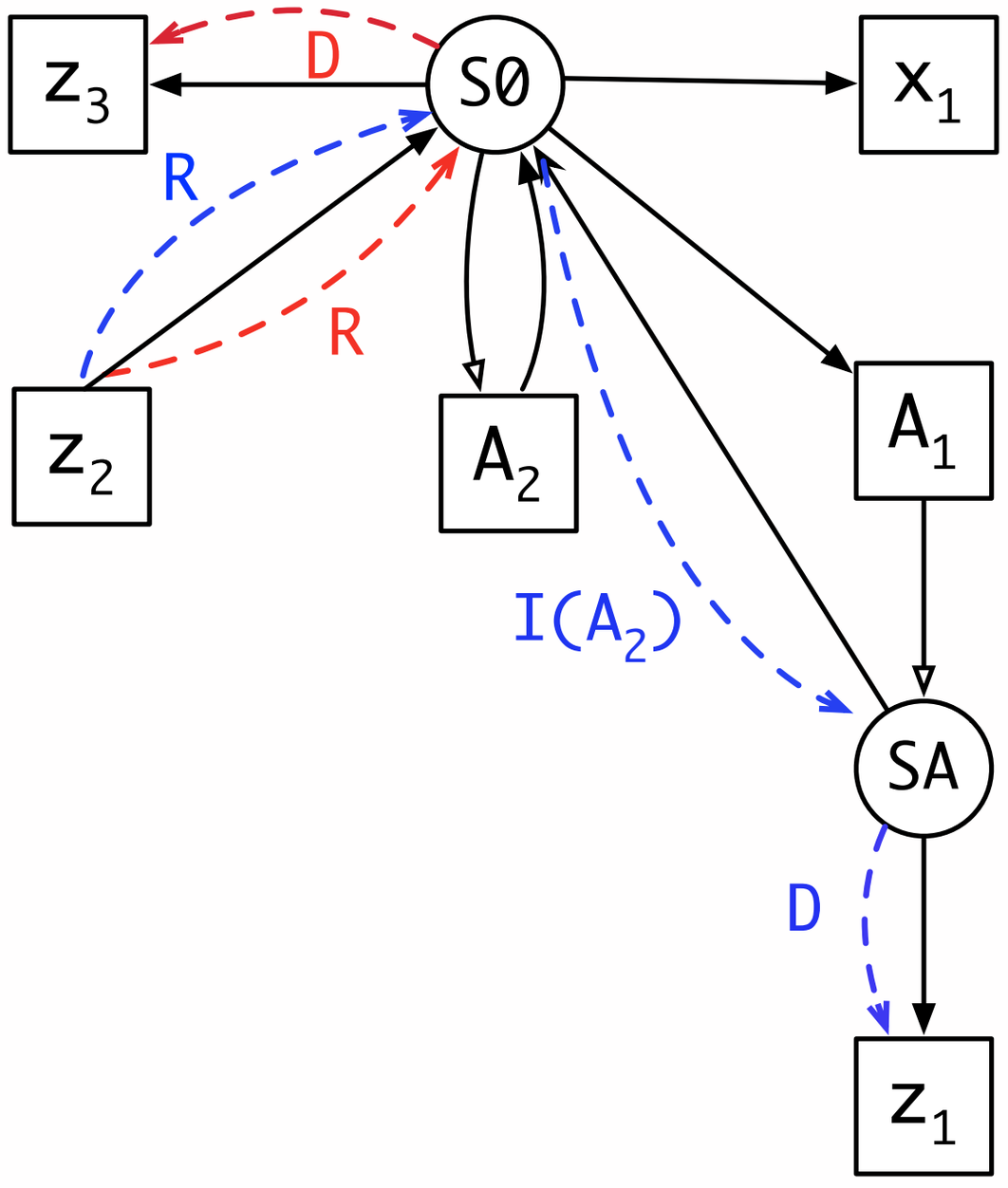
```
                                  S0
def z3 = 2
module A1 {
  def z1 = 5                      SA
}
include A2
def x1 = 1 + z2
```

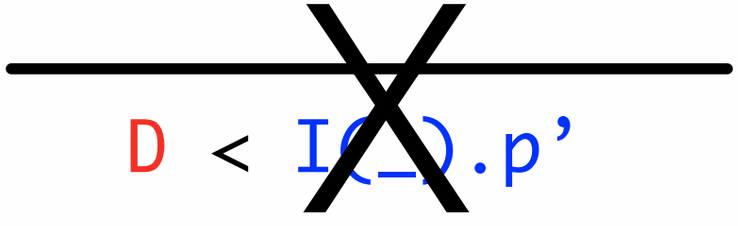


$$\frac{}{D < I(\_).p'} \quad \Rightarrow \quad R.D < R.I(A_2).D$$

Imports vs. Includes

```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}
include A2
def x1 = 1 + z2
```

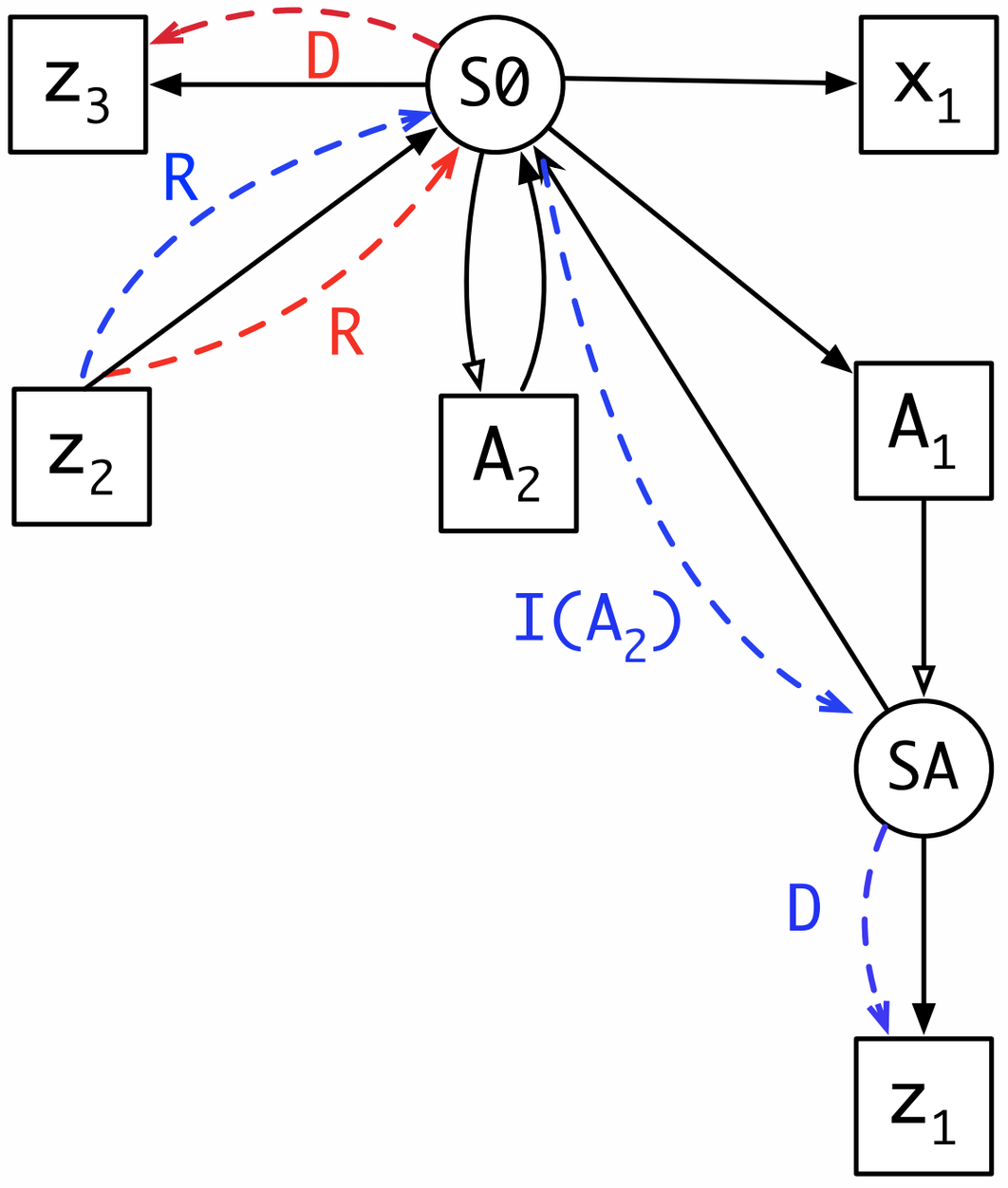


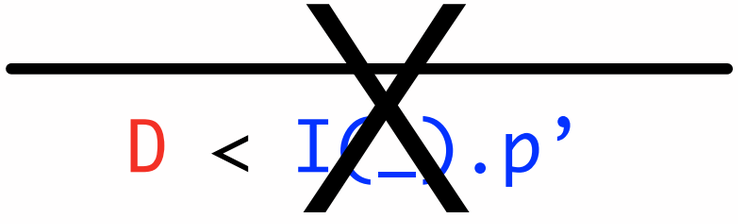


$\Rightarrow$ $R.D < R.I(A_2).D$

Imports vs. Includes

```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}
include A2
def x1 = 1 + z2
```





Qualified Names

S0

```
module N₁ {
    def s₁ = 5
}

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

Qualified Names

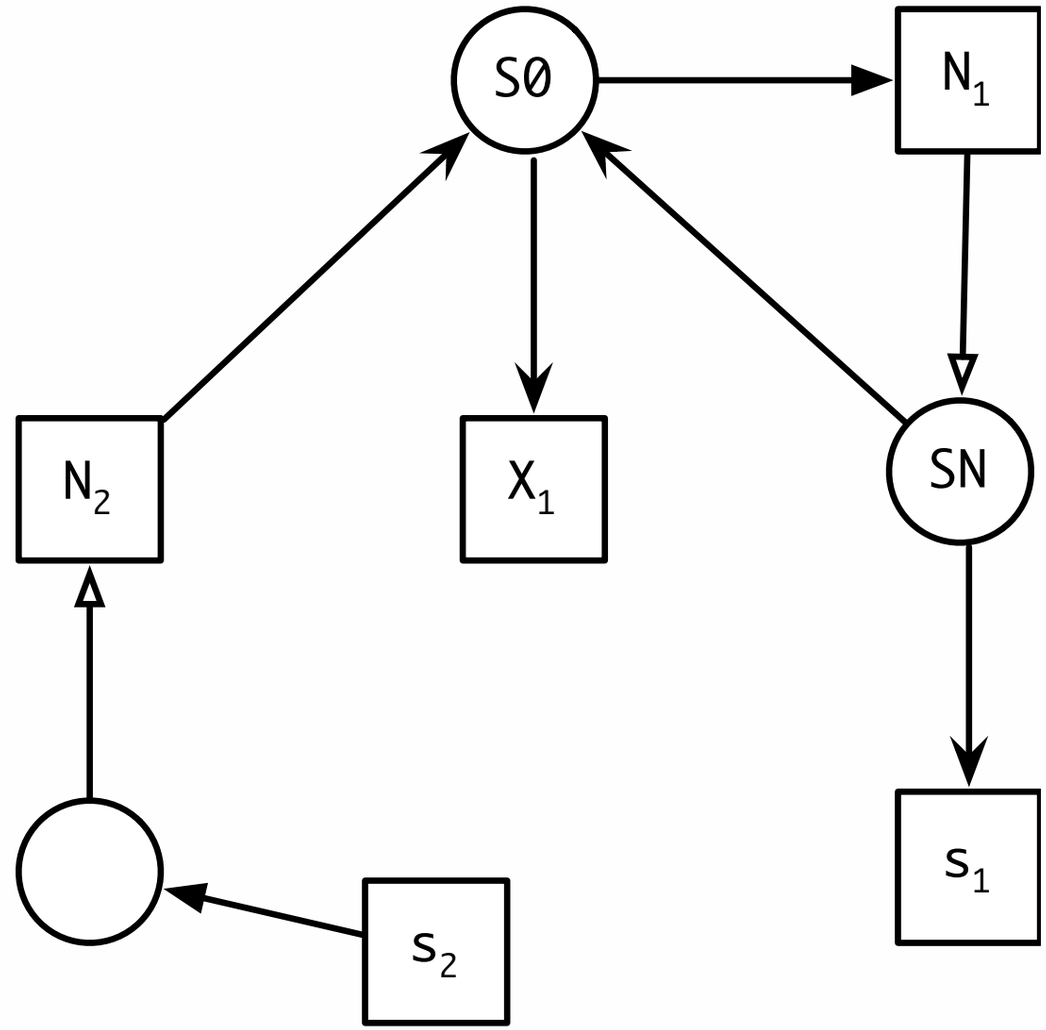
S0

