A Theory of Name Resolution

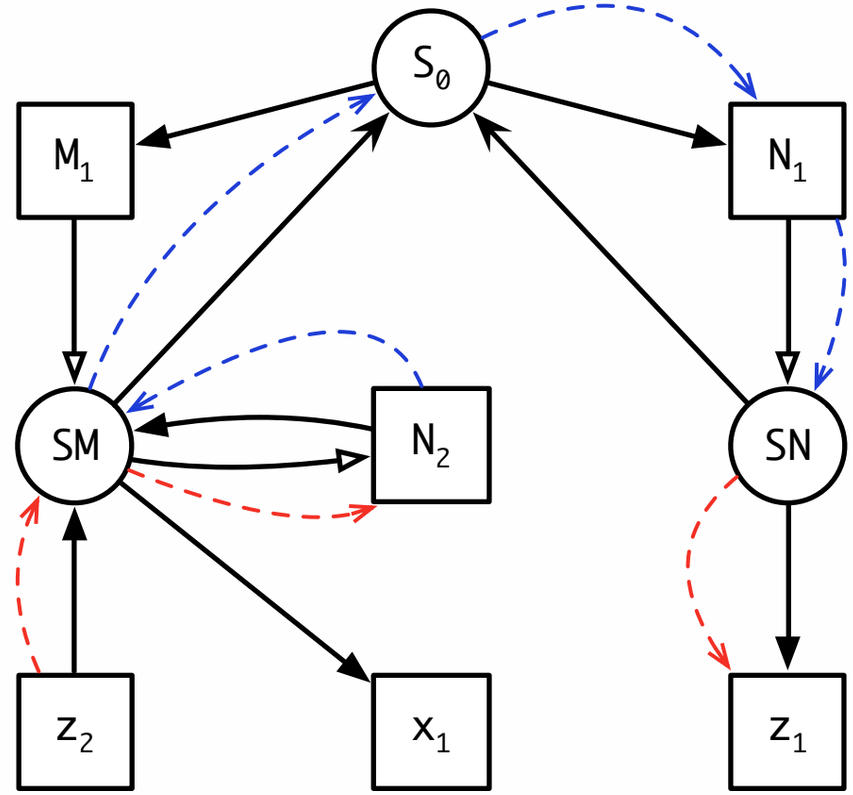
[ESOP15]

Pierre Neron[1]

Andrew Tolmach[2]

Eelco Visser[1]

Guido Wachsmuth[1]



1 TU Delft Delft University of Technology

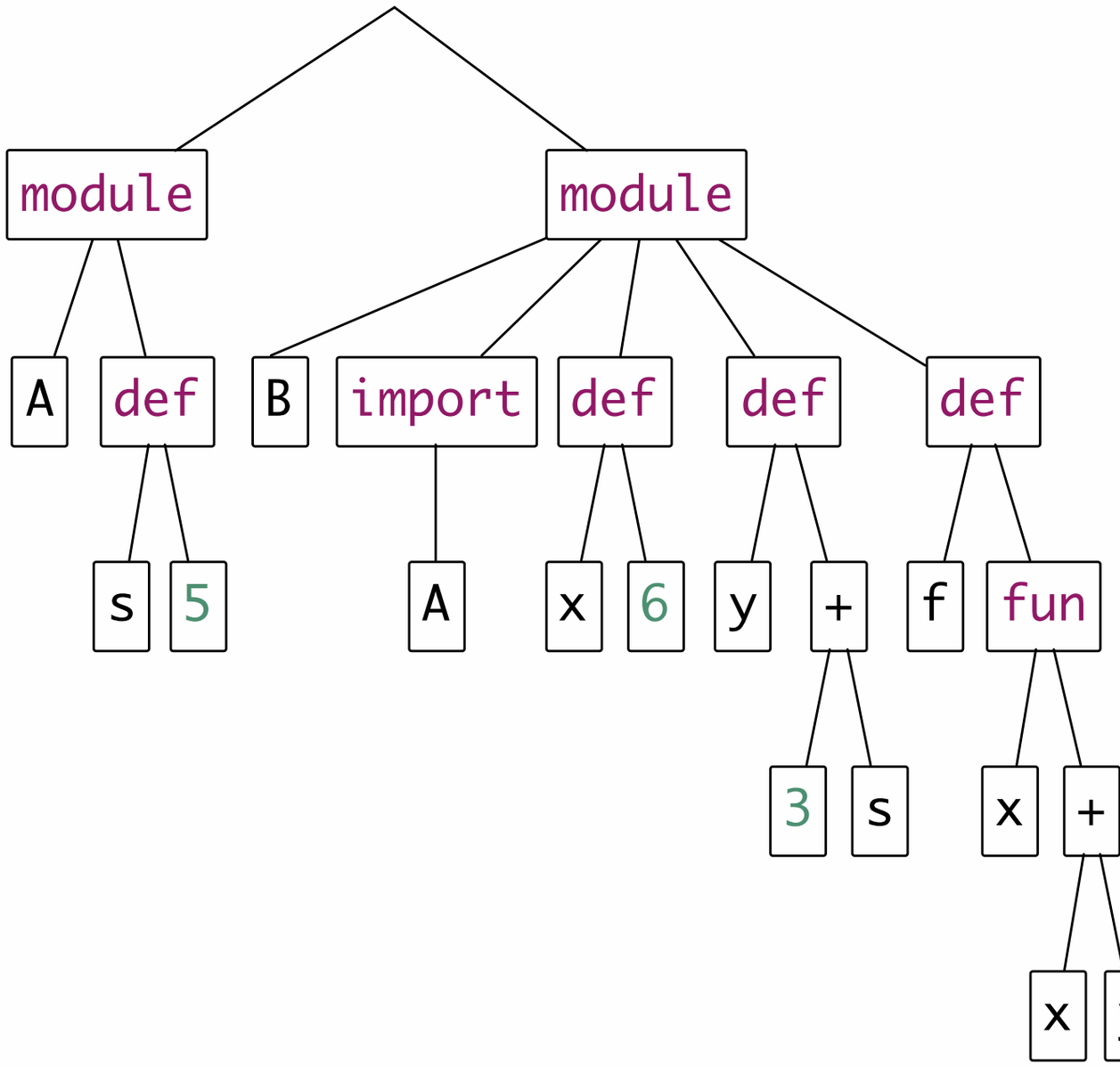
2 Portland State University

Name Resolution?

```
module A {
    def s = 5
}

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

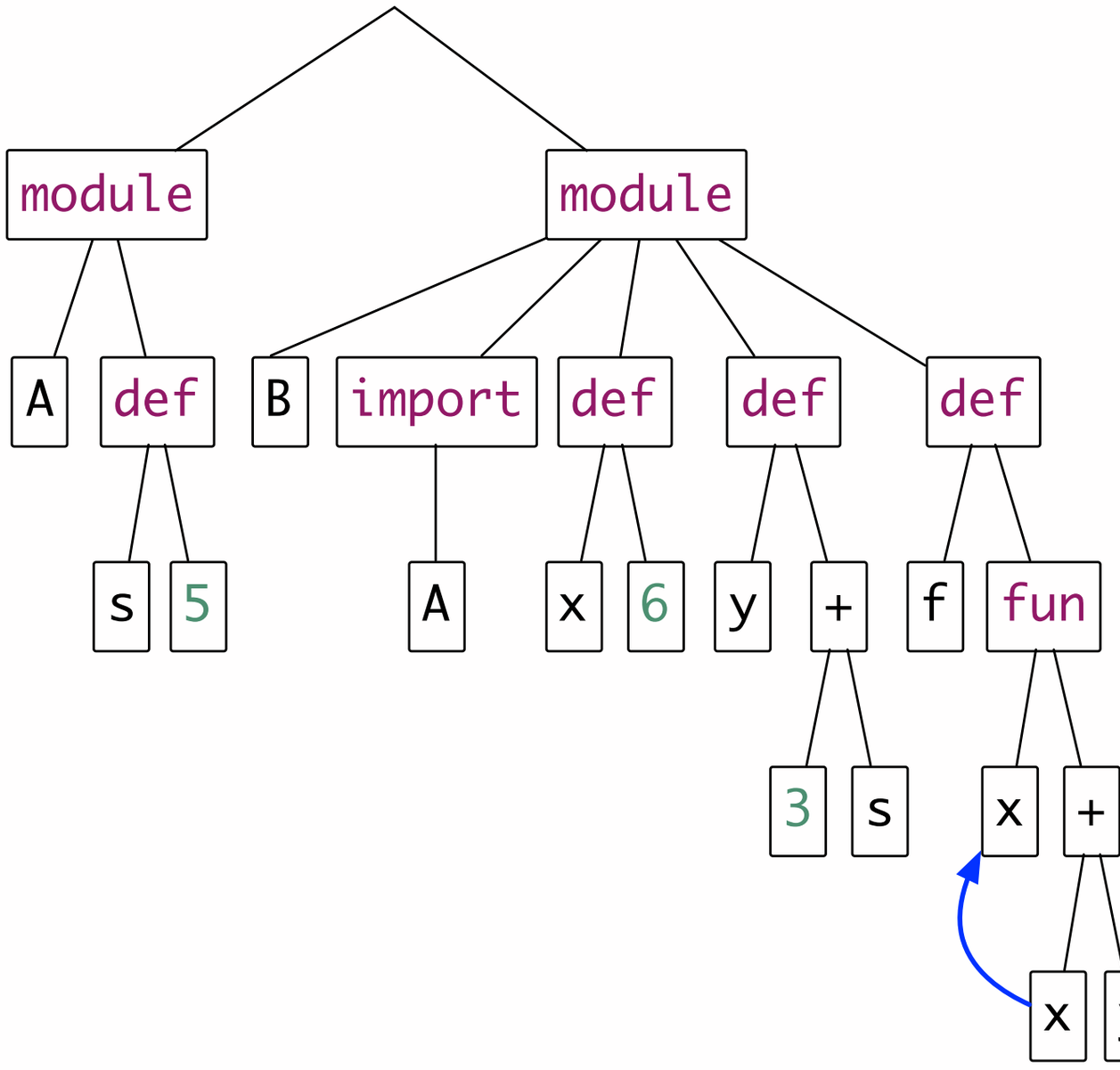
Name Resolution?