```
module N1 {
    def s1 = 5
}

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

Qualified Names

S0

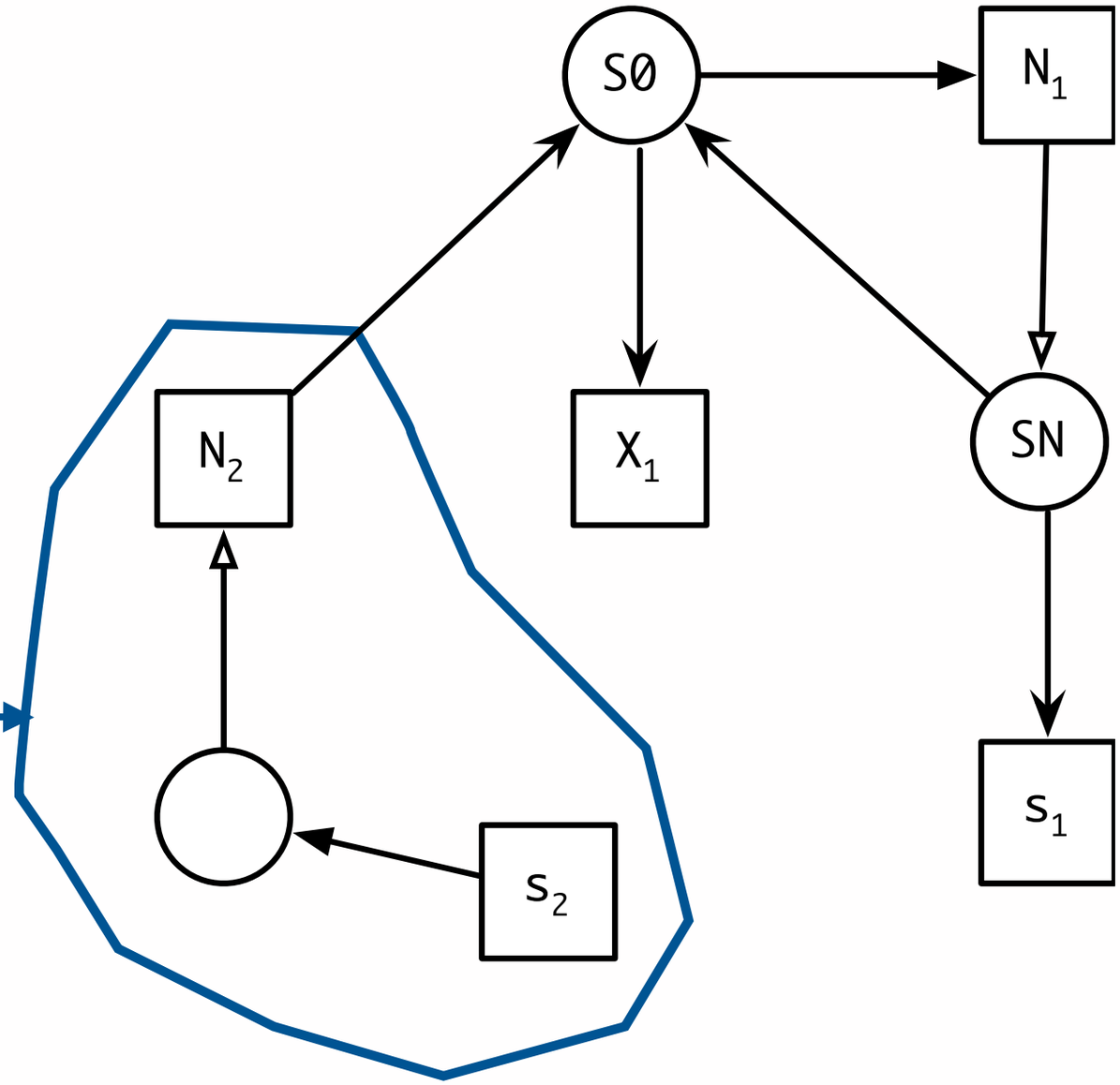
```
module N₁ {
    def s₁ = 5
}

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



Qualified Names

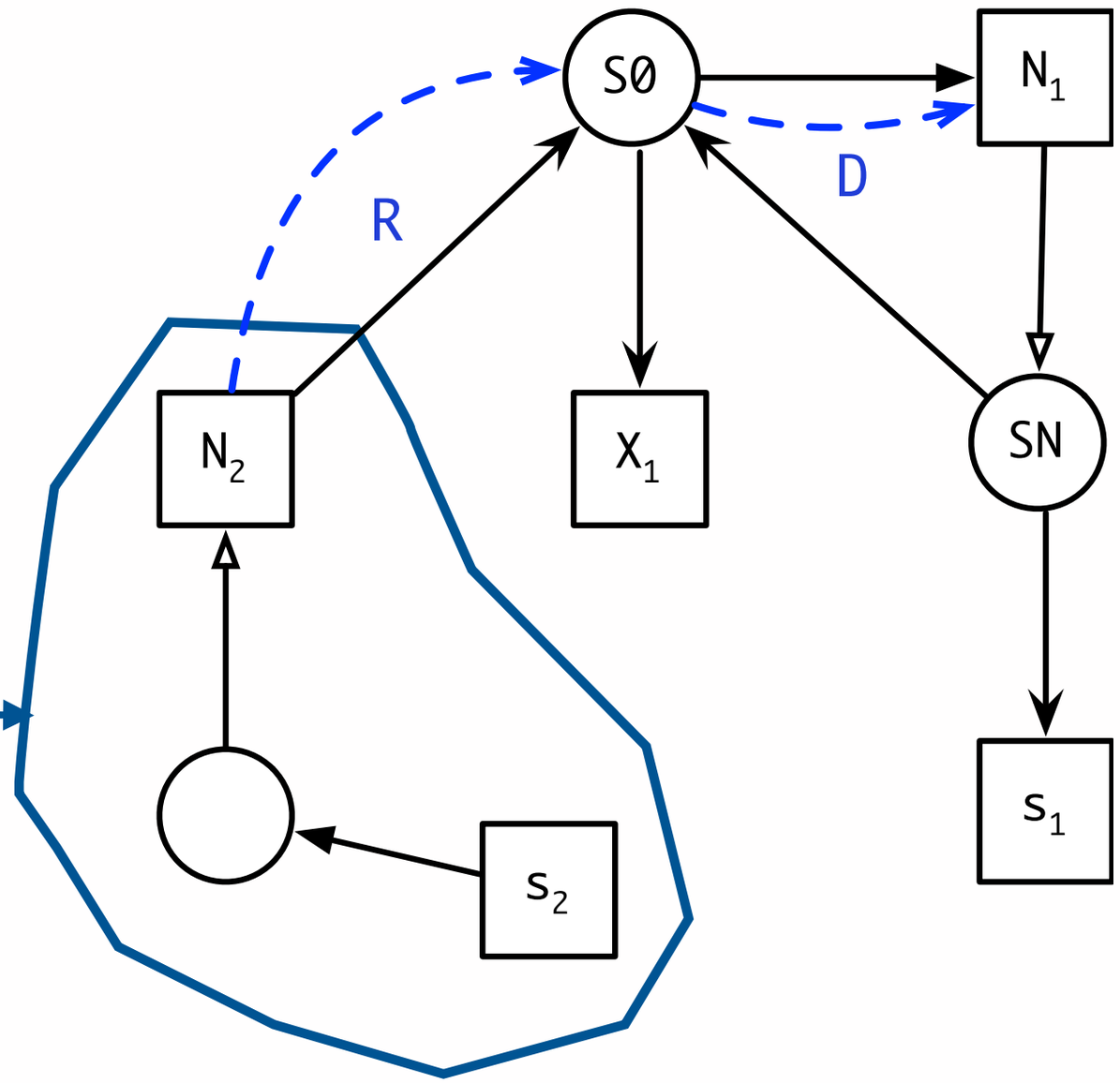
S0

```
module N1 {
    def s1 = 5
}

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



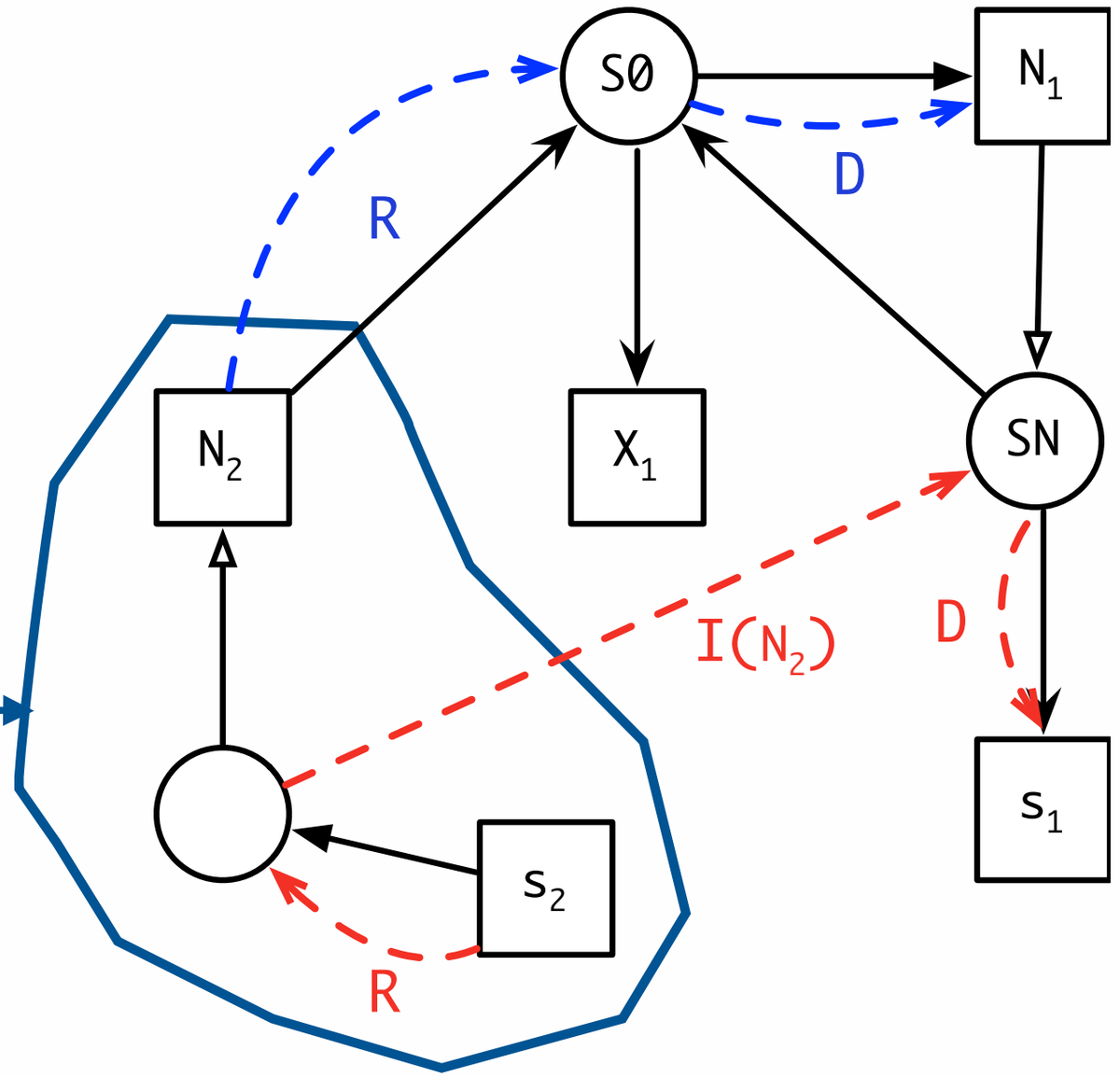
Qualified Names

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

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



Qualified Names

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

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

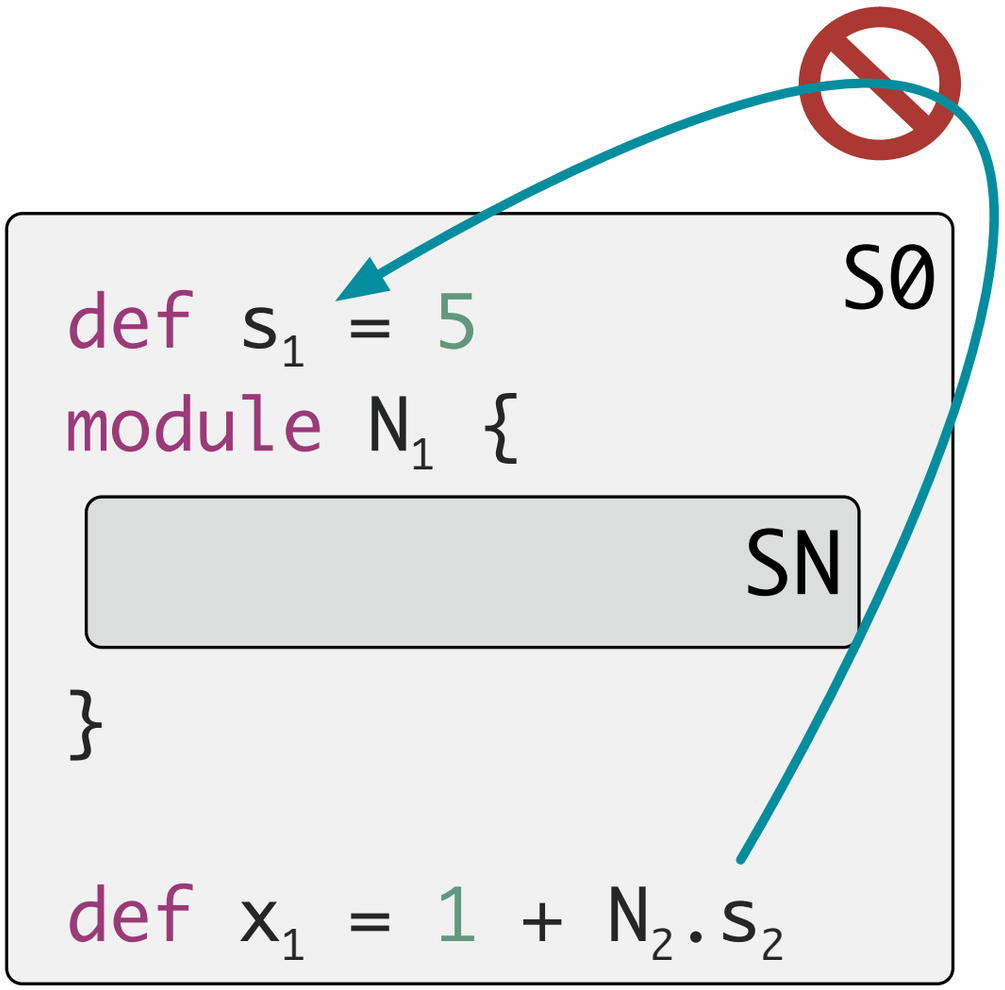


Import Parents

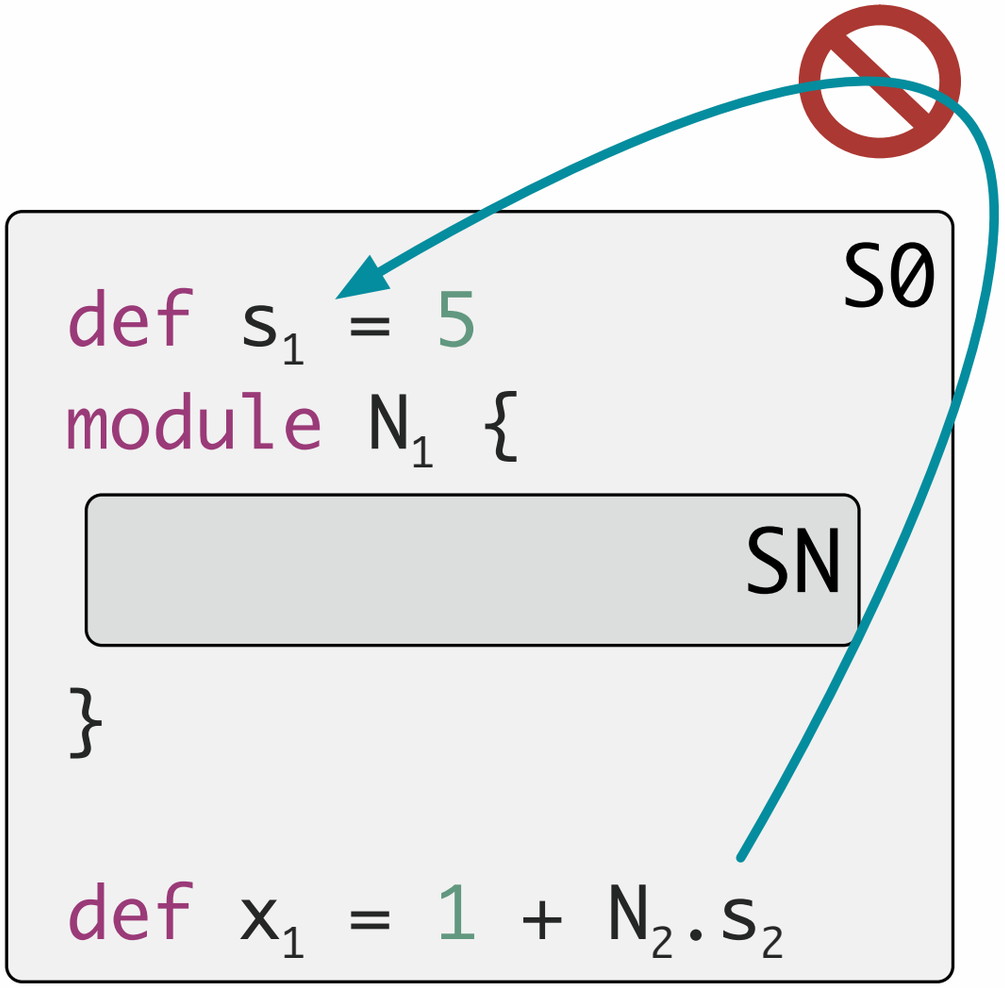
S0

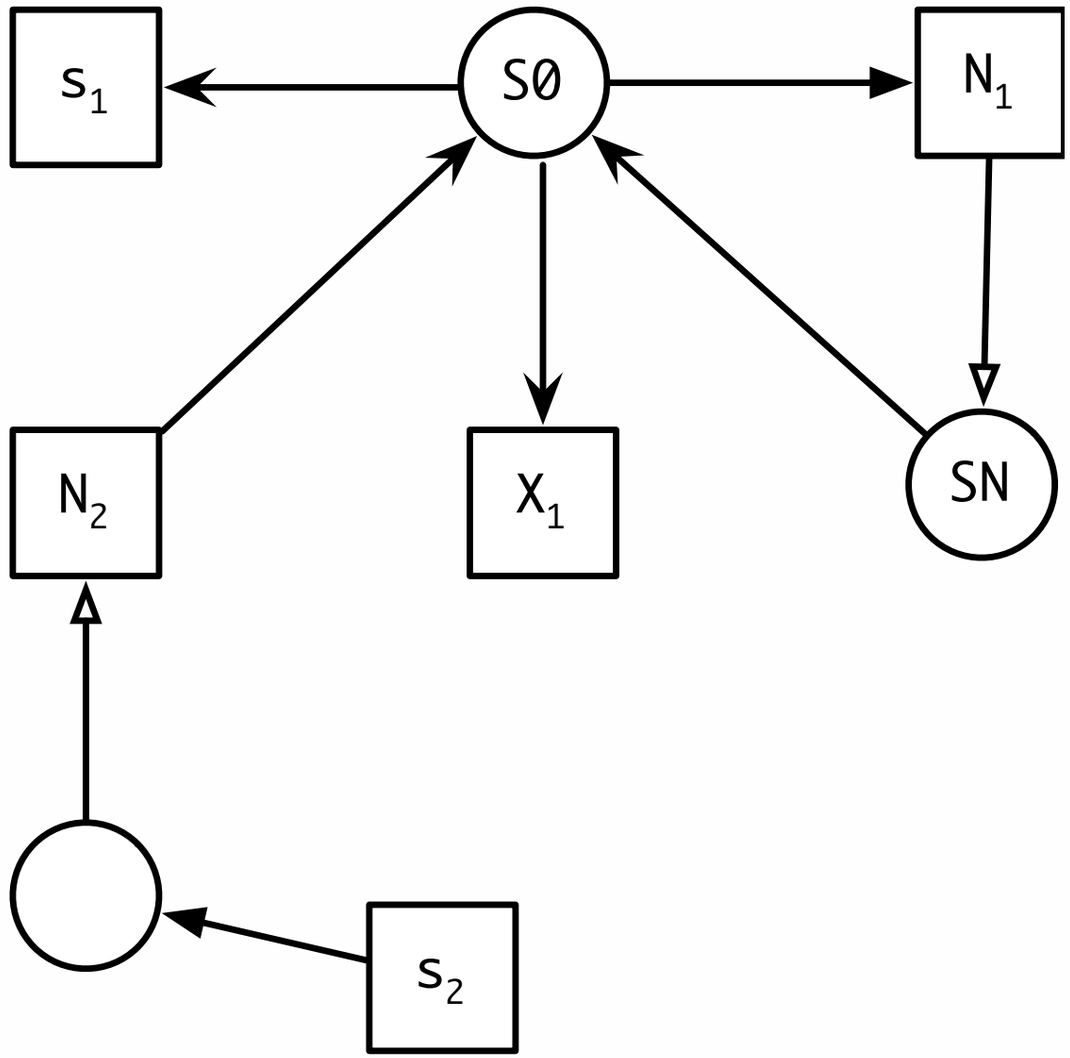
```
def s₁ = 5
module N₁ {
  SN
}

def x₁ = 1 + N₂.s₂
```