```
module A {
    def s = 5
}

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



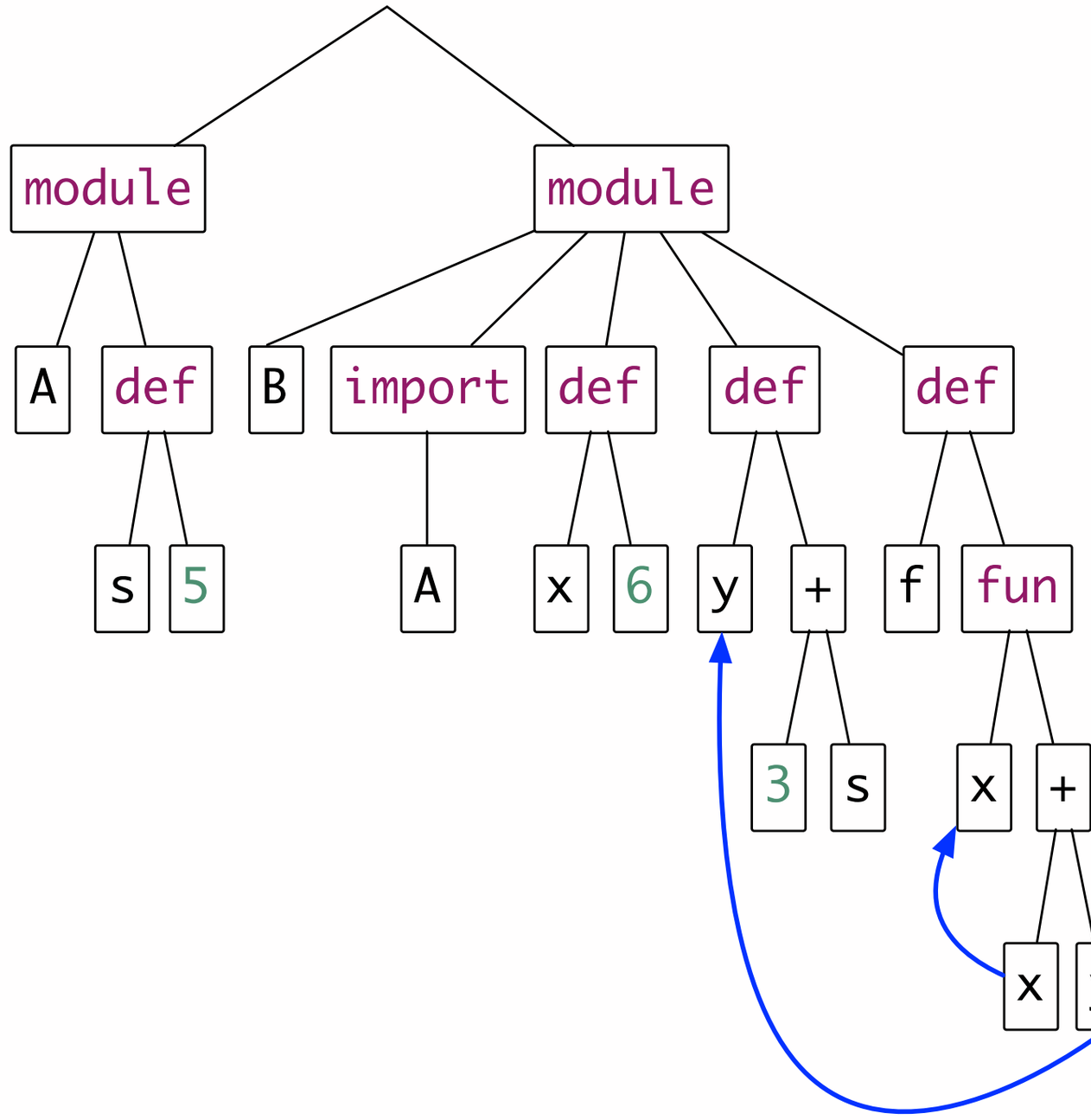
Name Resolution?

```
module A {
    def s = 5
}

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



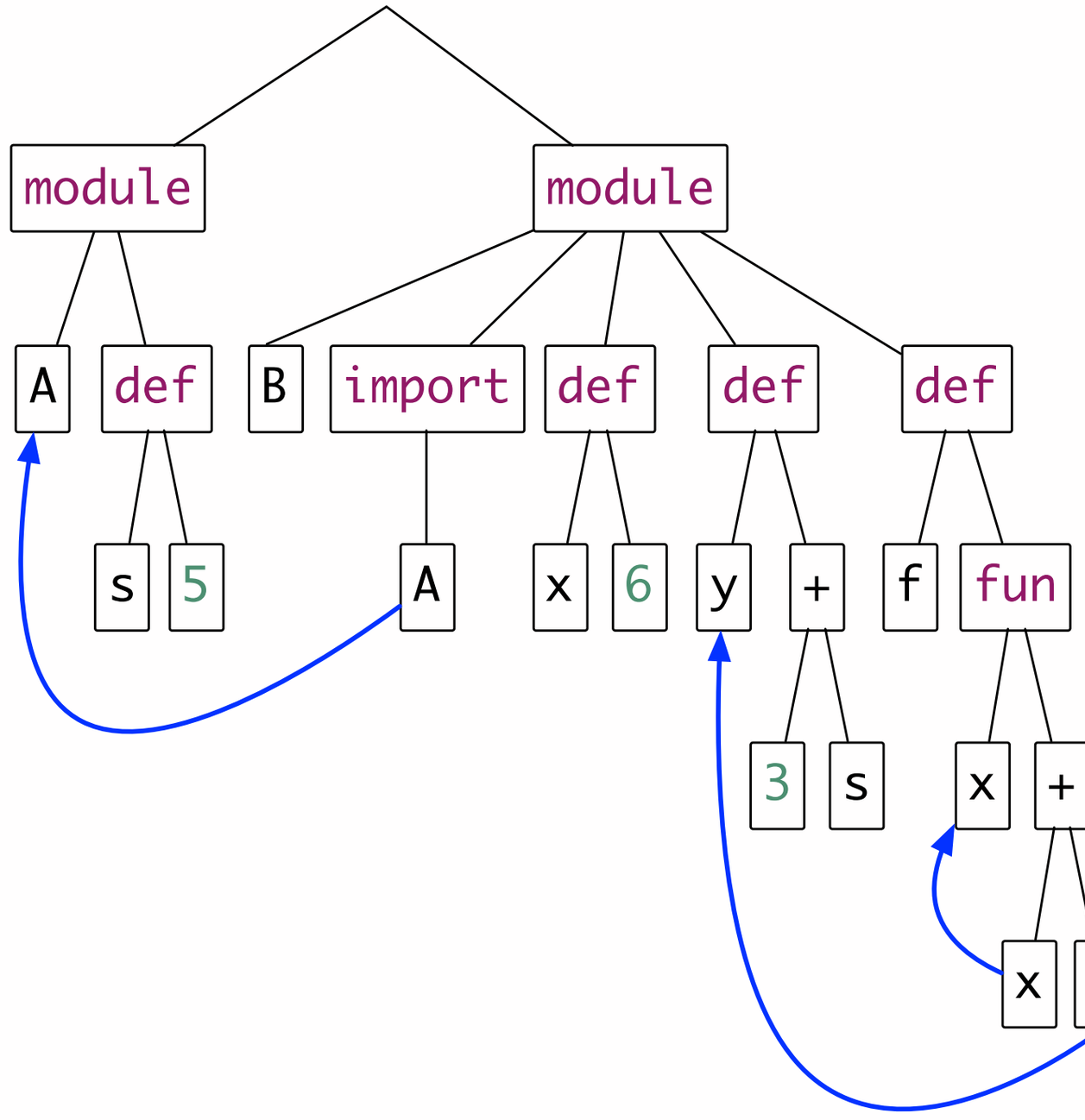
Name Resolution?

```
module A {
    def s = 5
}

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



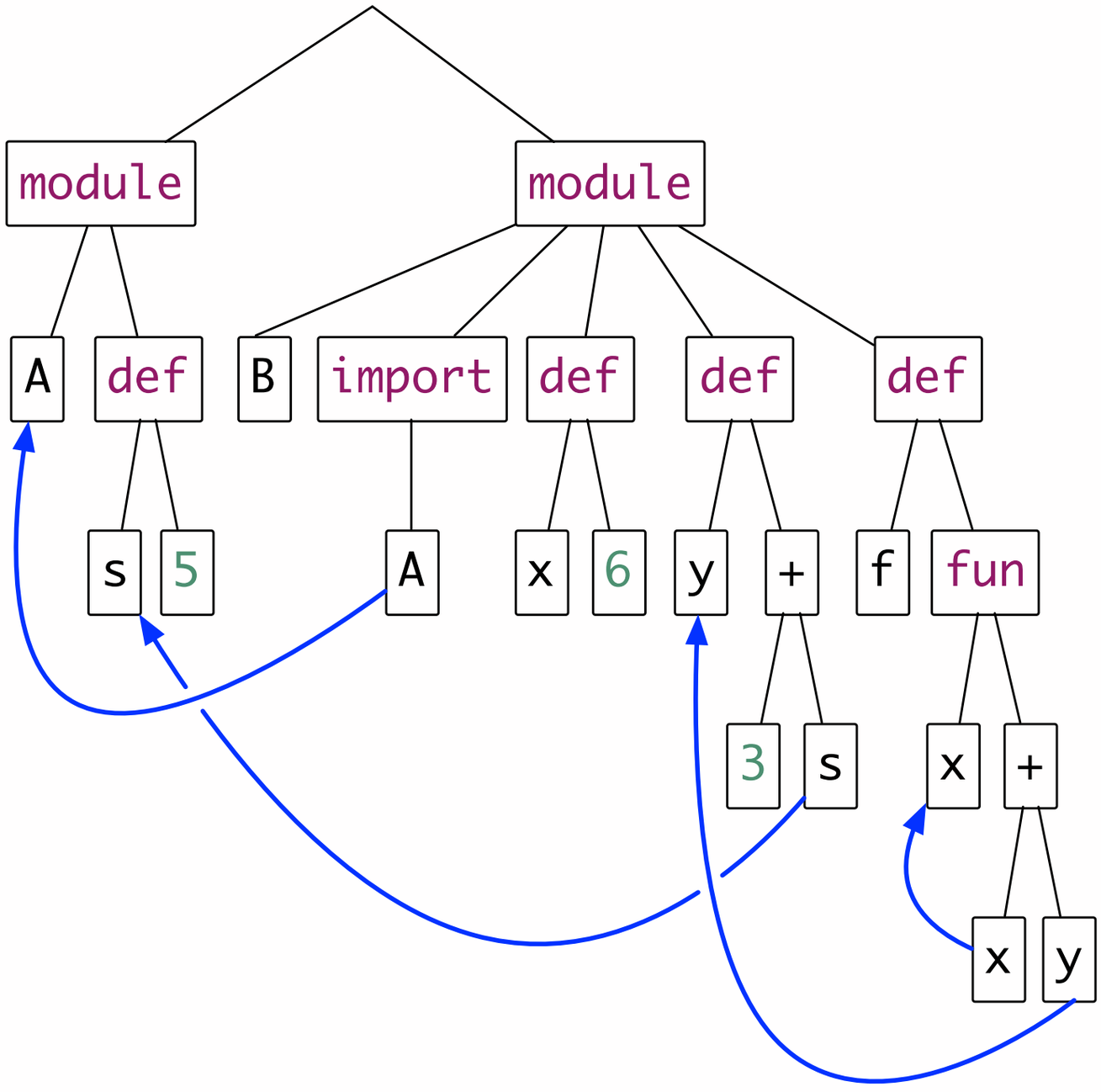
Name Resolution?

```
module A {
    def s = 5
}

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



Name Resolution?

```
module A {
    def s = 5
}

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



Name Resolution is Pervasive

Appears in many different artifacts…

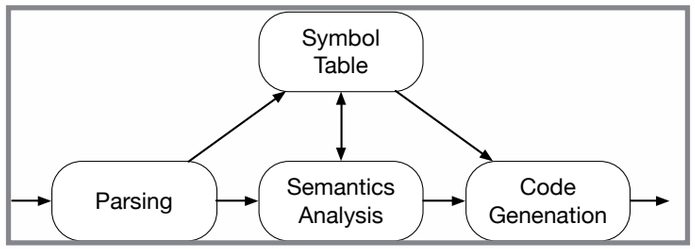
Name Resolution is Pervasive

Appears in many different artifacts…

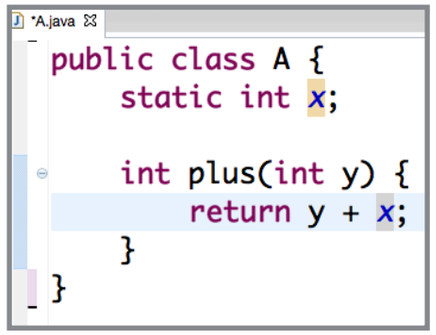
$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

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

Semantics



Compiler



IDE

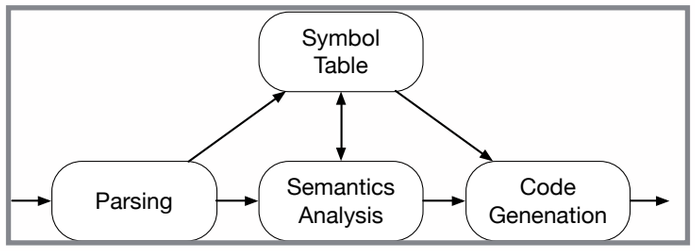
Name Resolution is Pervasive

Appears in many different artifacts…

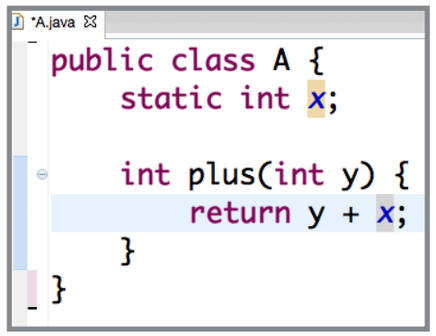
$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