Import Parents

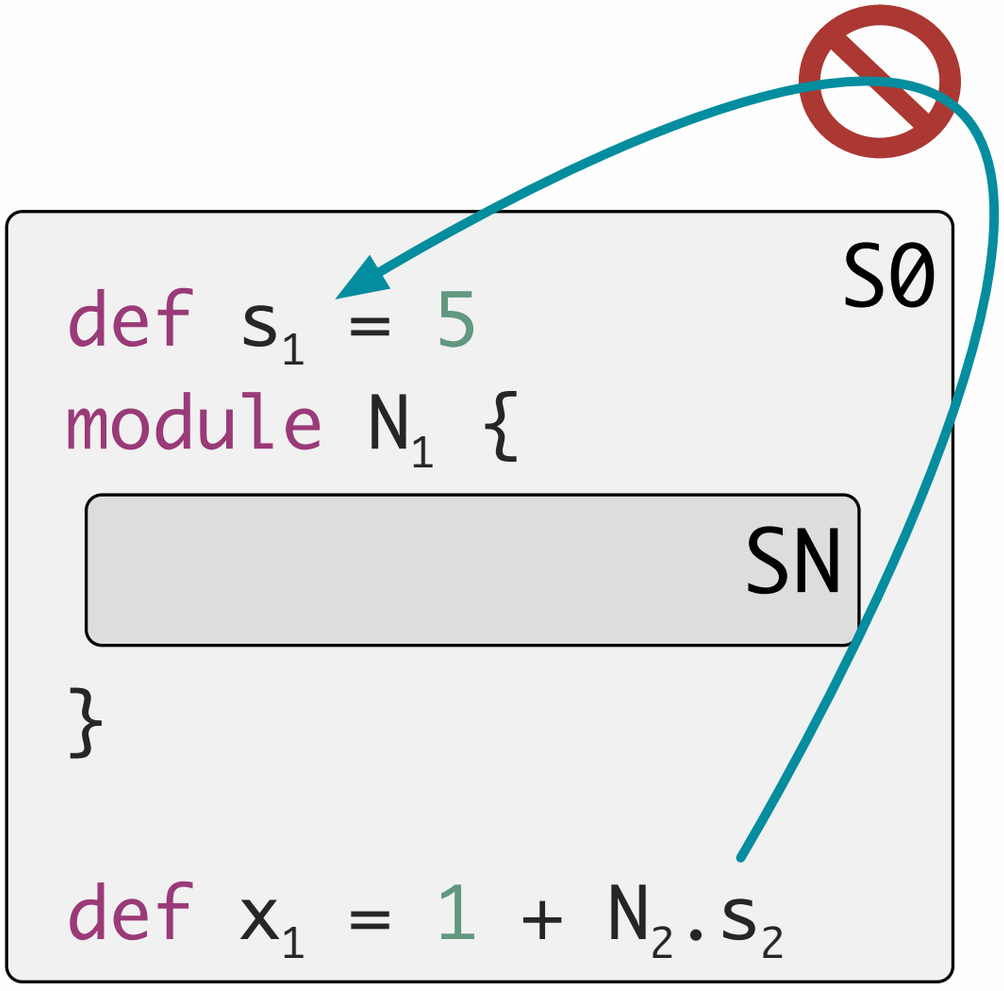


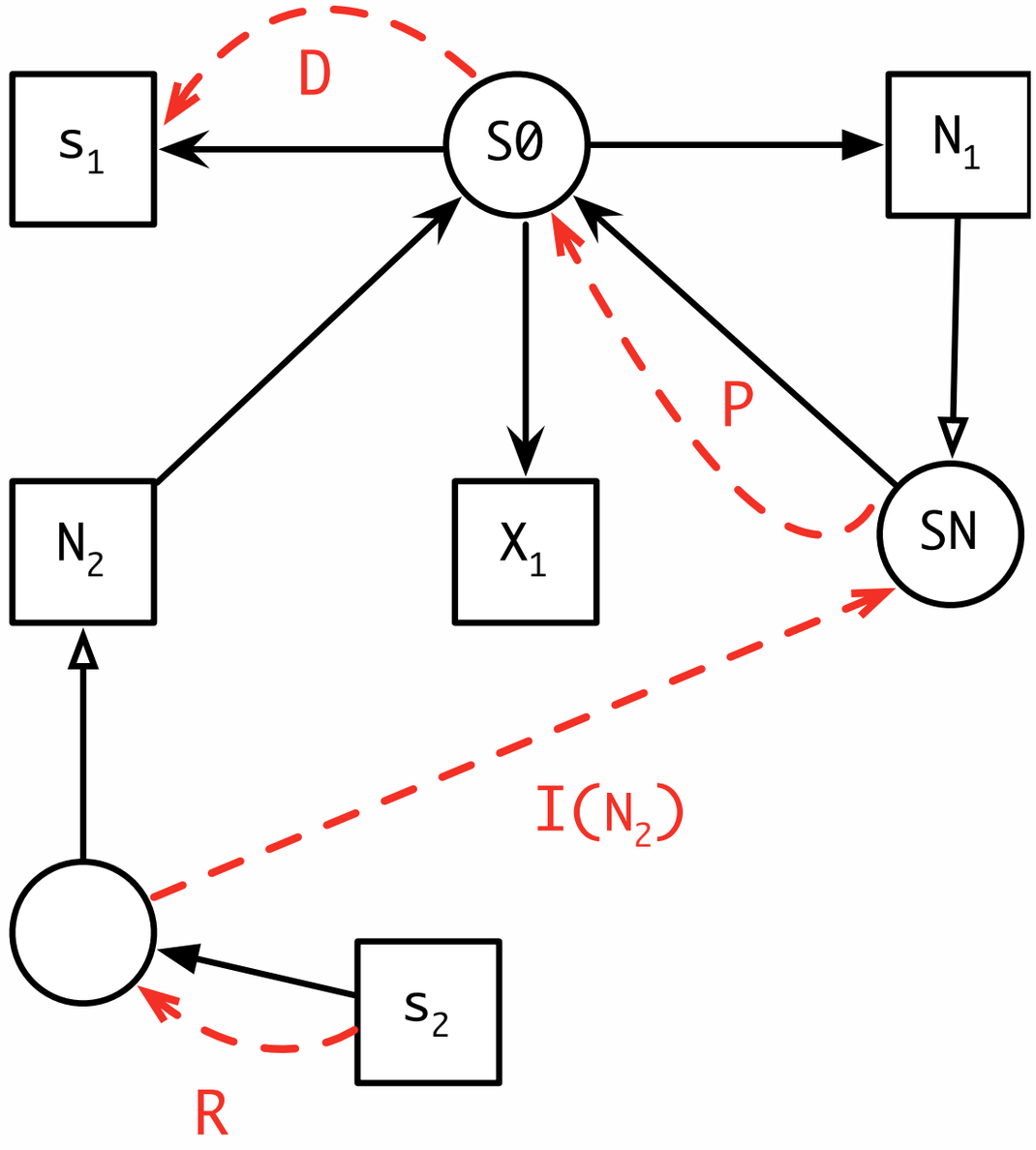
Import Parents



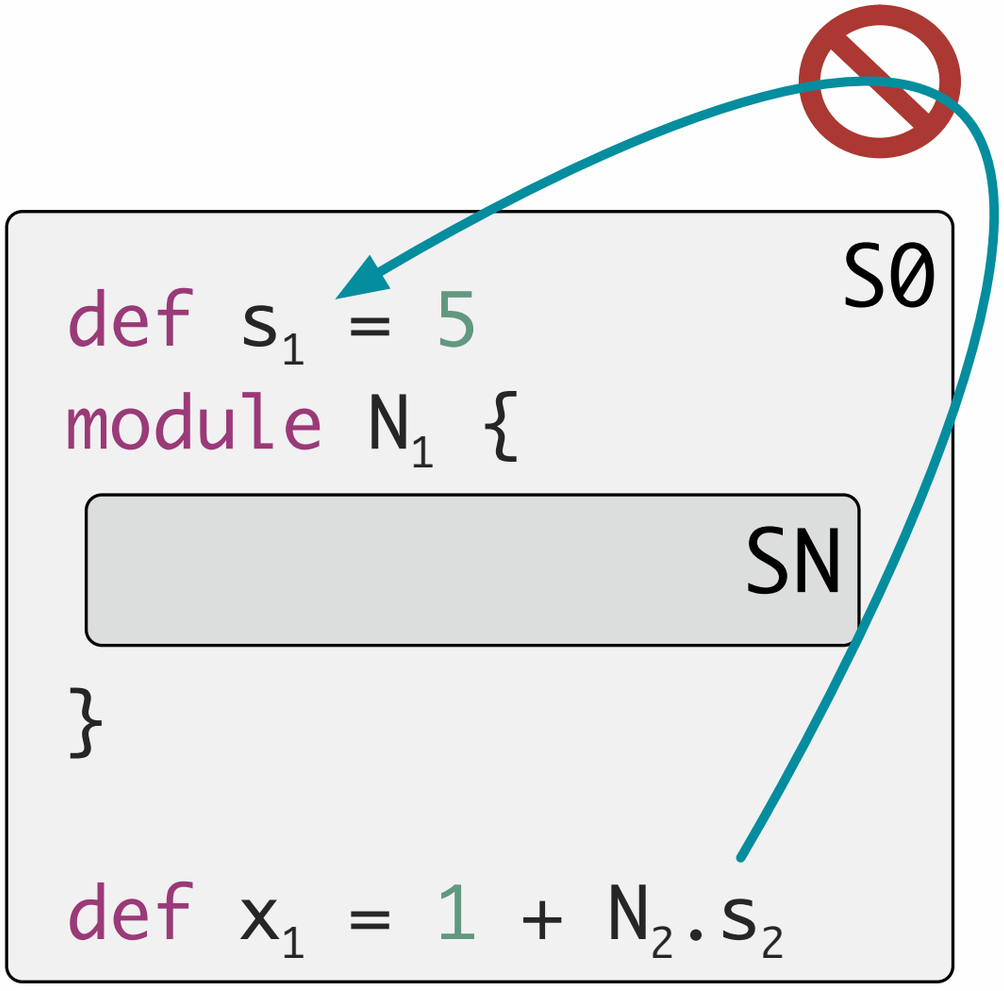


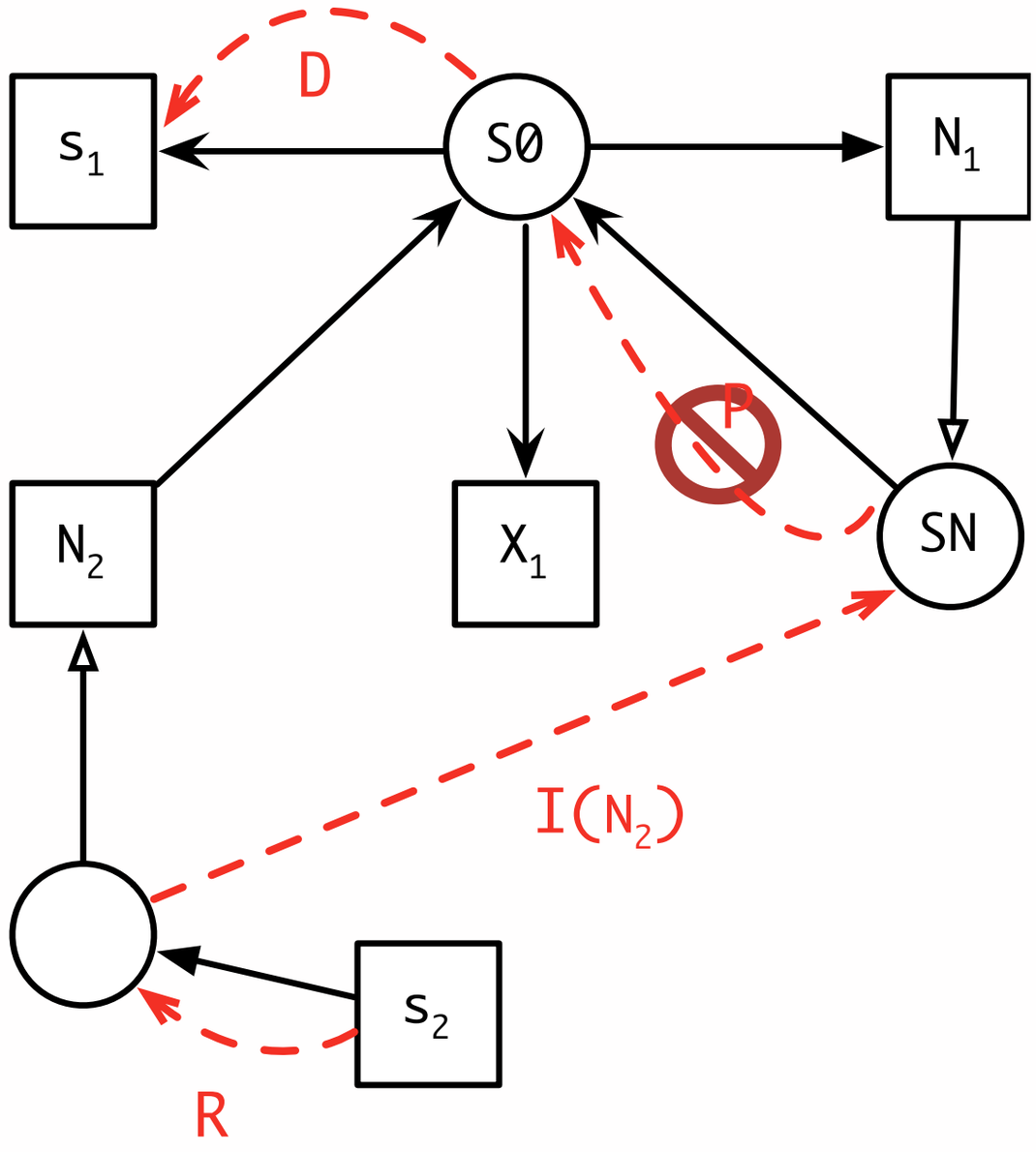
Import Parents



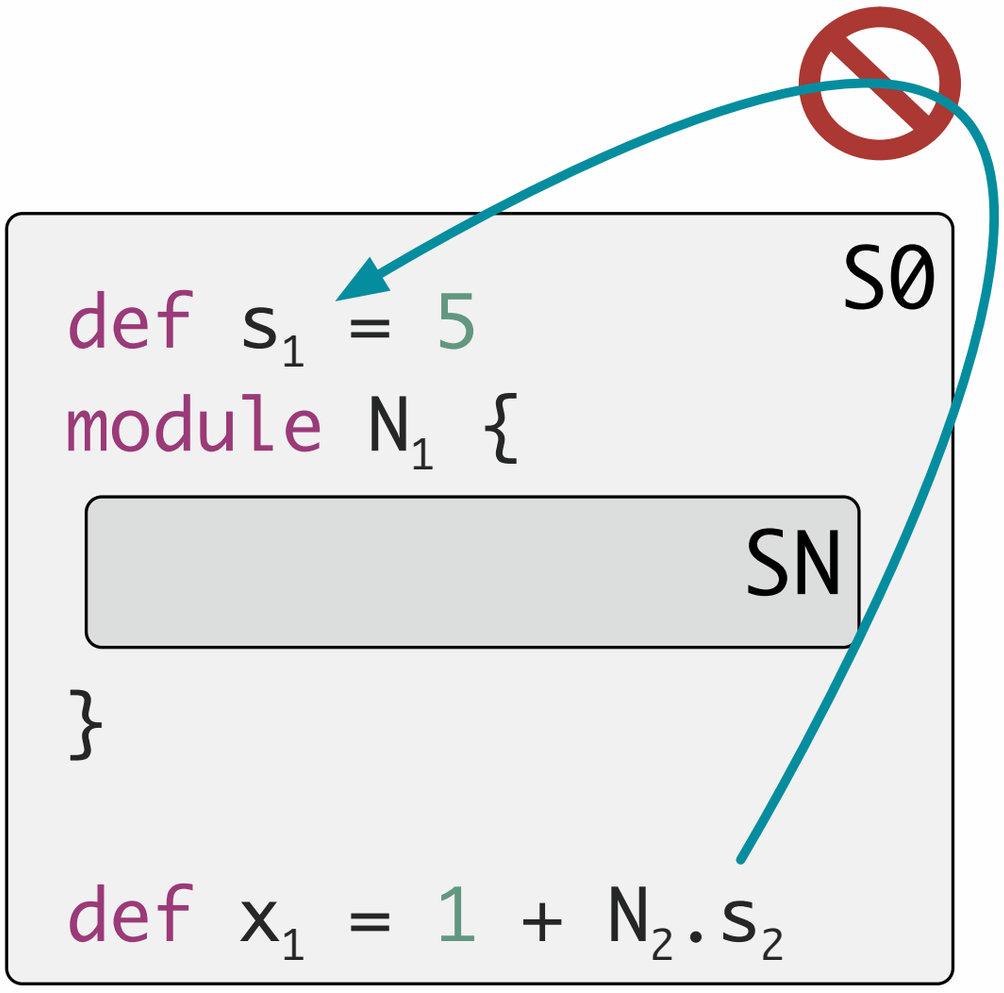


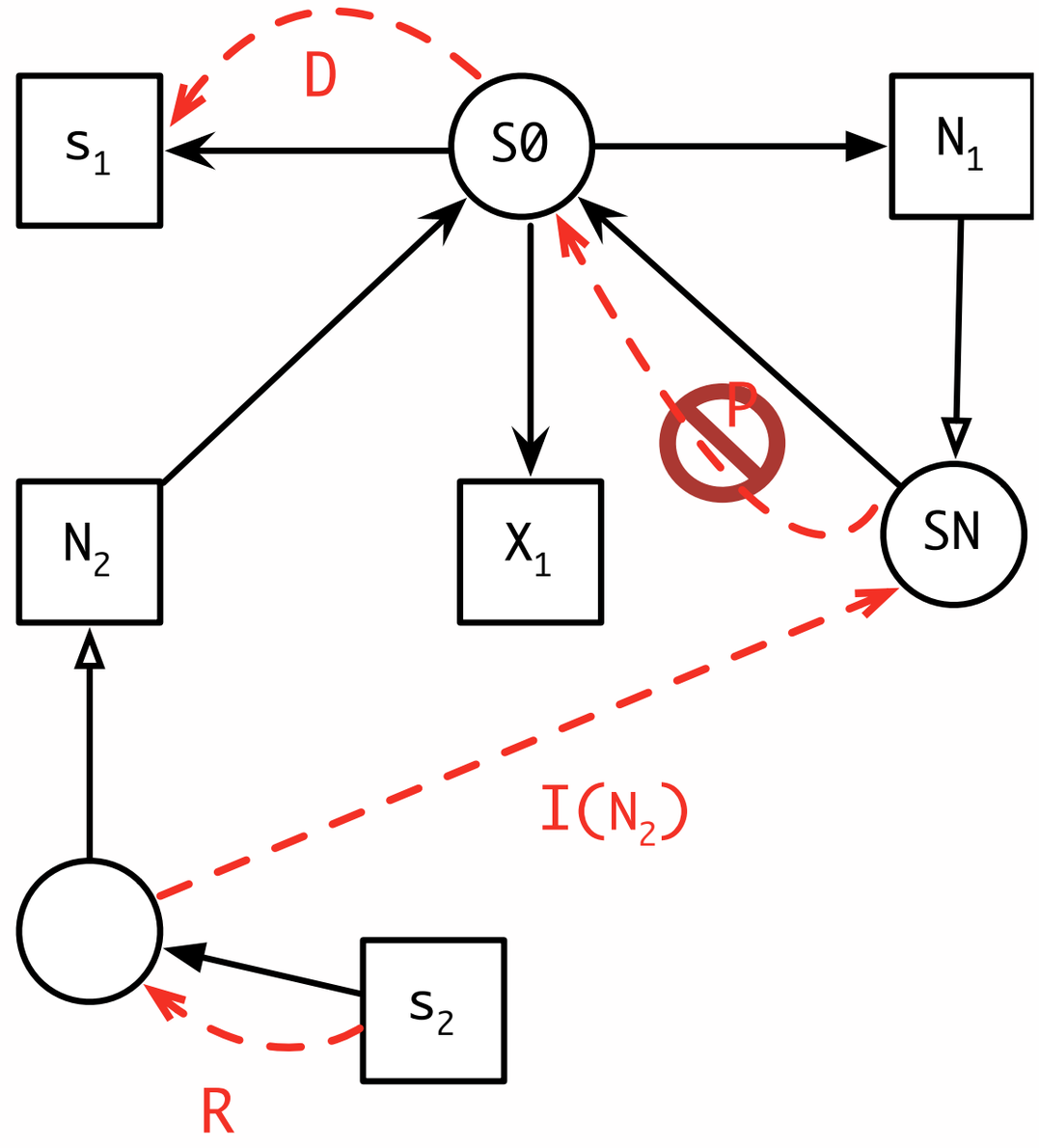
Import Parents





Import Parents



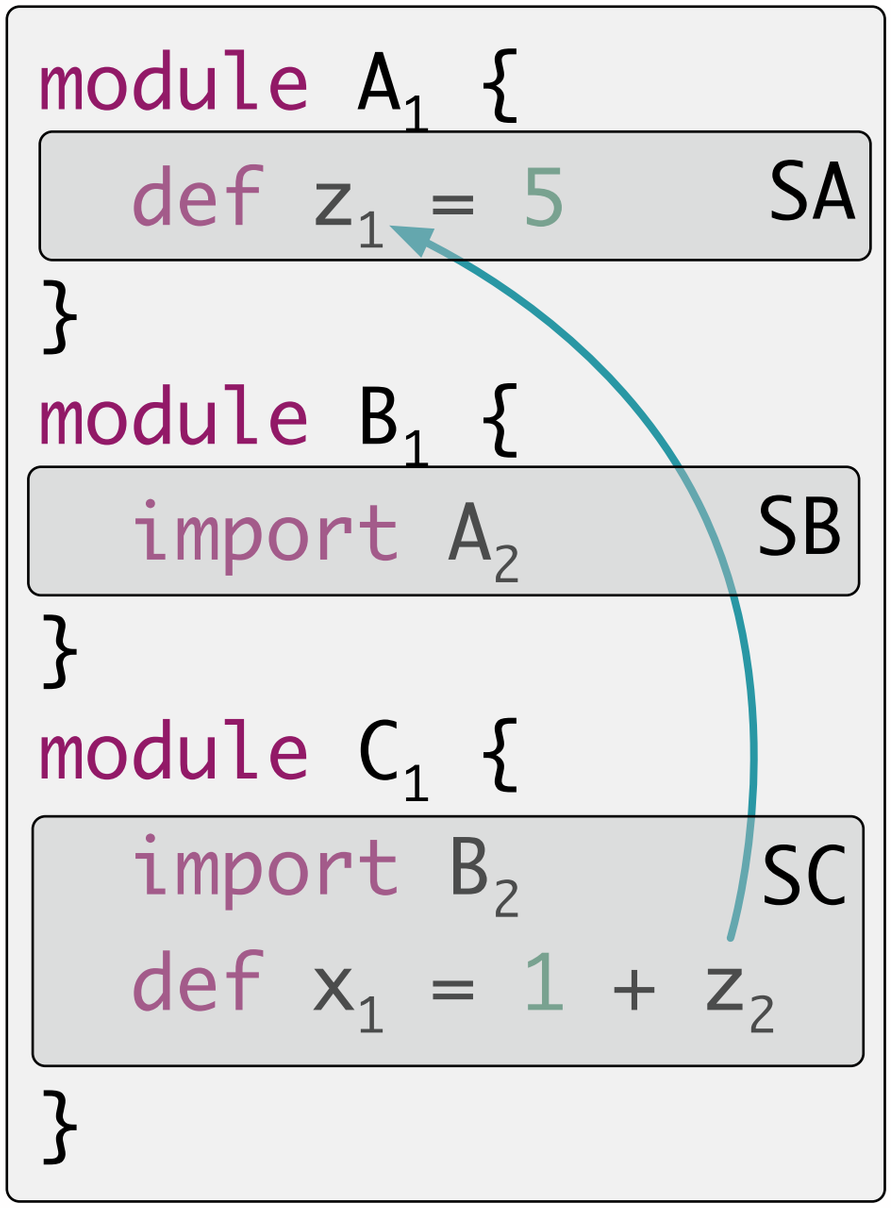


Well formed path: R.P*.I(_)*.D

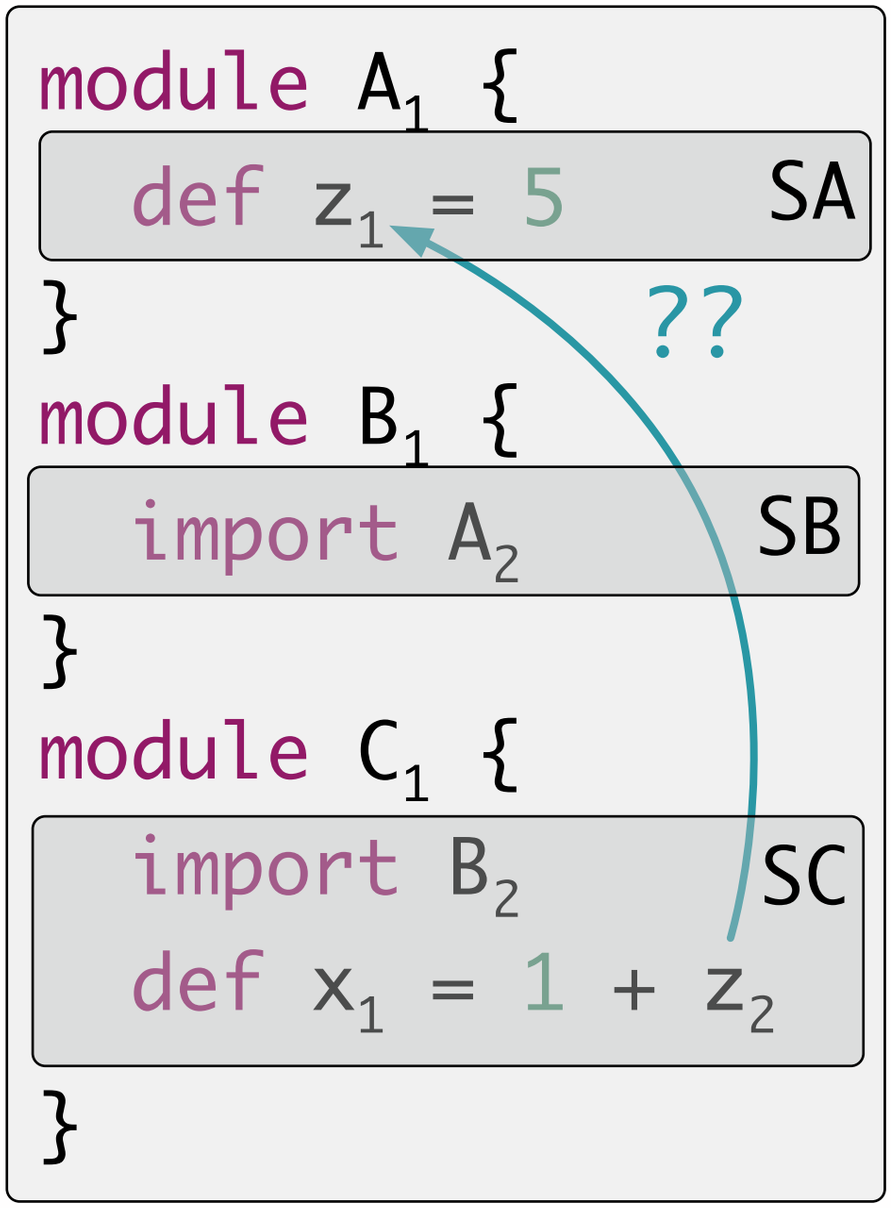
Transitive vs. Non-Transitive

```
module A1 {
  def z1 = 5                    SA
}
module B1 {
  import A2                     SB
}
module C1 {
  import B2                     SC
  def x1 = 1 + z2
}
```

Transitive vs. Non-Transitive

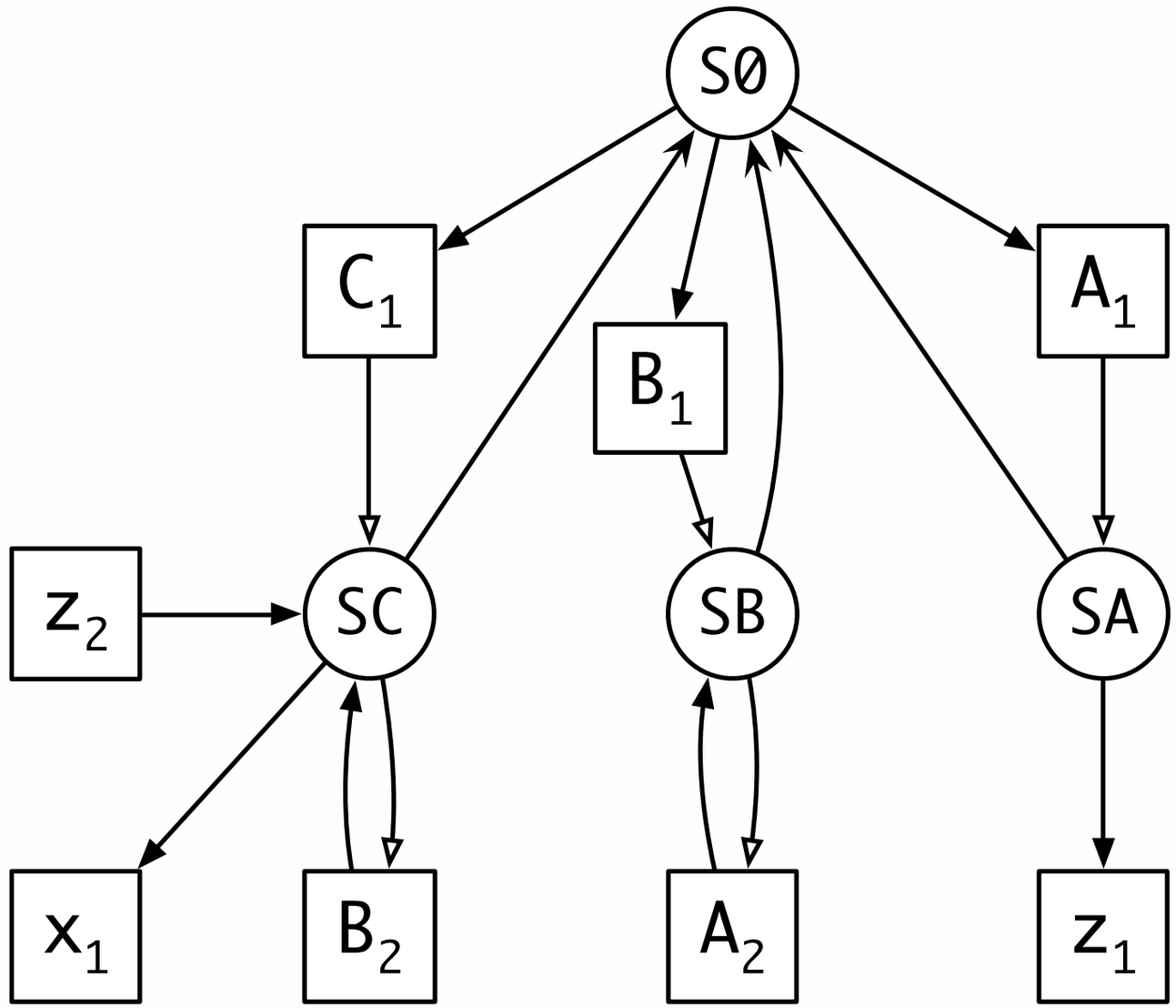


Transitive vs. Non-Transitive



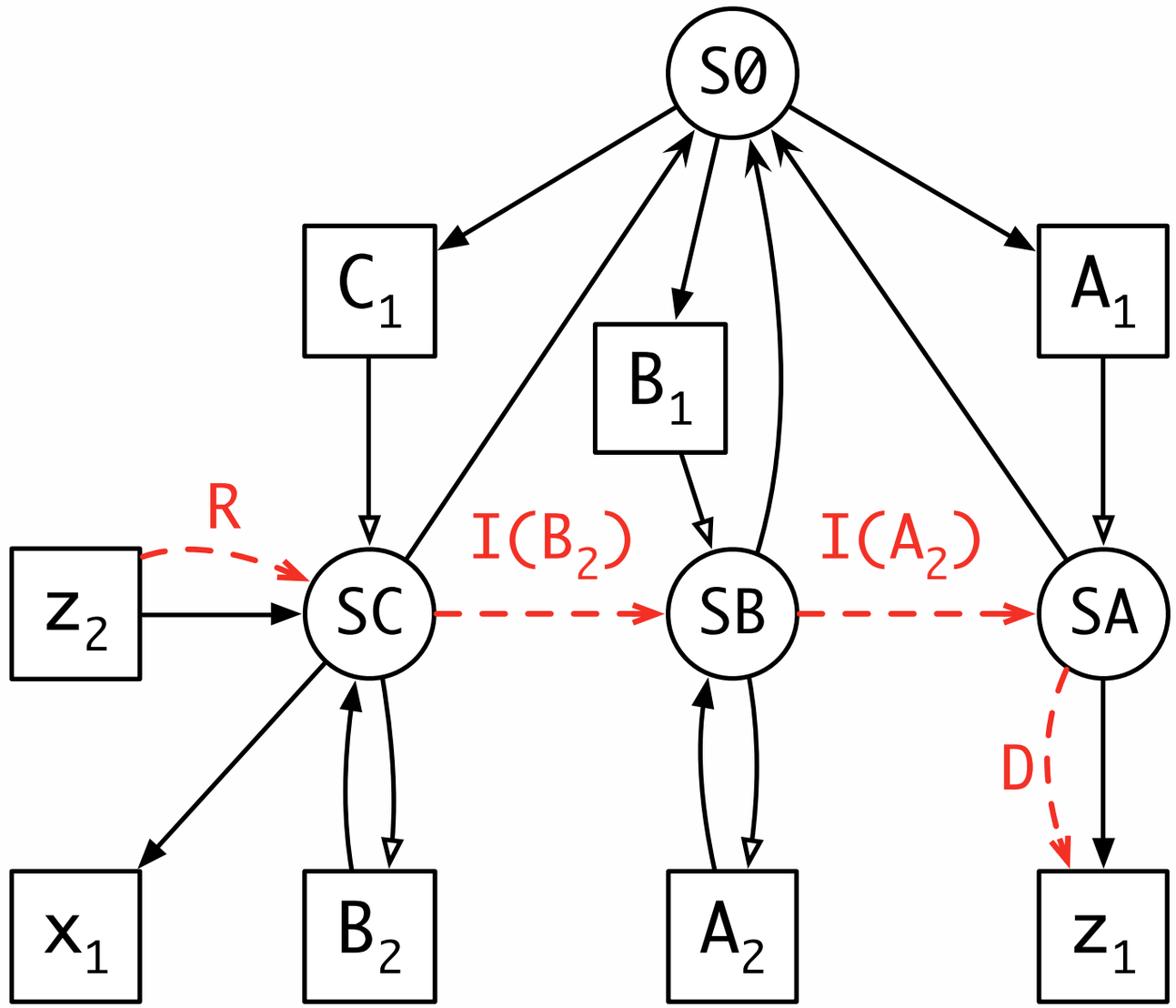
Transitive vs. Non-Transitive

```
module A1 {
  def z1 = 5                    SA
}                          ??
module B1 {
  import A2                     SB
}
module C1 {
  import B2                     SC
  def x1 = 1 + z2
}
```



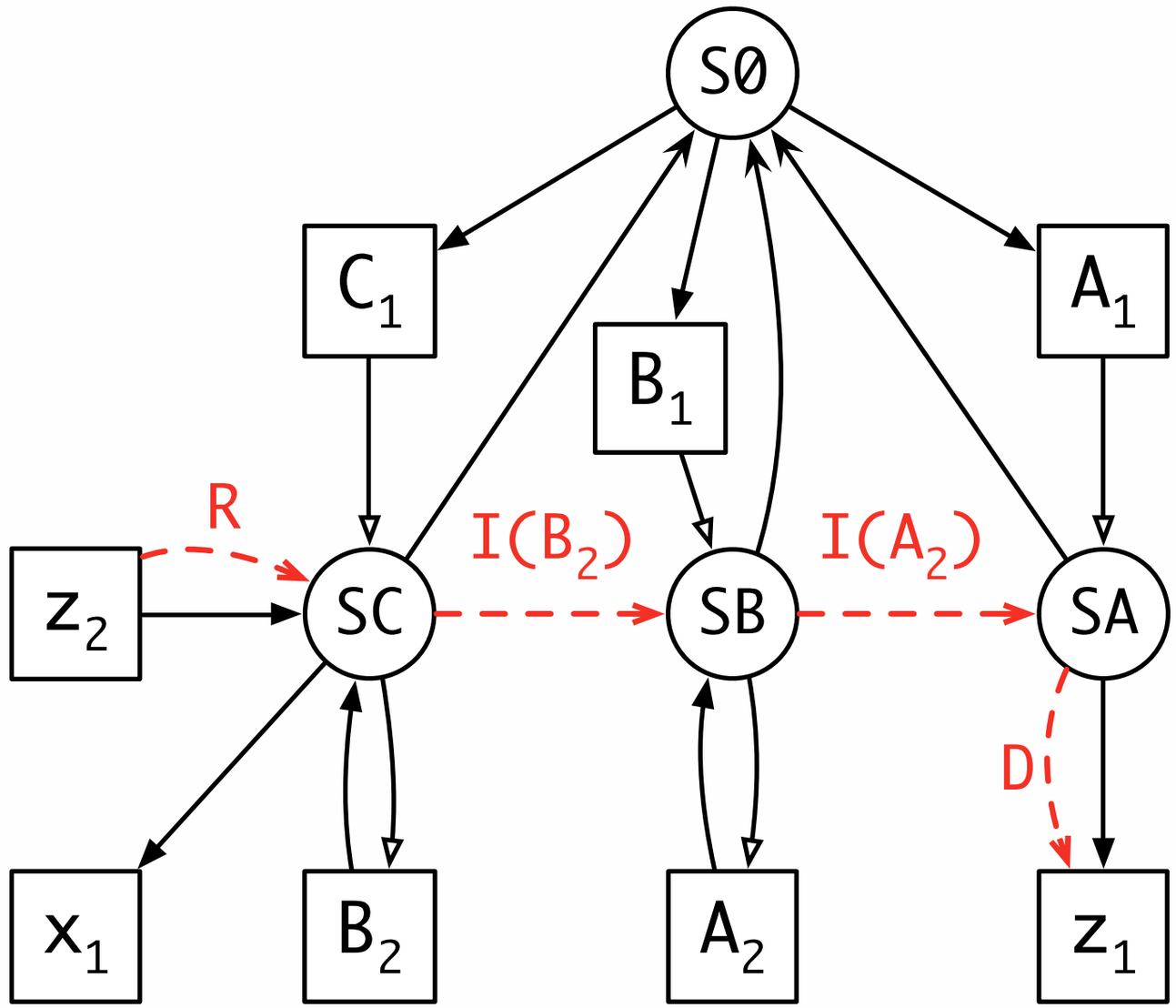
Transitive vs. Non-Transitive

```
module A1 {
  def z1 = 5                    SA
}                                 ??
module B1 {
  import A2                     SB
}
module C1 {
  import B2                     SC
  def x1 = 1 + z2
}
```



Transitive vs. Non-Transitive

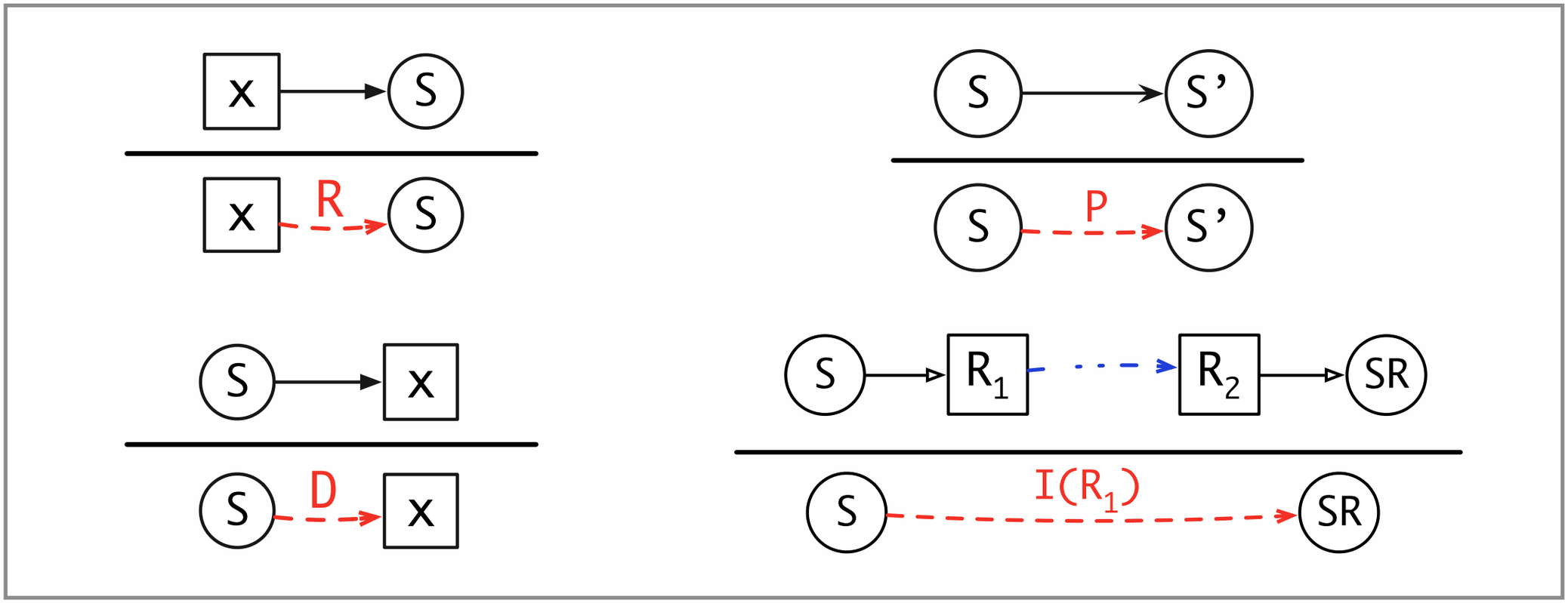
```
module A1 {
  def z1 = 5                    SA
}                          ??
module B1 {
  import A2                     SB
}
module C1 {
  import B2                     SC
  def x1 = 1 + z2
}
```