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

Semantics



Compiler

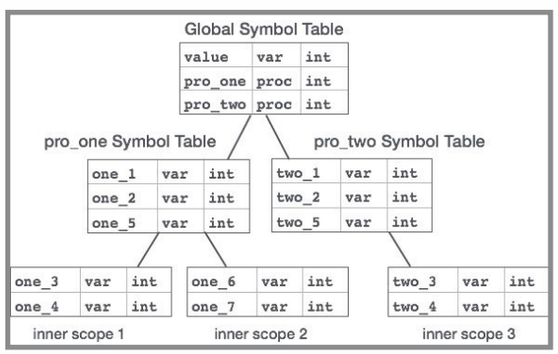


IDE

… encoded in many different ad-hoc ways

`x:int, Γ`

`[3/x].σ`



`lookup(xi)`

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

A standard formalism

Provides

```
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

- Parser
- AST
- Pretty-Printing
- Highlighting

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 definition

Program

↓

Scope Graph

A standard formalism

Scope Graphs

A Theory of Name Resolution

*For **statically lexically scoped** languages*

A unique definition

Program

↓

Scope Graph

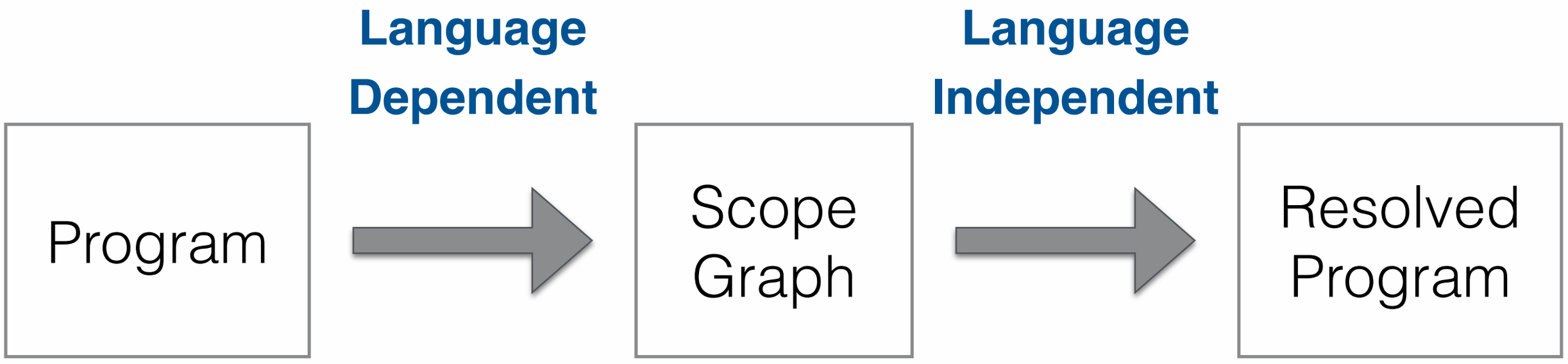
A standard formalism

Scope Graphs

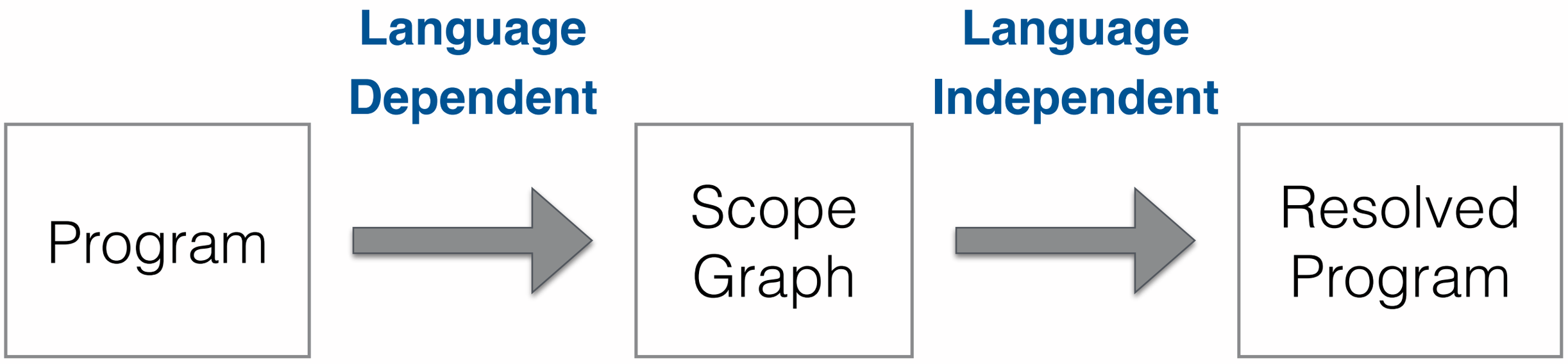
Provides

- 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**

Scope Graph