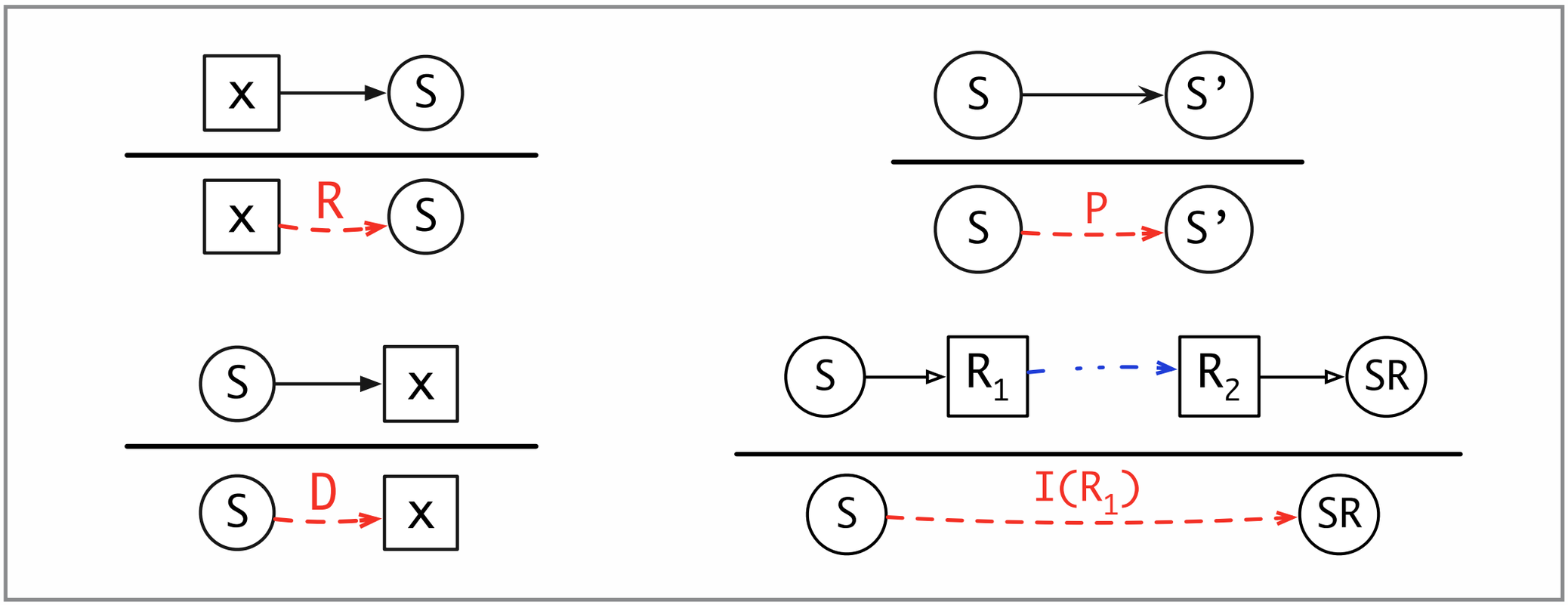
With transitive imports, a well formed path is `R.P*.I(_)*.D`

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

A Calculus for Name Resolution



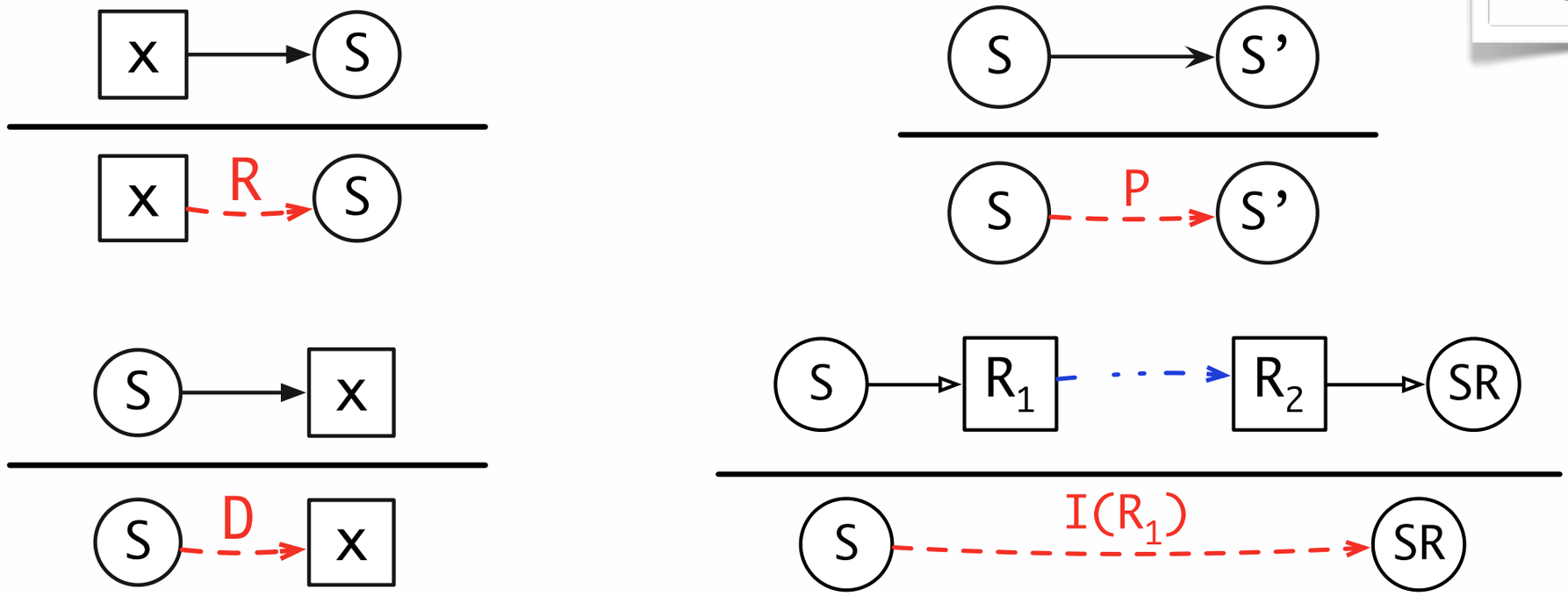
A Calculus for Name Resolution



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

A Calculus for Name Resolution

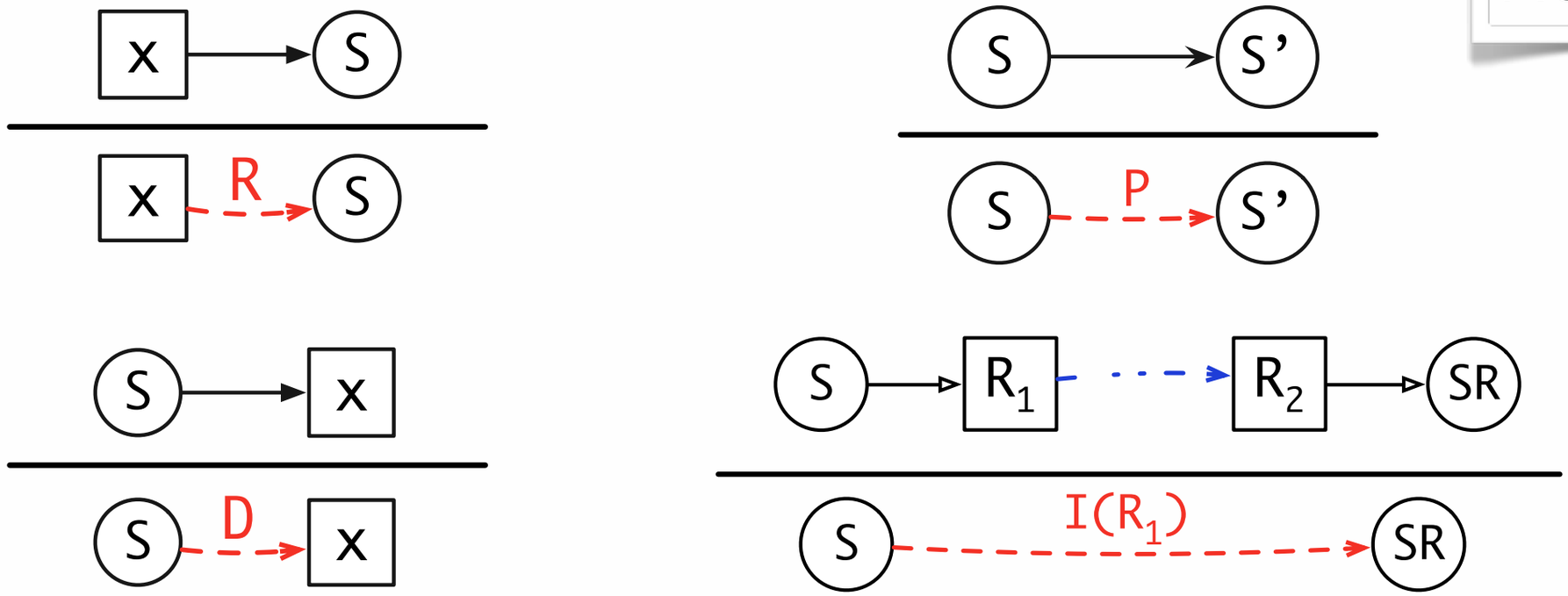
Reachability



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

A Calculus for Name Resolution

Reachability



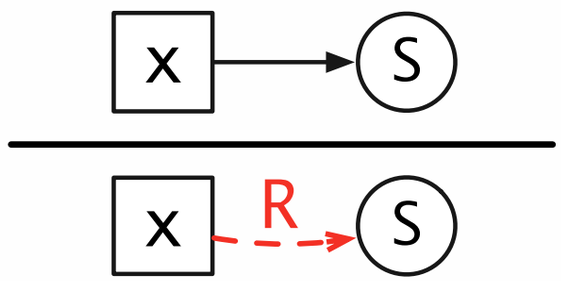
Well formed path: R.P*.I(_)*.D

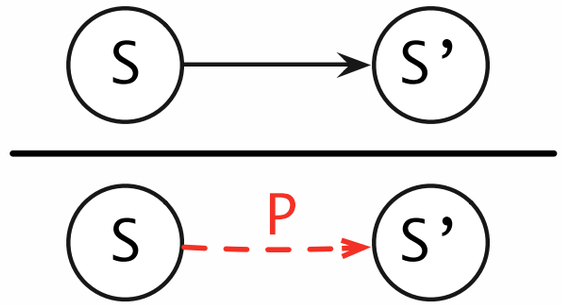
$$\frac{}{D < P.p} \qquad \frac{}{I(\_).p' < P.p}$$

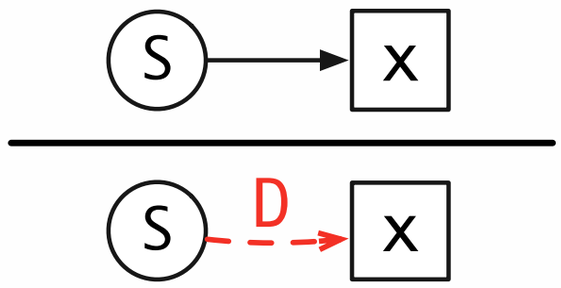
$$\frac{p < p'}{s.p < s.p'} \qquad \frac{}{D < I(\_).p'}$$

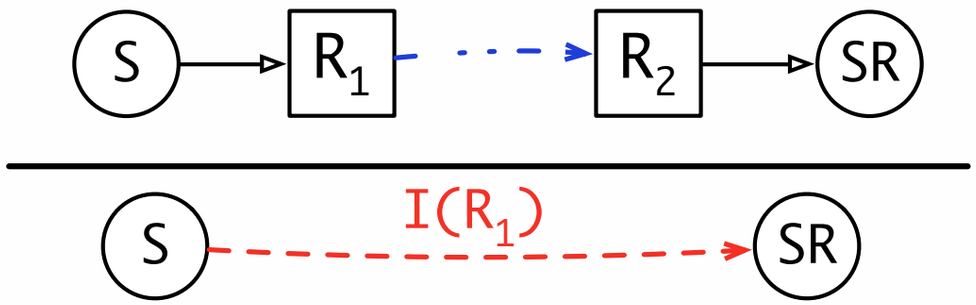
A Calculus for Name Resolution

Reachability









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

Visibility

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

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

Theory of Scope Graphs

Scope graphs, formally

$x_i^{\mathsf{D}}{:}S$ declaration with name x at position i with optional associated scope S

x_i^{R} reference with name x at position i

Scope graphs, formally

$x_i^{\mathsf{D}}{:}S$ declaration with name x at position i with optional associated scope S

x_i^{R} reference with name x at position i

$\mathcal{G}$: scope graph
$\mathcal{S}(\mathcal{G})$: scopes S in $\mathcal{G}$
$\mathcal{D}(S)$: declarations $x_i^{\mathsf{D}}{:}S'$ in S
$\mathcal{R}(S)$: references x_i^{R} in S
$\mathcal{I}(S)$: imports x_i^{R} in S
$\mathcal{P}(S)$: parent scope of S

Scope graphs, formally

$x_i^{\mathsf{D}}{:}S$ declaration with name x at position i with optional associated scope S

x_i^{R} reference with name x at position i

$\mathcal{G}$: scope graph
$\mathcal{S}(\mathcal{G})$: scopes S in $\mathcal{G}$
$\mathcal{D}(S)$: declarations $x_i^{\mathsf{D}}{:}S'$ in S
$\mathcal{R}(S)$: references x_i^{R} in S
$\mathcal{I}(S)$: imports x_i^{R} in S
$\mathcal{P}(S)$: parent scope of S

- $\mathcal{P}(S)$ is a partial function
- The parent relation is well-founded
- Each x_i^{R} and x_i^{D} appears in exactly one scope S

Resolution calculus, formally

Resolution paths

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

Resolution calculus, formally

Resolution paths

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

Resolution calculus, formally

Resolution paths

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

Visibility paths

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p' (\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}}$$

Resolution calculus, formally

Resolution paths

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

Visibility paths

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p' (\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}}$$

Reachability paths

$$\frac{\vdots}{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}}}$$

Resolution calculus, formally

Resolution paths

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

Visibility paths

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p'(\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}}$$

Reachability paths

$$\frac{\vdots}{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}}}$$

...may include import steps

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

Resolution calculus, formally

Resolution paths

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

Visibility paths

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p'(\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}}$$

Reachability paths

$$\frac{\vdots}{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}}}$$

Well-founded!