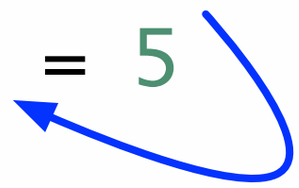
```
def y = x + 1
def x = 5
```

Scope Graph

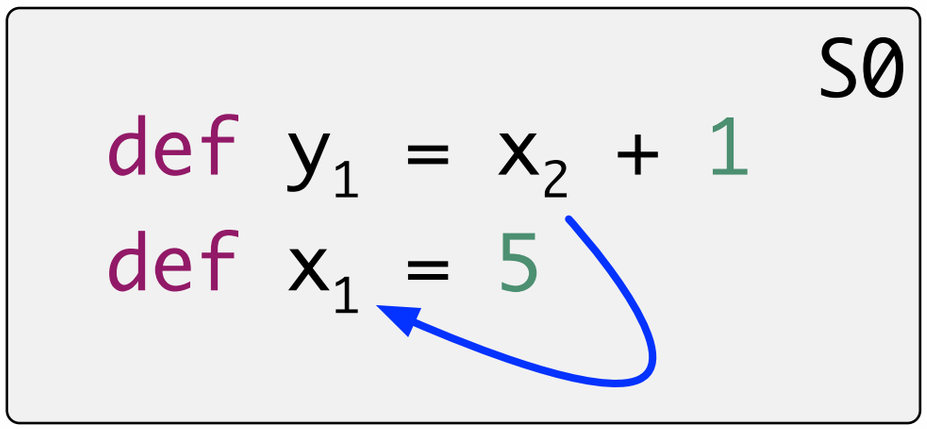
```
def y1 = x2 + 1
def x1 = 5
```

Scope Graph

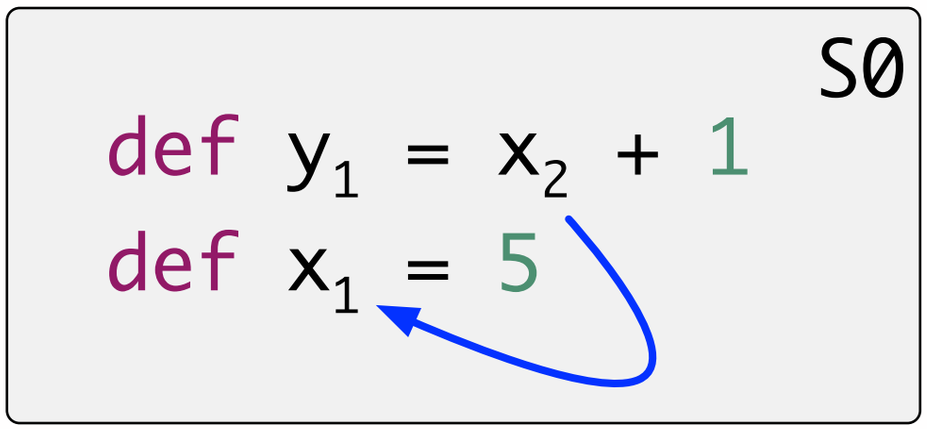
```
def y1 = x2 + 1
def x1 = 5
```

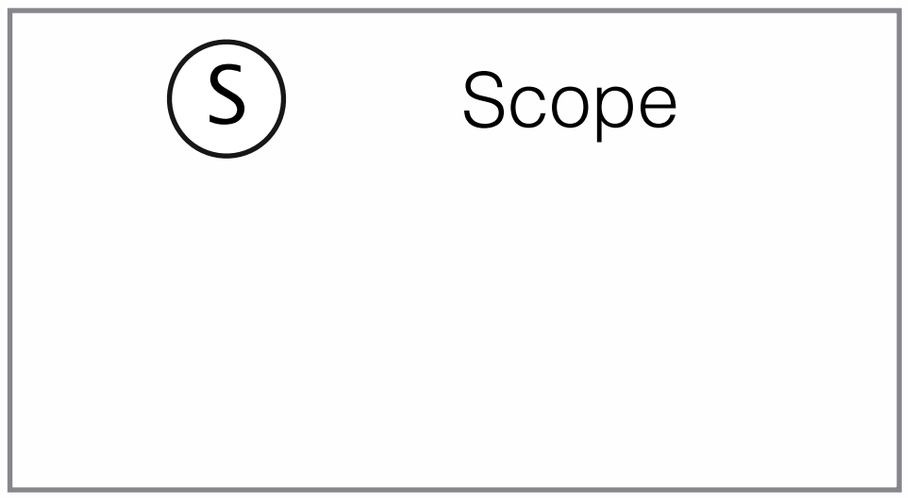


Scope Graph

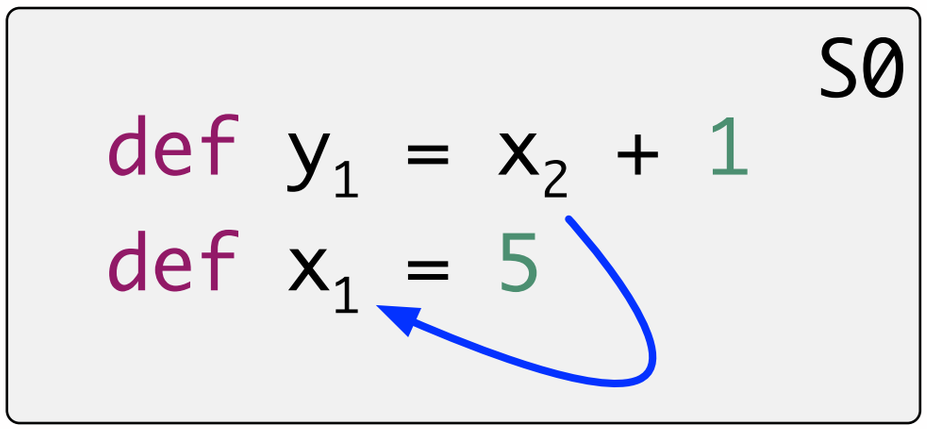


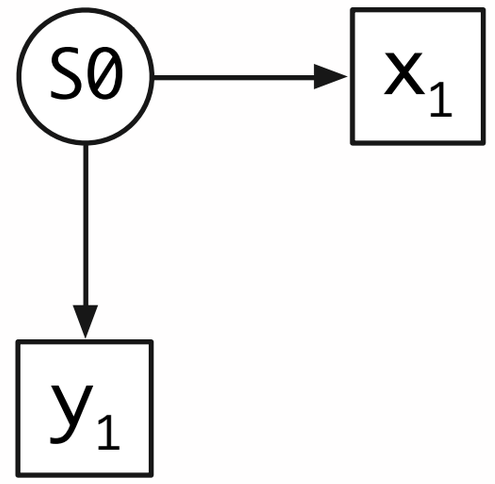
Scope Graph

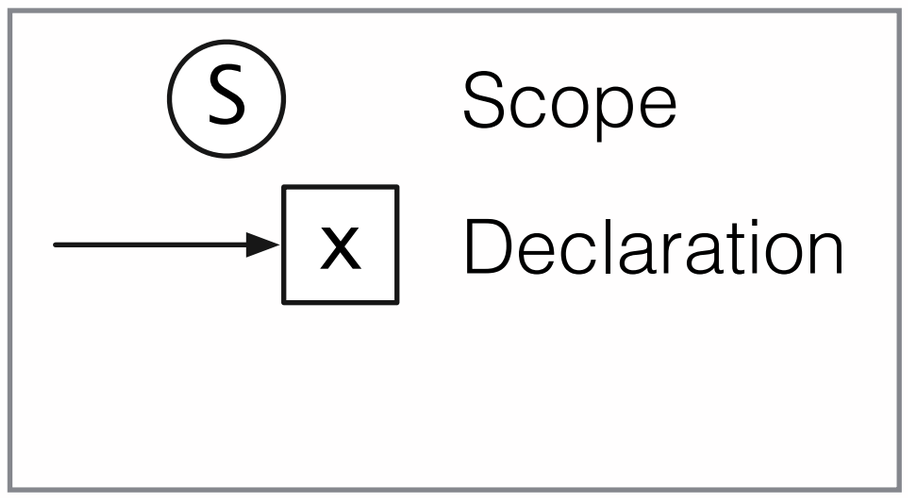




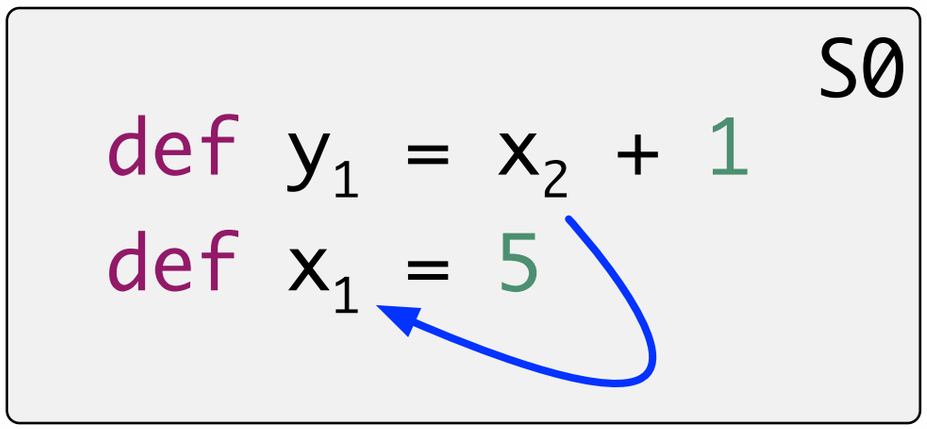
Scope Graph

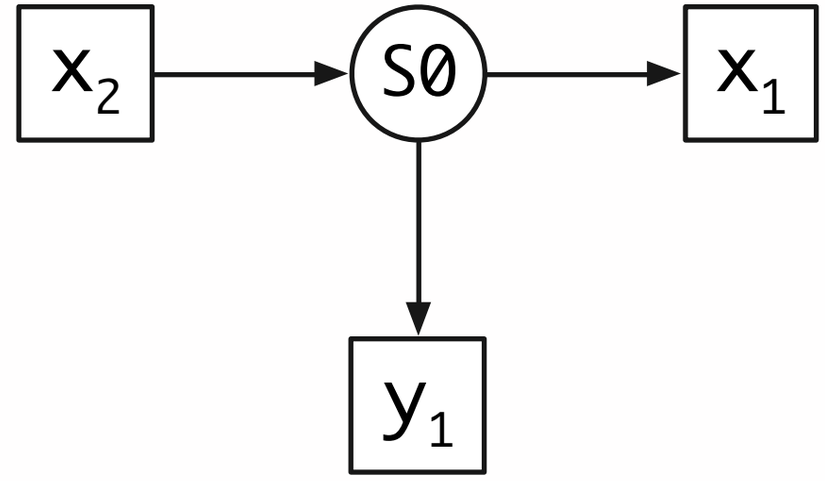


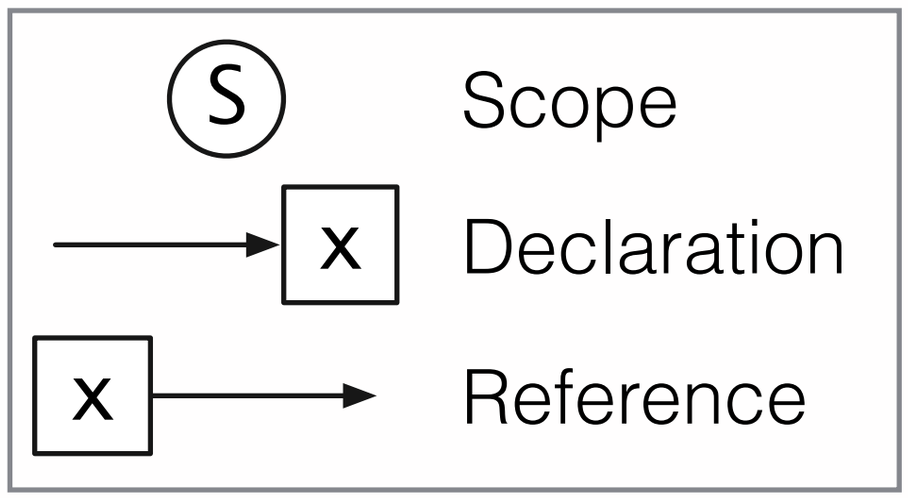




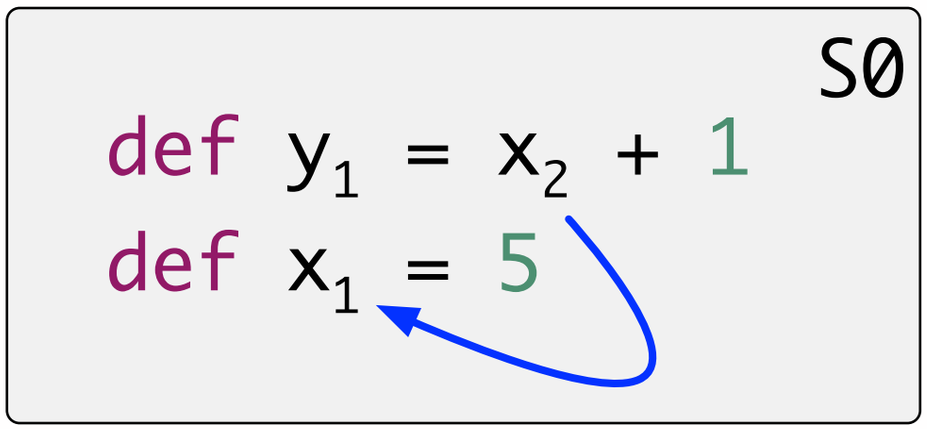
Scope Graph

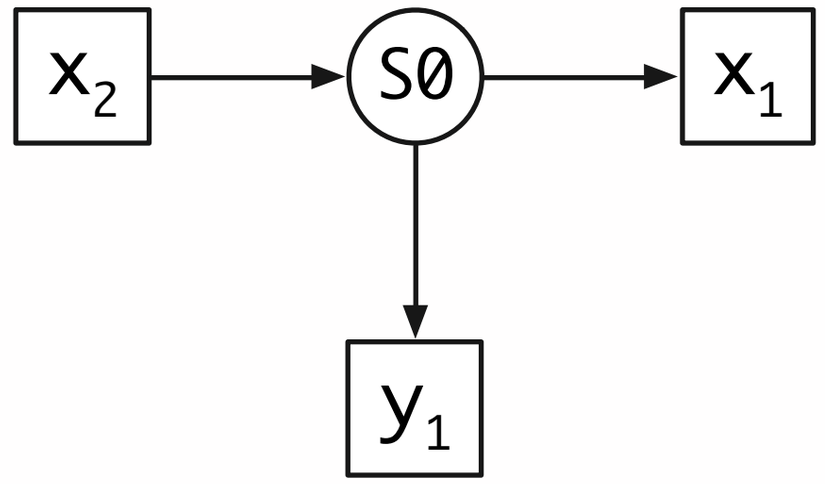

S0
def y_1 = x_2 + 1
def x_1 = 5

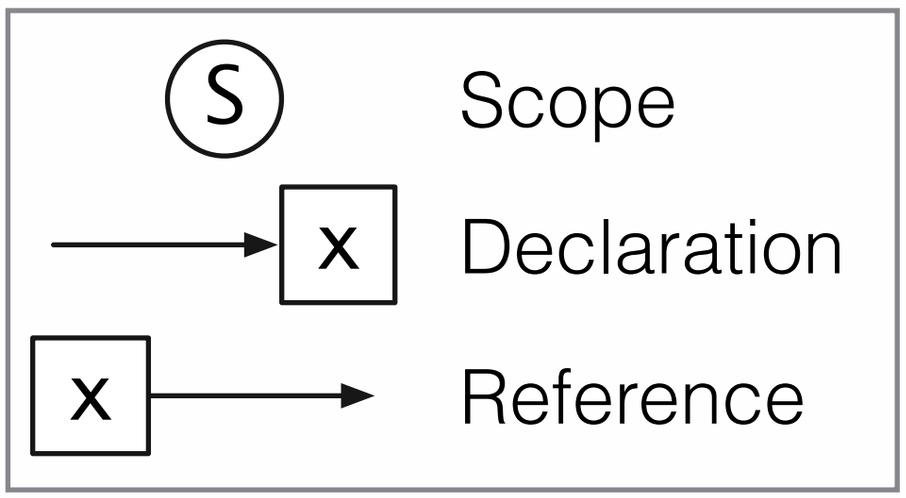

x_2
S0
x_1
y_1

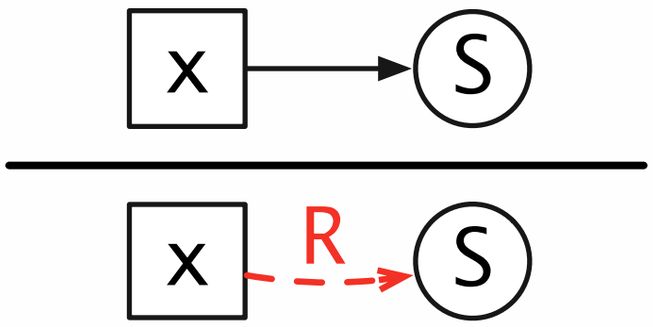

S
Scope
x
Declaration