...may include import steps

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

Resolution Algorithm

Resolution Algorithm

- Resolution calculus says how to validate paths but not how to find them
  - and reachability paths can have cycles

Resolution Algorithm

- Resolution calculus says how to validate paths but not how to find them
  - and reachability paths can have cycles
- But there is a terminating resolution algorithm
  - for path well-foundedness `RP*I*D`
  - and path ordering `D<I<P`

Resolution Algorithm

- Resolution calculus says how to validate paths but not how to find them
  - and reachability paths can have cycles
- But there is a terminating resolution algorithm
  - for path well-foundedness `RP*I*D`
  - and path ordering `D<I<P`
- Uses familiar notions of environments and shadowing

Resolution Algorithm v1

$$E_D(S) := \mathcal{D}(S)$$

$$E_P(S) := E_V(\mathcal{P}(S))$$

$$E_I(S) := \bigcup \left\{ E_L(S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \wedge y_j^{\mathsf{D}} {:} S_y \in Resolve(y_i^{\mathsf{R}}) \right\}$$

$$E_L(S) := E_D(S) \triangleleft E_I(S)$$

$$E_V(S) := E_L(S) \triangleleft E_P(S)$$

$$Resolve(x_i^{\mathsf{R}}) := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V(S)\}$$

where $E_1 \triangleleft E_2 := E_1 \cup \{x_i^{\mathsf{D}} \in E_2 \mid \nexists \ x_{i'}^{\mathsf{D}} \in E_1\}$

Resolution Algorithm v1

$$E_D(S) := \mathcal{D}(S)$$

$$E_P(S) := E_V(\mathcal{P}(S))$$

$$E_I(S) := \bigcup \left\{ E_L(S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \wedge y_j^{\mathsf{D}} {:} S_y \in Resolve(y_i^{\mathsf{R}}) \right\}$$

$$E_L(S) := E_D(S) \triangleleft E_I(S)$$

$$E_V(S) := E_L(S) \triangleleft E_P(S)$$

$$Resolve(x_i^{\mathsf{R}}) := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V(S)\}$$

where $E_1 \triangleleft E_2 := E_1 \cup \{x_i^{\mathsf{D}} \in E_2 \mid \nexists \ x_{i'}^{\mathsf{D}} \in E_1\}$

- still need to incorporate “seen imports”

Resolution Algorithm v2

$$E_D[\mathbb{I}](S) := \mathcal{D}(S)$$

$$E_P[\mathbb{I}](S) := E_V[\mathbb{I}](\mathcal{P}(S))$$

$$E_I[\mathbb{I}](S) := \bigcup \left\{ E_L[\mathbb{I}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}} {:} S_y \in \mathit{Resolve}[\mathbb{I}](y_i^{\mathsf{R}}) \right\}$$

$$E_L[\mathbb{I}](S) := E_D[\mathbb{I}](S) \triangleleft E_I[\mathbb{I}](S)$$

$$E_V[\mathbb{I}](S) := E_L[\mathbb{I}](S) \triangleleft E_P[\mathbb{I}](S)$$

$$\mathit{Resolve}[\mathbb{I}](x_i^{\mathsf{R}}) := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}](S)\}$$

where $E_1 \triangleleft E_2 := E_1 \cup \{x_i^{\mathsf{D}} \in E_2 \mid \nexists \ x_{i'}^{\mathsf{D}} \in E_1\}$

Resolution Algorithm v2

$$
\begin{aligned}
E_D[\mathbb{I}](S) &:= \mathcal{D}(S) \\
E_P[\mathbb{I}](S) &:= E_V[\mathbb{I}](\mathcal{P}(S)) \\
E_I[\mathbb{I}](S) &:= \bigcup \left\{ E_L[\mathbb{I}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}} {:} S_y \in \mathit{Resolve}[\mathbb{I}](y_i^{\mathsf{R}}) \right\} \\
E_L[\mathbb{I}](S) &:= E_D[\mathbb{I}](S) \triangleleft E_I[\mathbb{I}](S) \\
E_V[\mathbb{I}](S) &:= E_L[\mathbb{I}](S) \triangleleft E_P[\mathbb{I}](S)
\end{aligned}
$$

$$
\mathit{Resolve}[\mathbb{I}](x_i^{\mathsf{R}}) := \{ x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}](S) \}
$$

where $E_1 \triangleleft E_2 := E_1 \cup \{ x_i^{\mathsf{D}} \in E_2 \mid \nexists \ x_{i'}^{\mathsf{D}} \in E_1 \}$

- but still might not terminate due to cycles (e.g. consider a scope that imports itself)

Resolution Algorithm v3

$$E_D[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \mathcal{D}(S) \end{cases}$$

$$E_P[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ E_V[\mathbb{I},\{S\} \cup \mathbb{S}](\mathcal{P}(S)) \end{cases}$$

$$E_I[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \bigcup \left\{ E_L[\mathbb{I},\{S\} \cup \mathbb{S}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}} {:} S_y \in \mathit{Resolve}[\mathbb{I}](y_i^{\mathsf{R}}) \right\} \end{cases}$$

$$E_L[\mathbb{I},\mathbb{S}](S) \quad := E_D[\mathbb{I},\mathbb{S}](S) \triangleleft E_I[\mathbb{I},\mathbb{S}](S)$$

$$E_V[\mathbb{I},\mathbb{S}](S) \quad := E_L[\mathbb{I},\mathbb{S}](S) \triangleleft E_P[\mathbb{I},\mathbb{S}](S)$$

$$\mathit{Resolve}[\mathbb{I}](x_i^{\mathsf{R}}) := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](S)\}$$

Resolution Algorithm v3

$$E_D[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \mathcal{D}(S) \end{cases}$$

$$E_P[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ E_V[\mathbb{I}, \{S\} \cup \mathbb{S}](\mathcal{P}(S)) \end{cases}$$

$$E_I[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \bigcup \left\{ E_L[\mathbb{I}, \{S\} \cup \mathbb{S}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}} {:} S_y \in \mathit{Resolve}[\mathbb{I}](y_i^{\mathsf{R}}) \right\} \end{cases}$$

$$E_L[\mathbb{I},\mathbb{S}](S) \quad := E_D[\mathbb{I},\mathbb{S}](S) \triangleleft E_I[\mathbb{I},\mathbb{S}](S)$$

$$E_V[\mathbb{I},\mathbb{S}](S) \quad := E_L[\mathbb{I},\mathbb{S}](S) \triangleleft E_P[\mathbb{I},\mathbb{S}](S)$$

$$\mathit{Resolve}[\mathbb{I}](x_i^{\mathsf{R}}) := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](S)\}$$

Lemma: visibility paths never have cycles

Resolution Algorithm v3

$$E_D[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \mathcal{D}(S) \end{cases}$$

$$E_P[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ E_V[\mathbb{I}, \{S\} \cup \mathbb{S}](\mathcal{P}(S)) \end{cases}$$

$$E_I[\mathbb{I},\mathbb{S}](S) \quad := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \bigcup \left\{ E_L[\mathbb{I}, \{S\} \cup \mathbb{S}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}} {:} S_y \in Resolve[\mathbb{I}](y_i^{\mathsf{R}}) \right\} \end{cases}$$

$$E_L[\mathbb{I},\mathbb{S}](S) \quad := E_D[\mathbb{I},\mathbb{S}](S) \triangleleft E_I[\mathbb{I},\mathbb{S}](S)$$

$$E_V[\mathbb{I},\mathbb{S}](S) \quad := E_L[\mathbb{I},\mathbb{S}](S) \triangleleft E_P[\mathbb{I},\mathbb{S}](S)$$

$$Resolve[\mathbb{I}](x_i^{\mathsf{R}}) := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in E_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](S)\}$$

Lemma: visibility paths never have cycles

Theorem: algorithm is sound and complete for calculus:

$$(x_j^{\mathsf{D}} \in Res[\mathbb{I}](x_i^{\mathsf{R}})) \iff (\exists p \ s.t. \ \mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}})$$

Language-independent α-equivalence

Program similarity

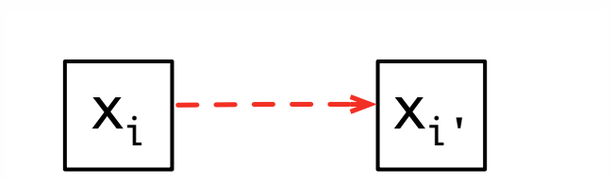
`P` $\simeq$ `P'` if have same AST ignoring identifier names

Language-independent α-equivalence

Program similarity

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

Position equivalence



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

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

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

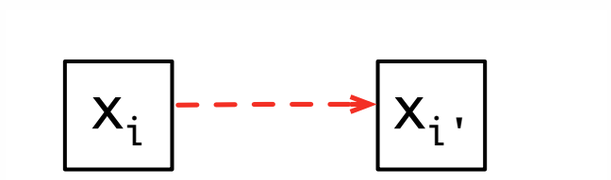
$$\frac{}{i \overset{\texttt{P}}{\sim} i}$$

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

$P1 \overset{\alpha}{\approx} P2$

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

Applying Scope Graphs

(ongoing work)

Validation

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
  - …

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
  - Java
  - ML
  - …

Generating Scope Graphs from AST

$$
\begin{array}{rcll}
[\![ds]\!]^{prog} & := & P(S) := \bot \ \wedge \ [\![ds]\!]^{decl^*}_{S} & \textit{(new } S\textit{)} \\
[\![\texttt{module}\ X_i\ \{\, ds \,\}]\!]^{decl}_{s} & := & X_i^{\mathsf{D}}{:}S' \in \mathcal{D}(s) \ \wedge \ P(S') := s \ \wedge \ [\![ds]\!]^{decl^*}_{S'} & \textit{(new } S'\textit{)} \\
[\![\texttt{import}\ Xs\,\texttt{.}\,X_i]\!]^{decl}_{s} & := & X_i^{\mathsf{R}} \in \mathcal{I}(s) \ \wedge \ [\![Xs\,\texttt{.}\,X_i]\!]^{qid}_{s} & \\
[\![\texttt{def}\ b]\!]^{decl}_{s} & := & [\![b]\!]^{bind}_{s,s} & \\
[\![x_i = e]\!]^{bind}_{s_r,s_d} & := & x_i^{\mathsf{D}} \in \mathcal{D}(s_d) \ \wedge \ [\![e]\!]^{exp}_{s_r} & \\
[\![x_i]\!]^{qid}_{s} & := & x_i^{\mathsf{R}} \in \mathcal{R}(s) & \\
[\![\texttt{fun}\ (x_i\texttt{:}t)\ \{e\}]\!]^{exp}_{s} & := & P(S') := s \ \wedge \ x_i^{\mathsf{D}} \in \mathcal{D}(S') \ \wedge \ [\![e]\!]^{exp}_{S'} & \textit{(new } S'\textit{)} \\
[\![\texttt{letrec}\ bs\ \texttt{in}\ e]\!]^{exp}_{s} & := & P(S') := s \ \wedge \ [\![bs]\!]^{bind^*}_{S',S'} \ \wedge \ [\![e]\!]^{exp}_{S'} & \textit{(new } S'\textit{)} \\
[\![\texttt{letpar}\ bs\ \texttt{in}\ e]\!]^{exp}_{s} & := & P(S') := s \ \wedge \ [\![bs]\!]^{bind^*}_{S,S'} \ \wedge \ [\![e]\!]^{exp}_{S'} & \textit{(new } S'\textit{)} \\
[\![Xs\,\texttt{.}\,x_i]\!]^{exp}_{s} & := & [\![Xs\,\texttt{.}\,x_i]\!]^{qid}_{s} & \\
[\![e_1\ e_2]\!]^{exp}_{s} & := & [\![e_1]\!]^{exp}_{s} \ \wedge \ [\![e_2]\!]^{exp}_{s} &
\end{array}
$$

generate smallest graph satisfying constraints

Binding gives Types

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

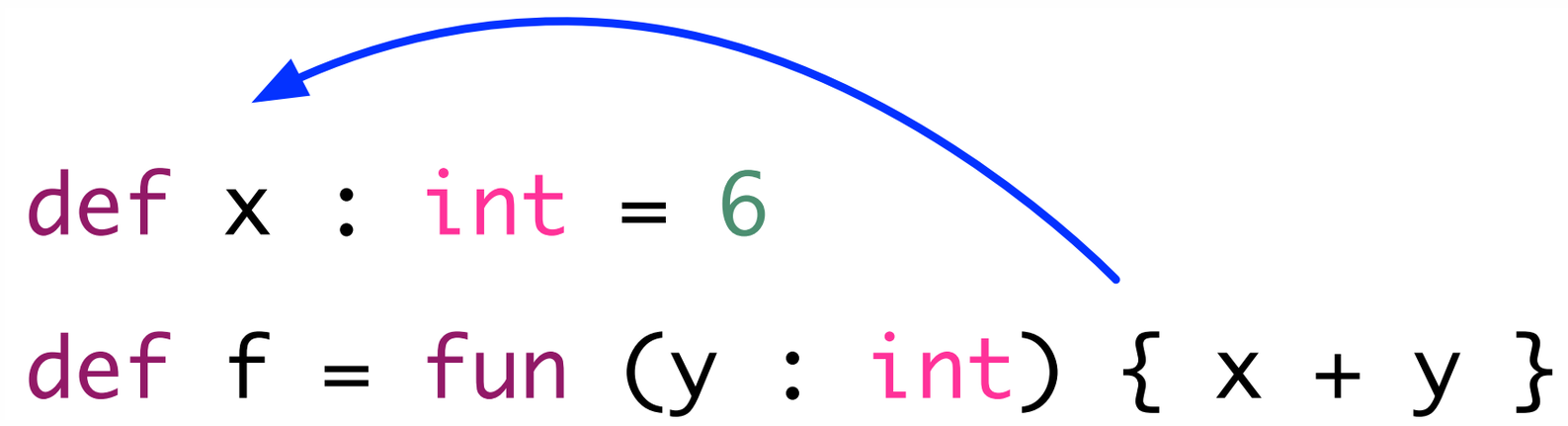
In many cases, we can perform resolution **before** doing type analysis

```
def x : int = 6

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

Binding gives Types

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

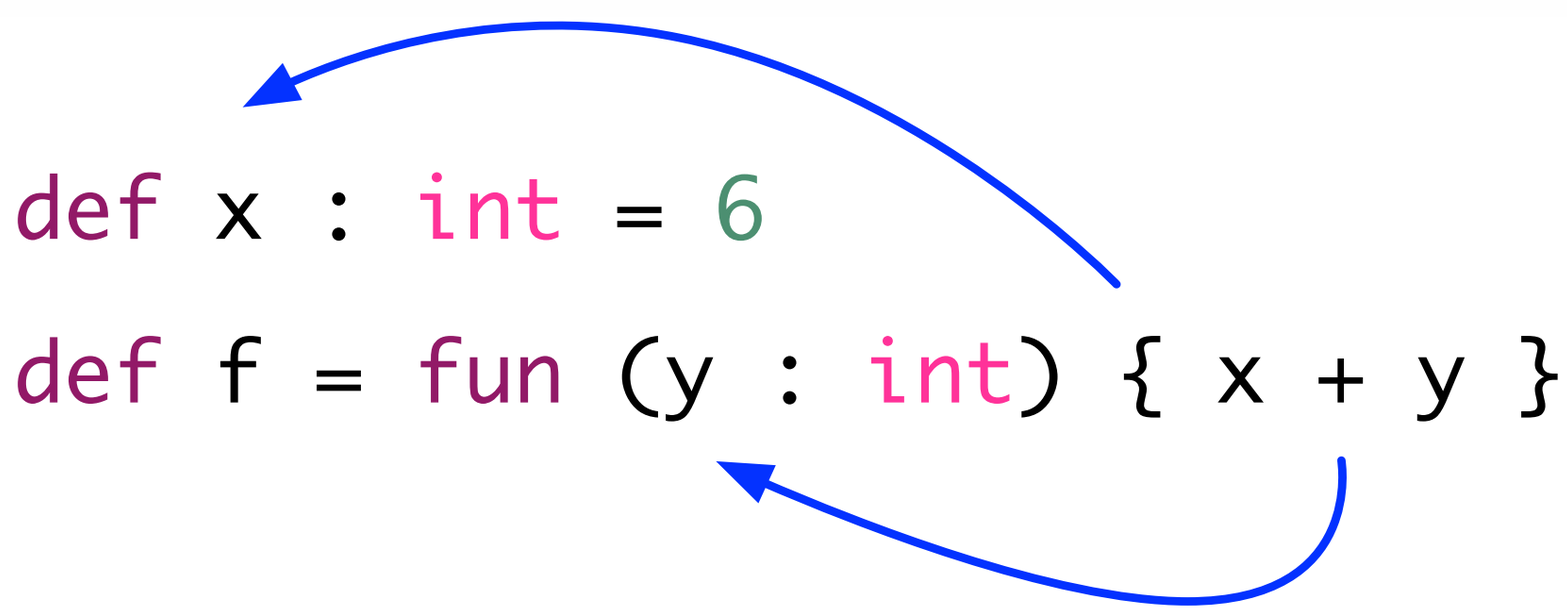
In many cases, we can perform resolution **before** doing type analysis



Binding gives Types

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

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



Types give Binding

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
```