x
Reference

Scope Graph



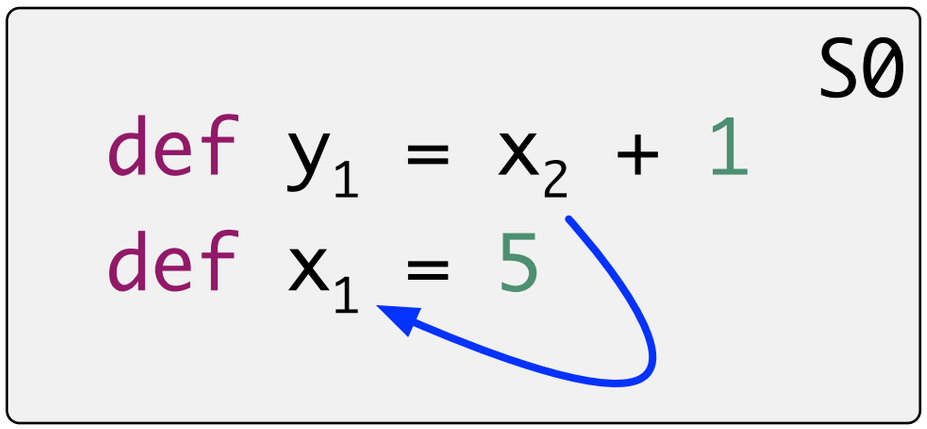


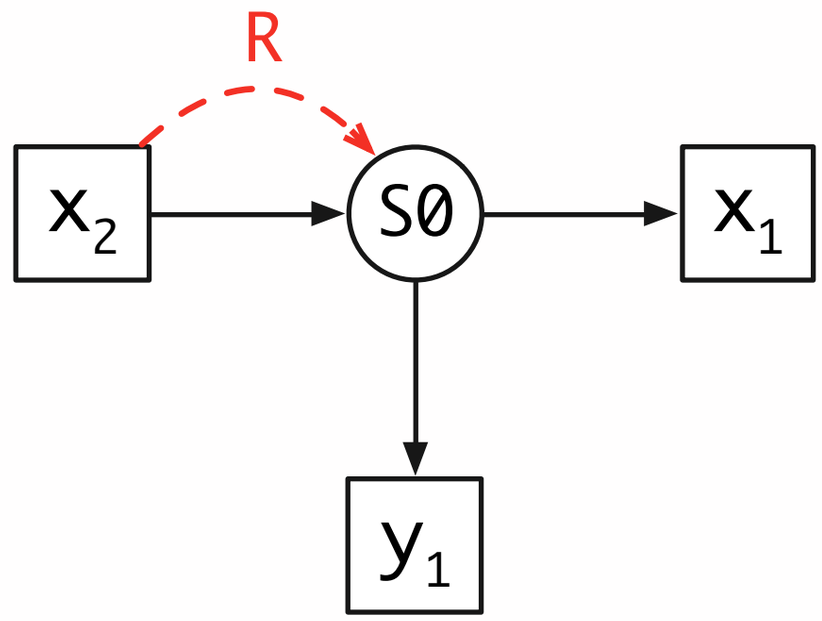


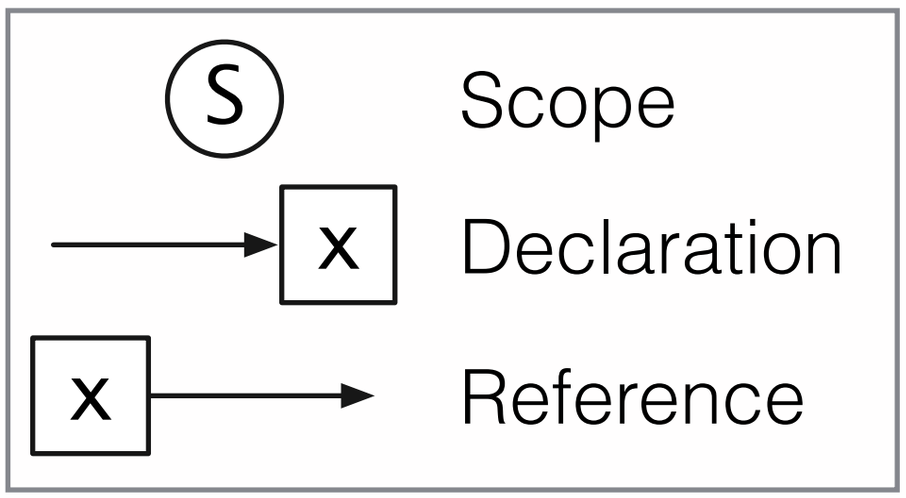


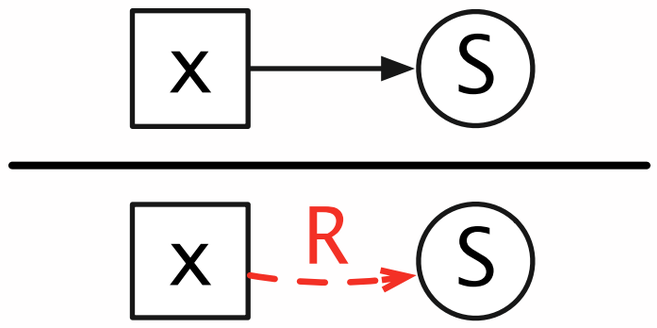
Reference Step

Scope Graph



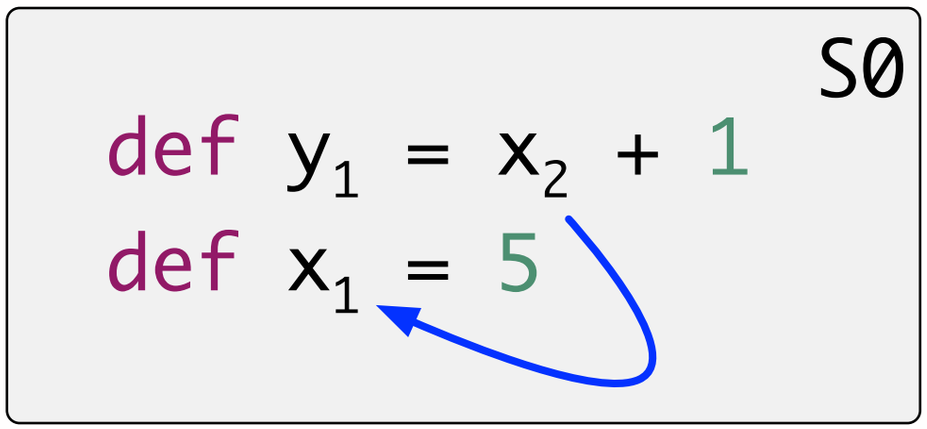


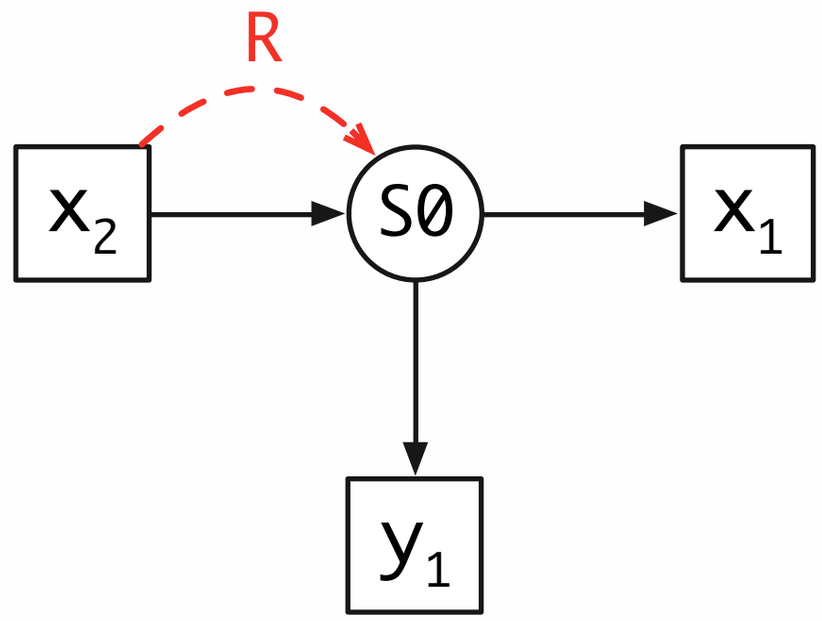


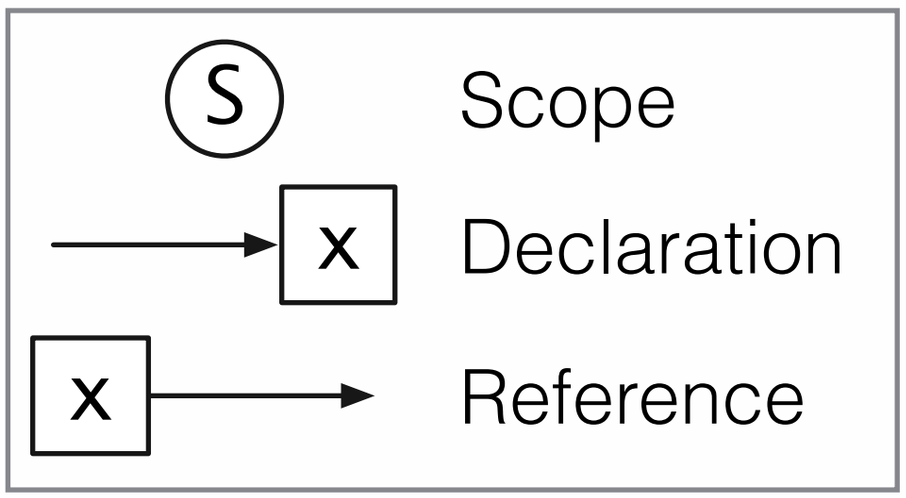


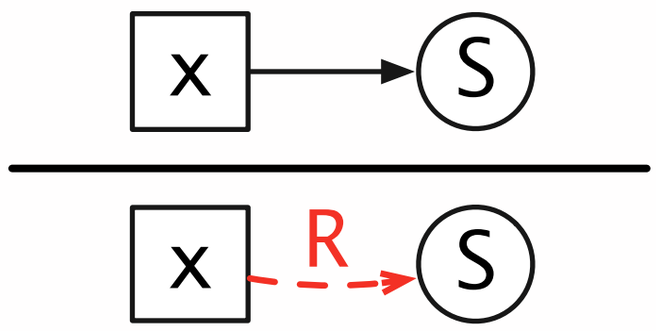
Reference Step

Scope Graph

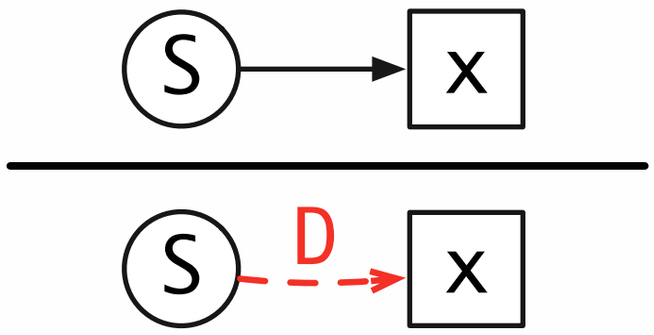






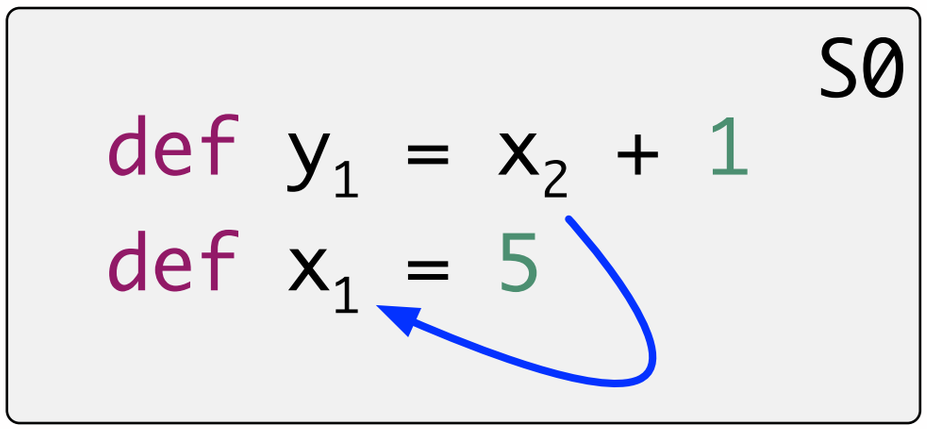


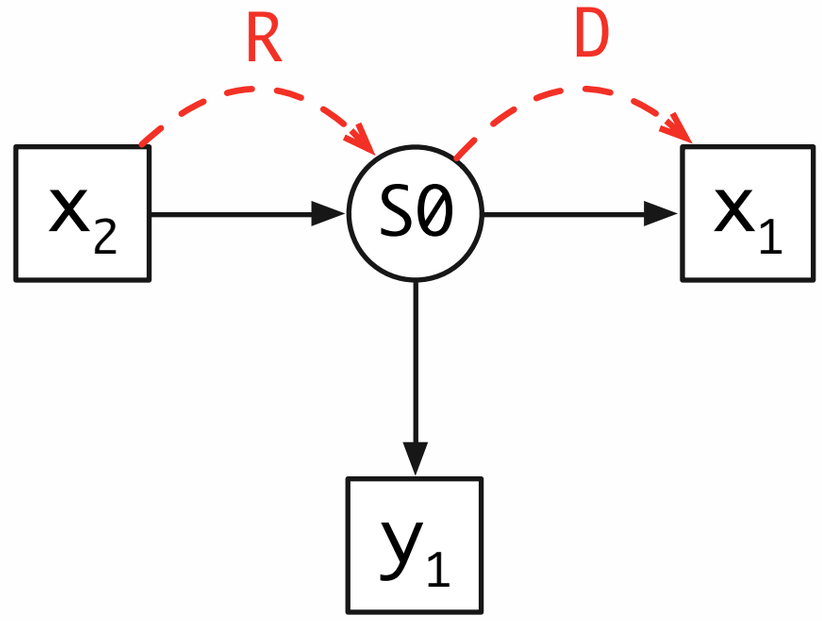
Reference Step

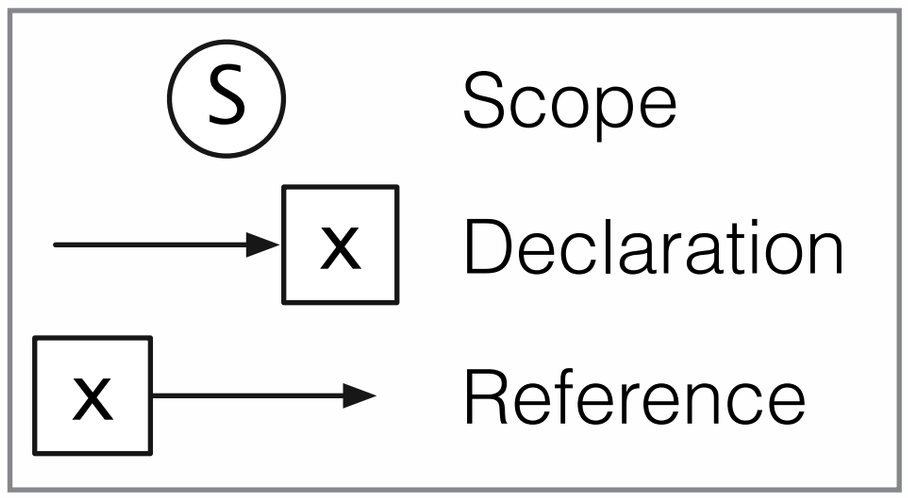


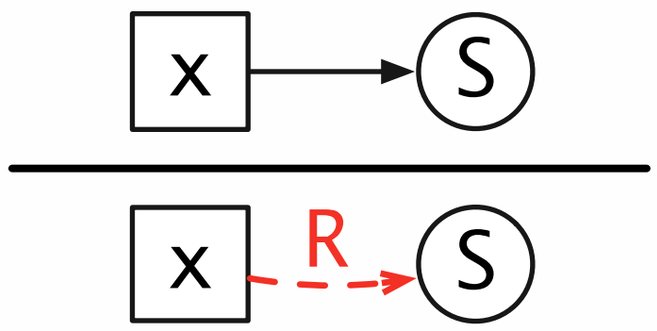
Declaration Step

Scope Graph

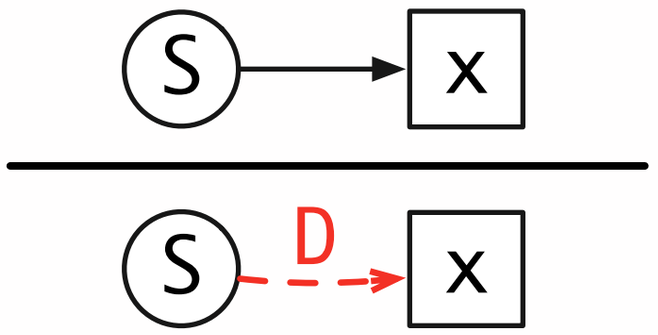








Reference Step



Declaration Step

Ambiguous Resolutions

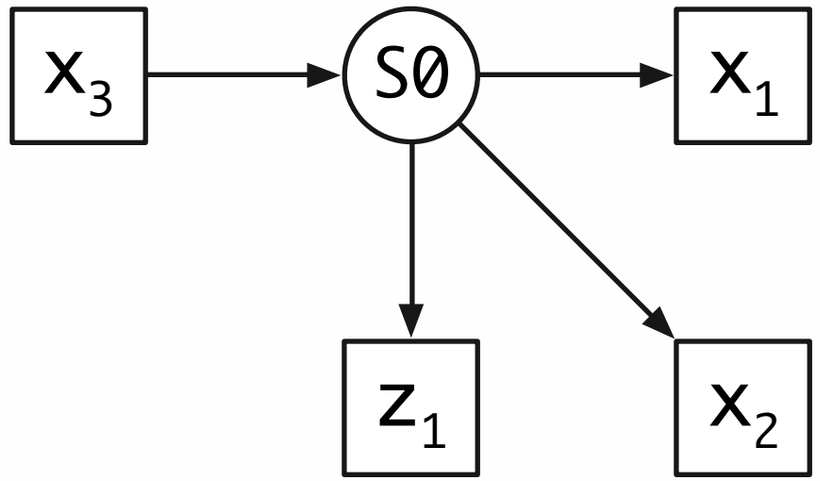
S0