Types give Binding

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
```

Types give Binding

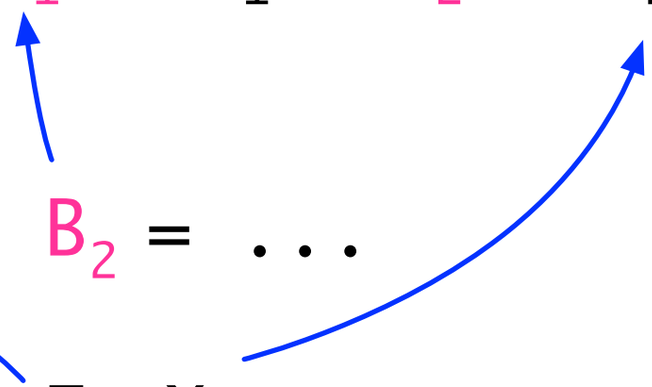
But sometimes we need types **before** we can do name resolution

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

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```

Types give Binding

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
```



Types give Binding

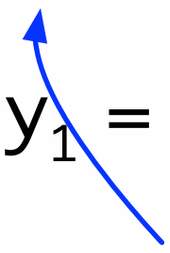
But sometimes we need types **before** we can do name resolution

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

def z₁ : B₂ = ...

def y₁ = z₂.x₃

def y₂ = z₃.a₂.x₄
```

Types give Binding

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

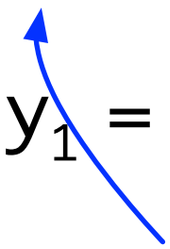
```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



Types give Binding

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

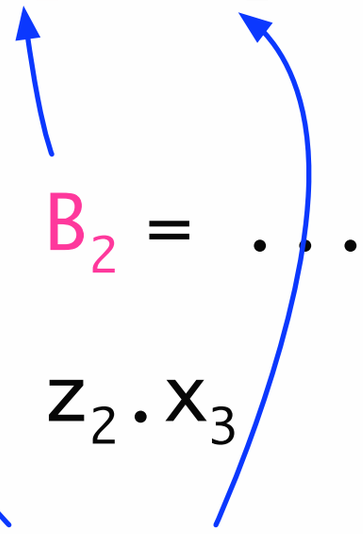
```
record A1 { x1 : int }
record B1 { a1 : A2 ; x2 : bool}

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



Types give Binding

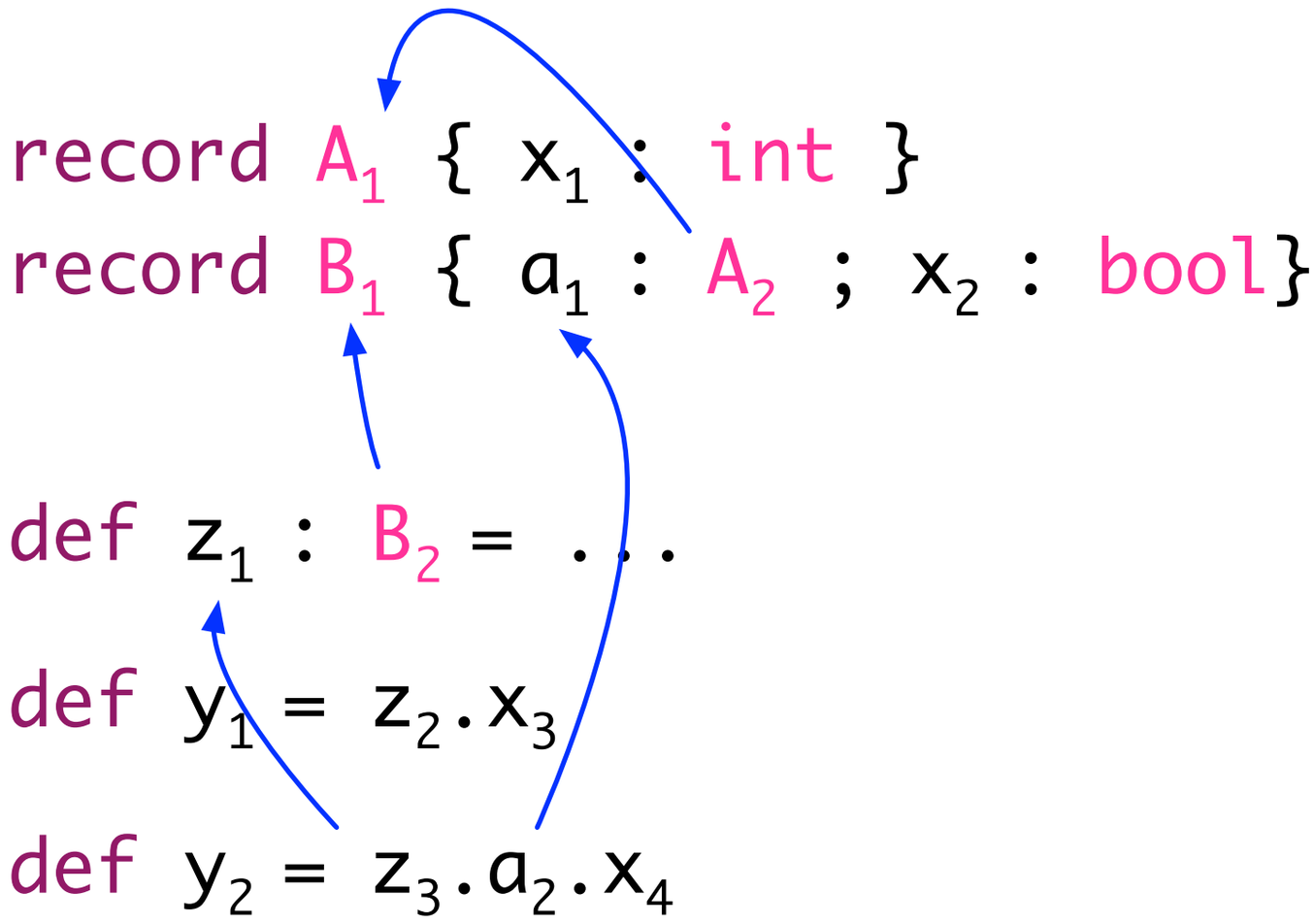
But sometimes we need types **before** we can do name resolution

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

def z1 : B2 = ...

def y1 = z2.x3

def y2 = z3.a2.x4
```



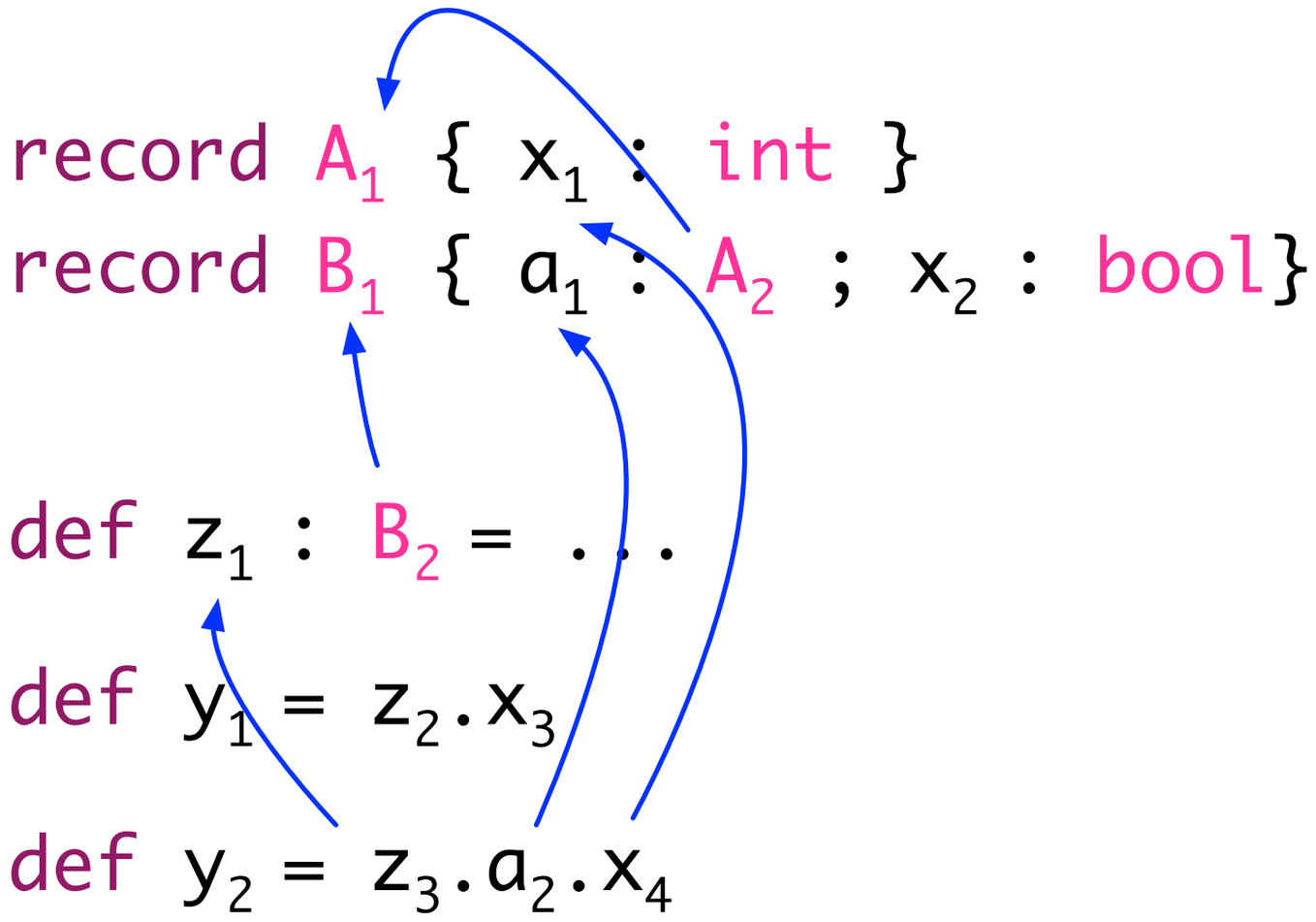
Types give Binding

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



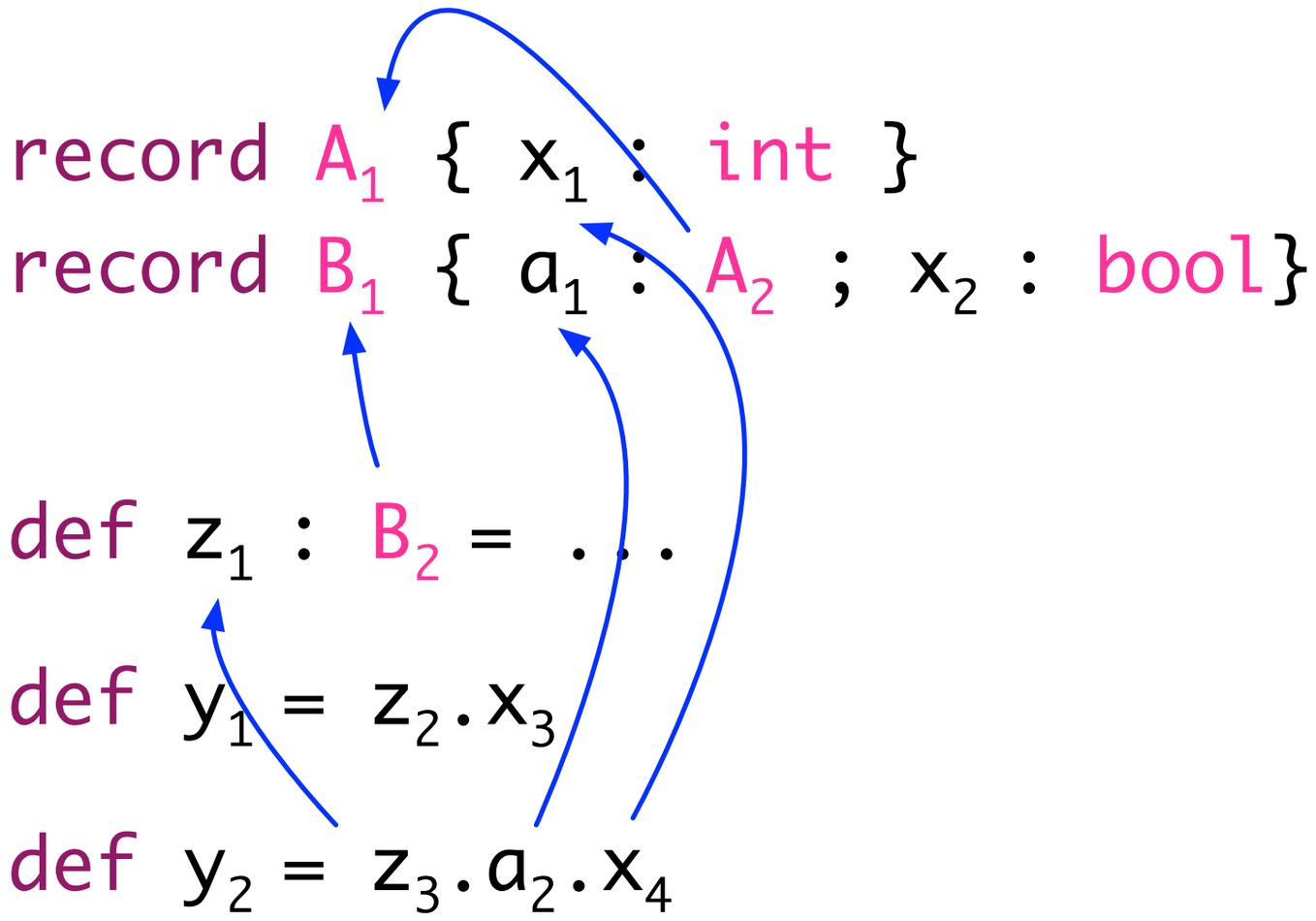
Types give Binding

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



Types give Binding

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



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

Also in the works...

Also in the works...

- Scope graph semantics for binding specification languages (starting with NaBL)

Also in the works...

- Scope graph semantics for binding specification languages (starting with NaBL)
- Resolution-sensitive program transformations (e.g. renaming, refactoring)

Also in the works...

- Scope graph semantics for binding specification languages (starting with NaBL)
- Resolution-sensitive program transformations (e.g. renaming, refactoring)
- Dynamic analogs to static scope graphs

Also in the works...

- Scope graph semantics for binding specification languages (starting with NaBL)
- Resolution-sensitive program transformations (e.g. renaming, refactoring)
- Dynamic analogs to static scope graphs
- Supporting mechanized language meta-theory

Use Scope Graphs to Describe Name Resolution

Use Scope Graphs to Describe Name Resolution !!