```
def x₁ = 5

def x₂ = 3

def z₁ = x₃ + 1
```

Ambiguous Resolutions

S0

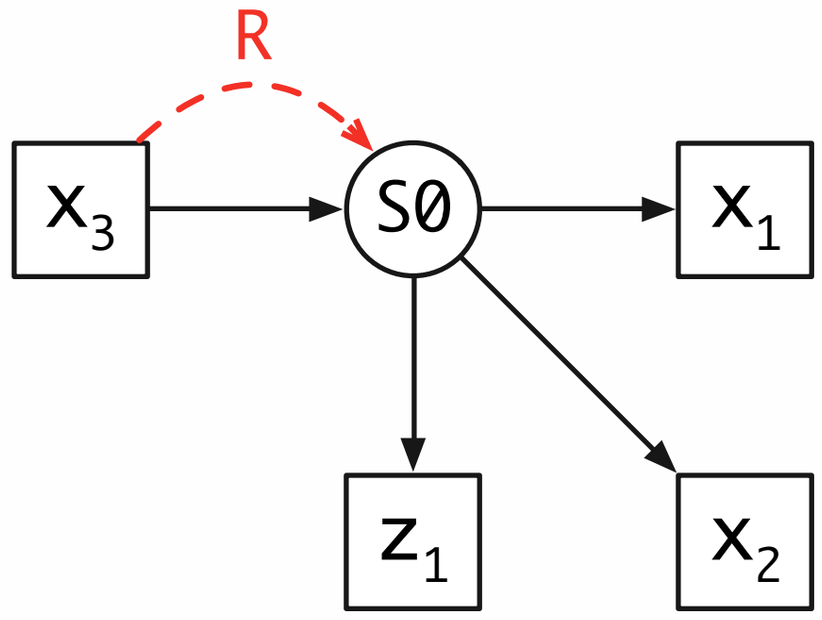
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

S0

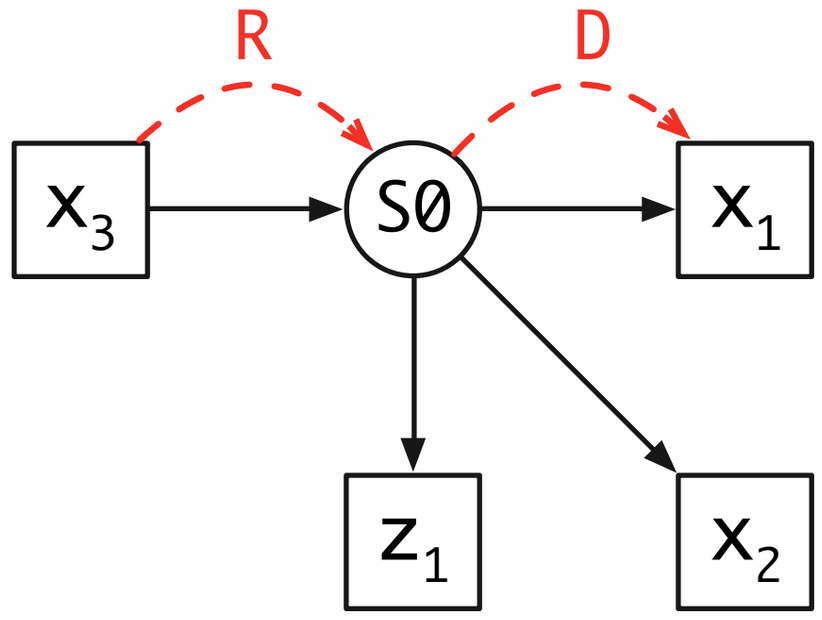
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

S0

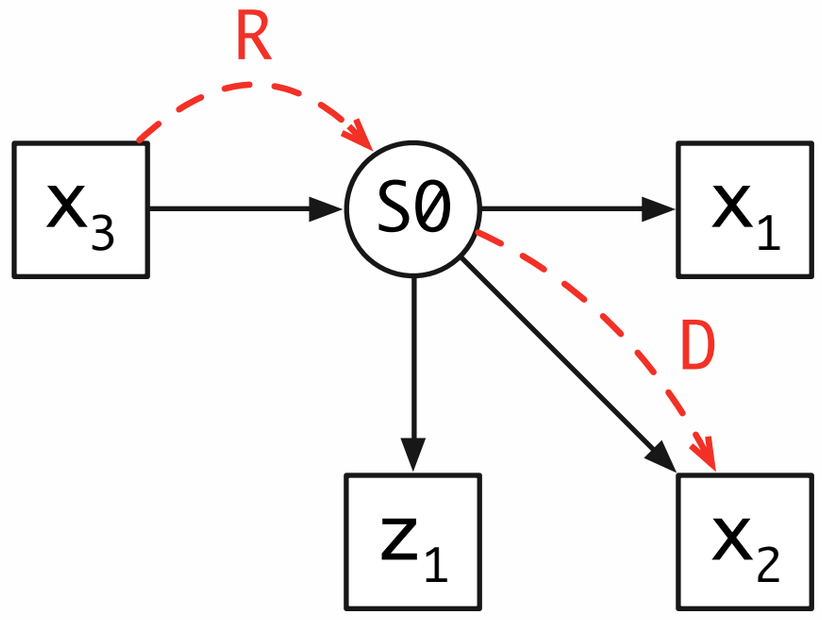
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

S0

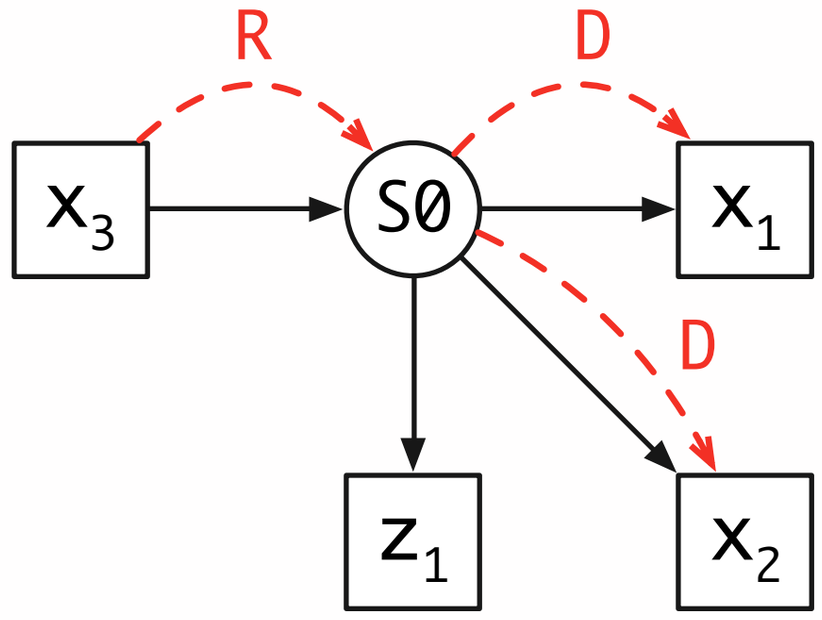
```
def x1 = 5

def x2 = 3

def z1 = x3 + 1
```



Ambiguous Resolutions

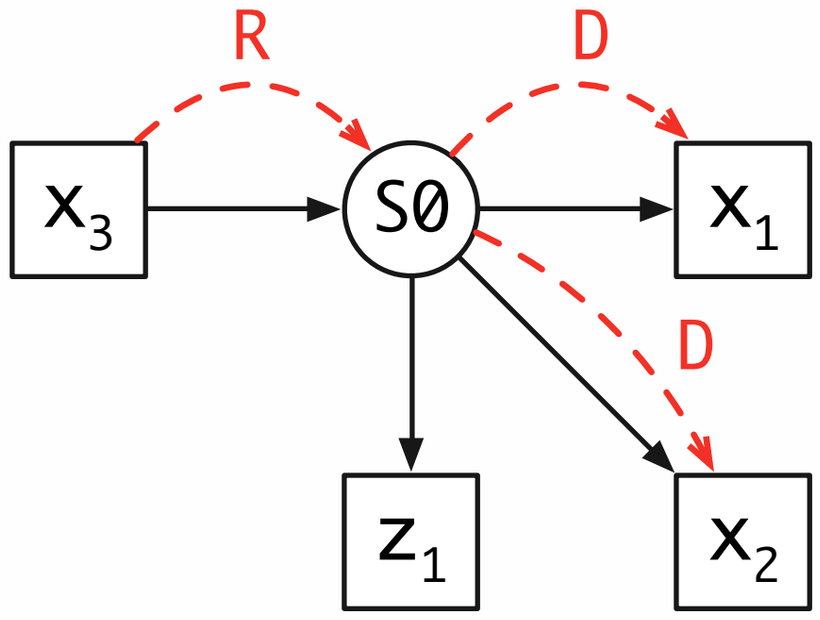
S0

```
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

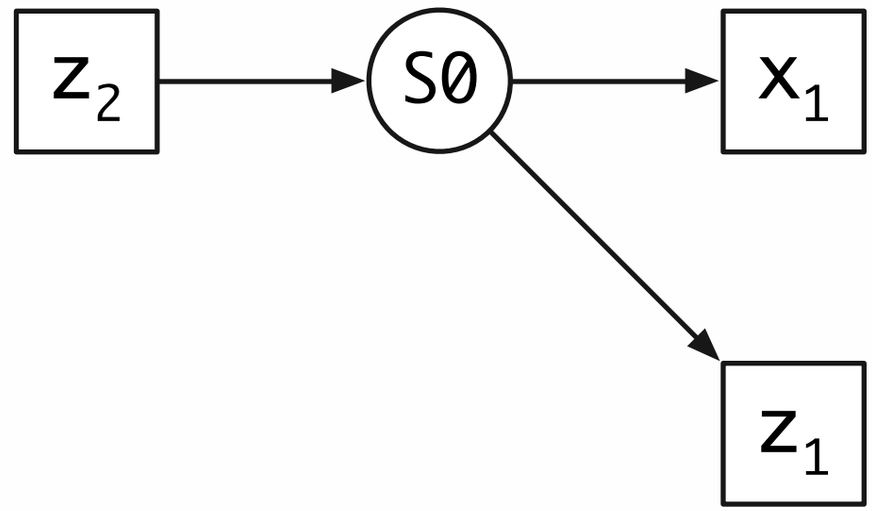
```
                                S0
def x1 = z2 5


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

Lexical Scoping

```
                                S0
def x1 = z2 5

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

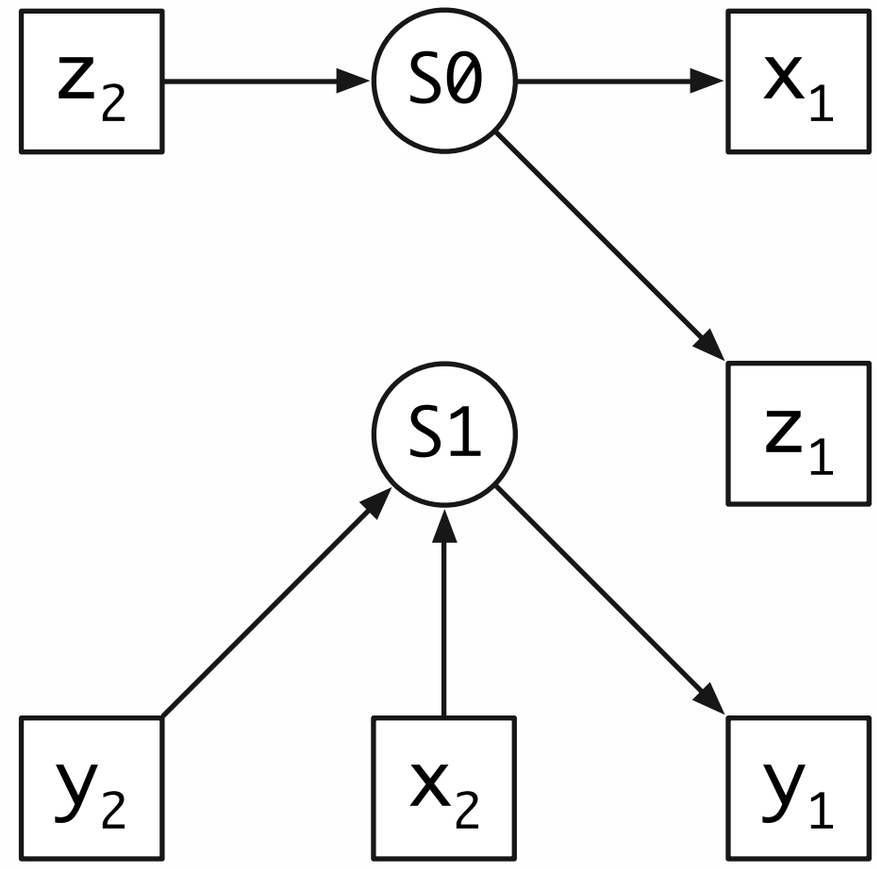
Lexical Scoping

```
                                    S0
def x1 = z2 5

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



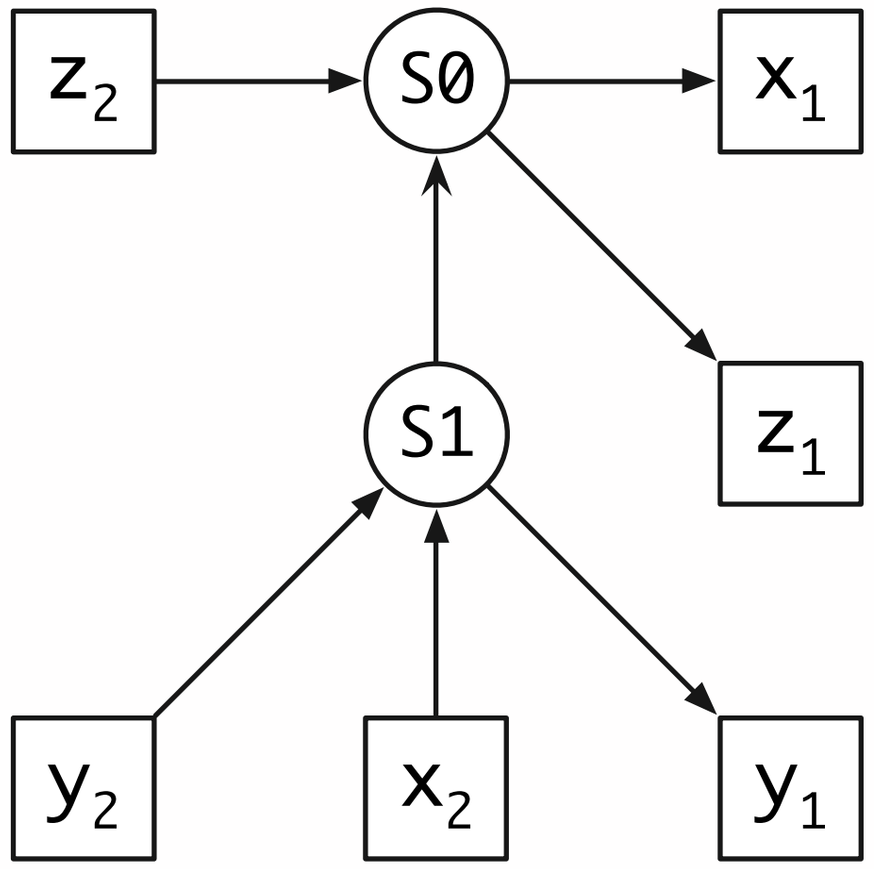
Lexical Scoping

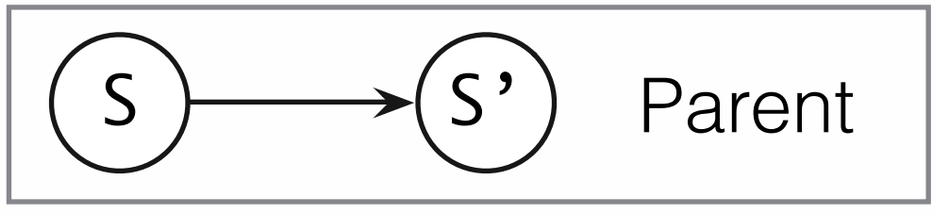
```
                                  S0
def x1 = z2 5

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



Lexical Scoping

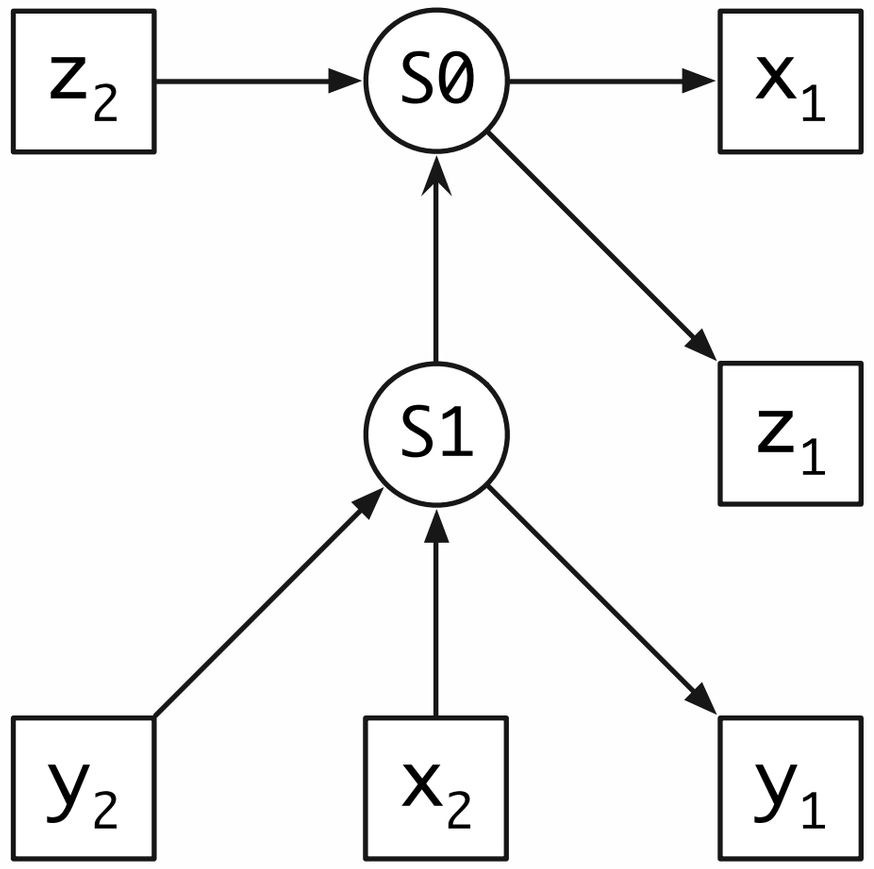
```
                                  S0
def x1 = z2 5

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

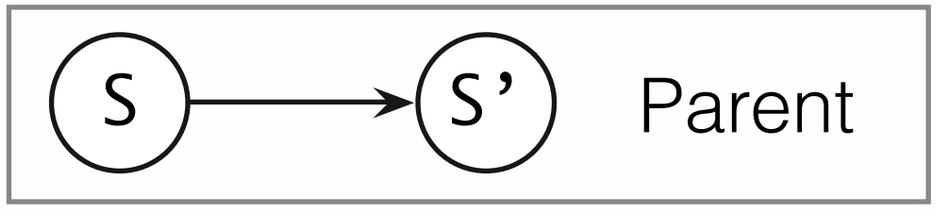




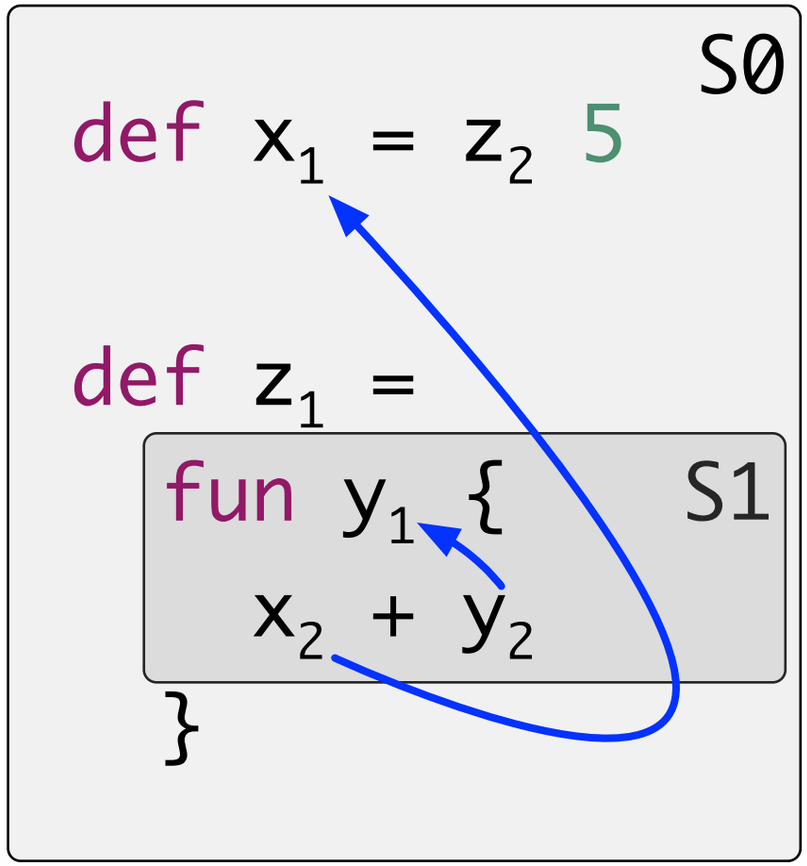
Lexical Scoping

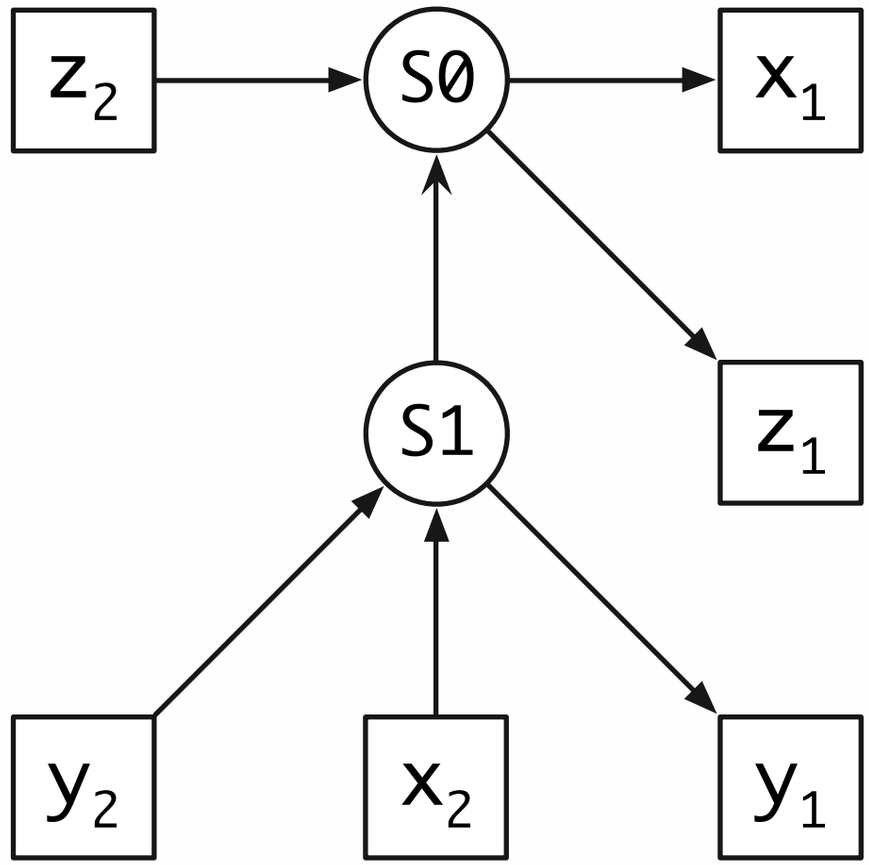
```
                                  S0
def x1 = z2 5

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

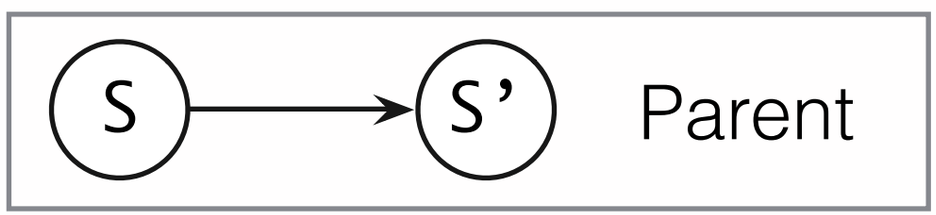




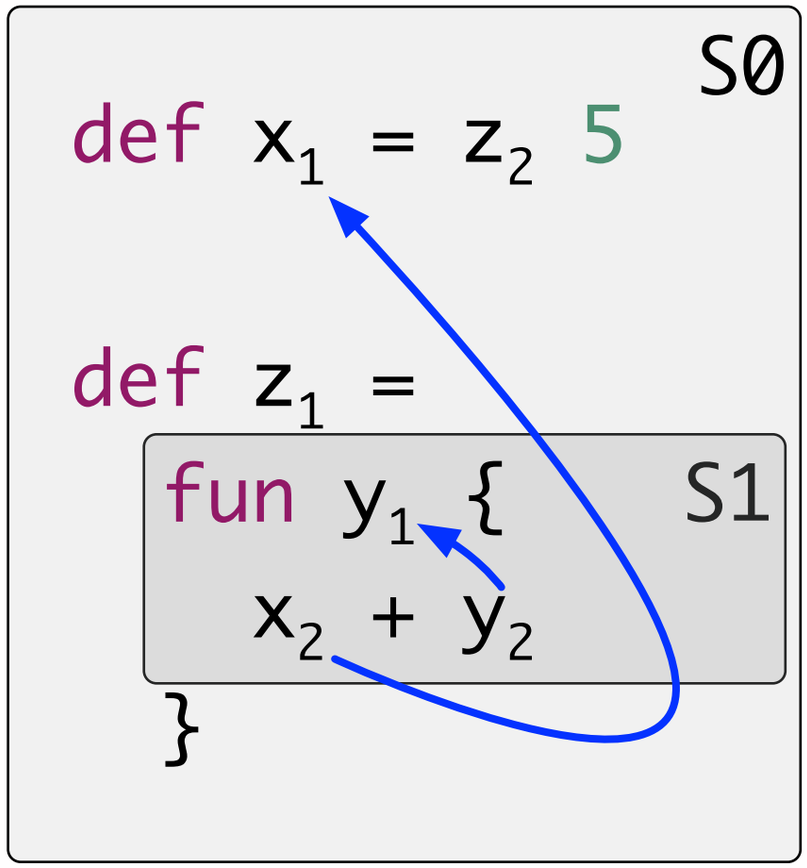
Lexical Scoping

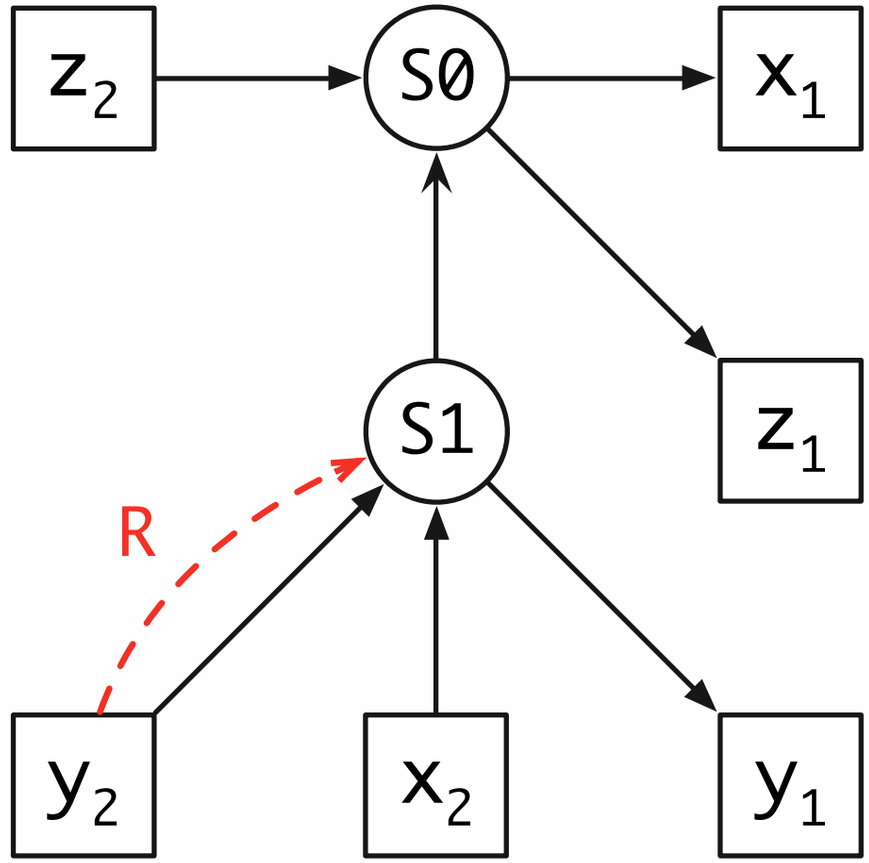


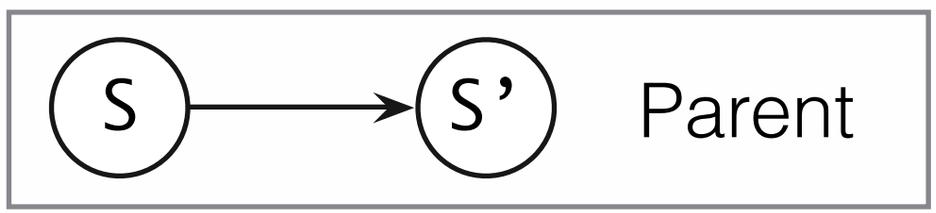




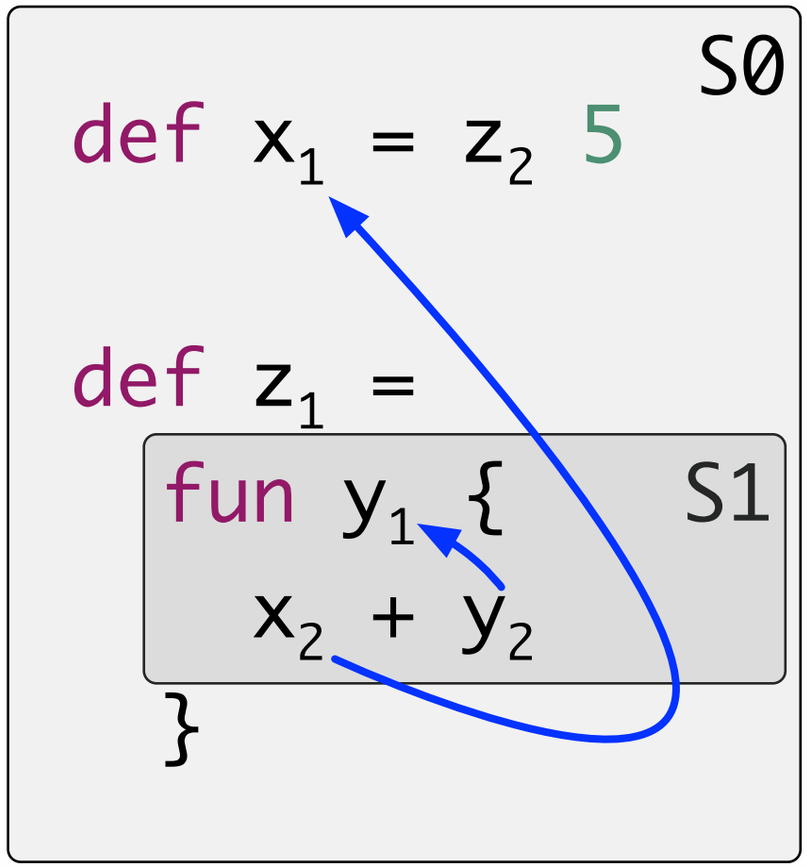
Lexical Scoping

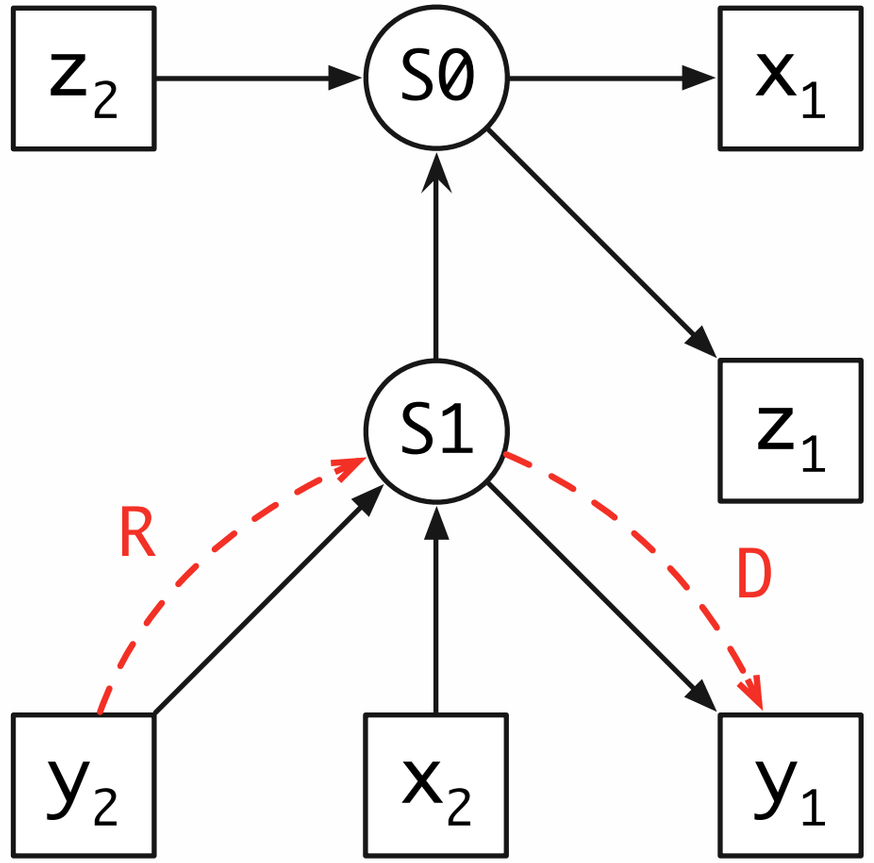


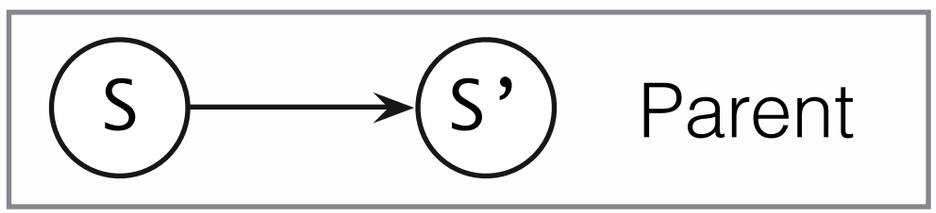




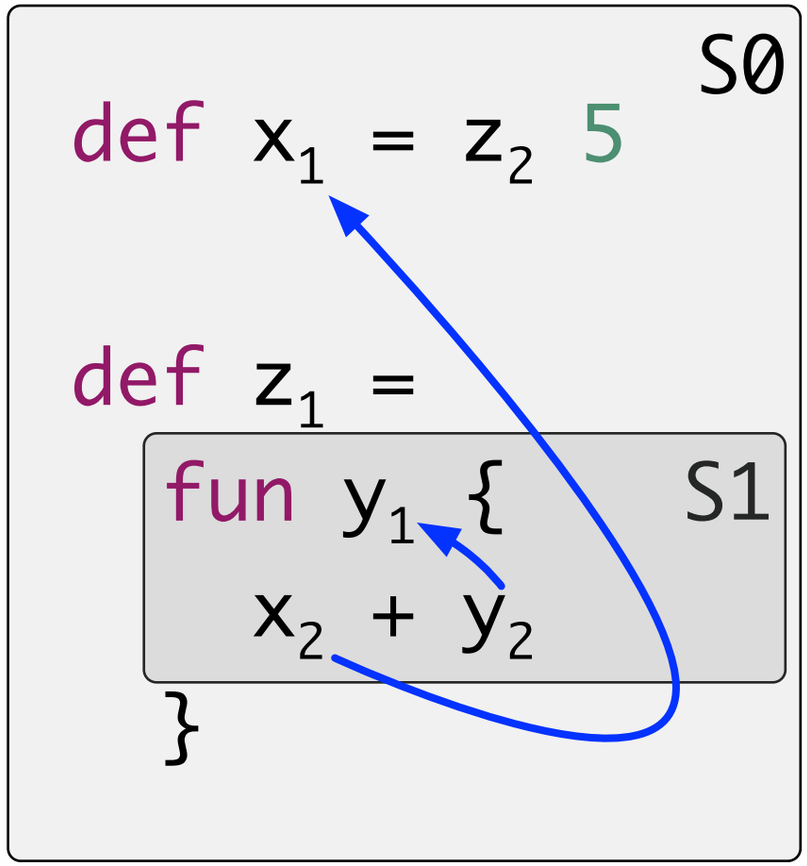
Lexical Scoping

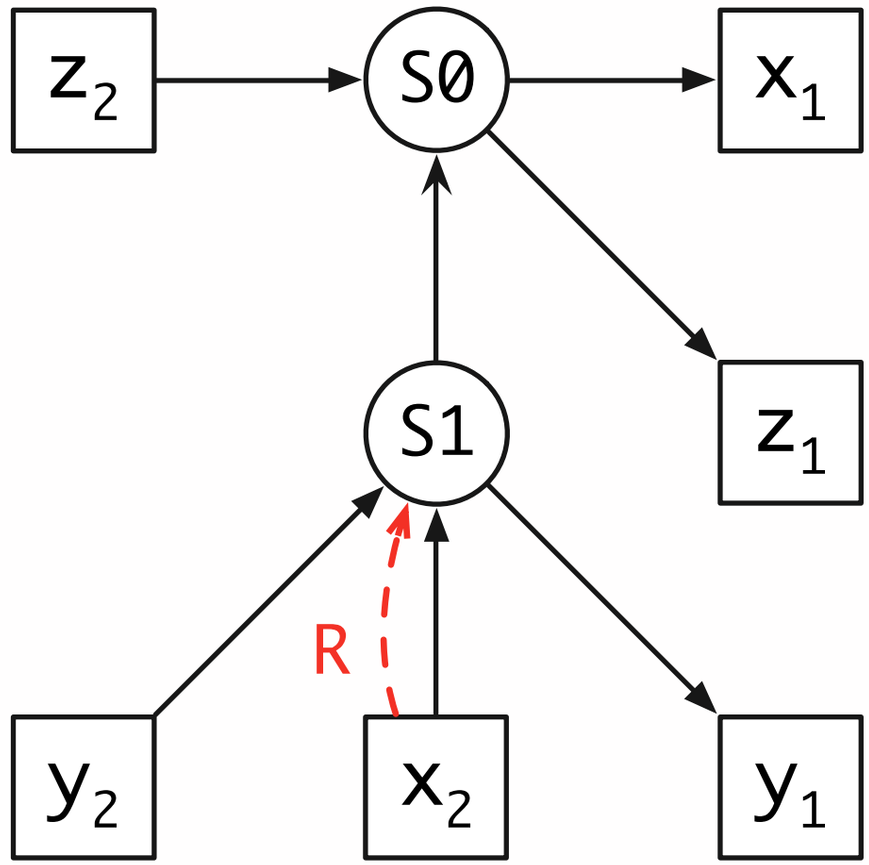


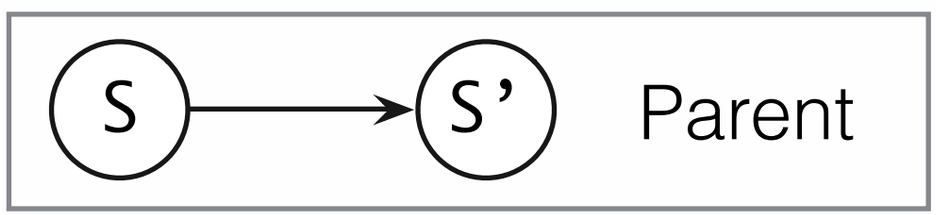




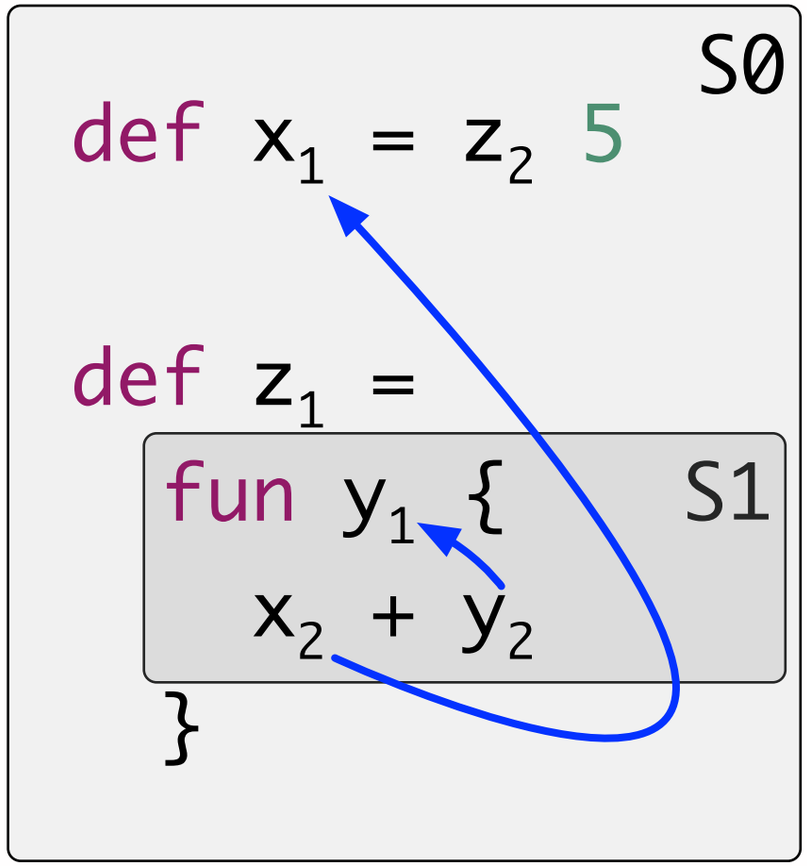
Lexical Scoping

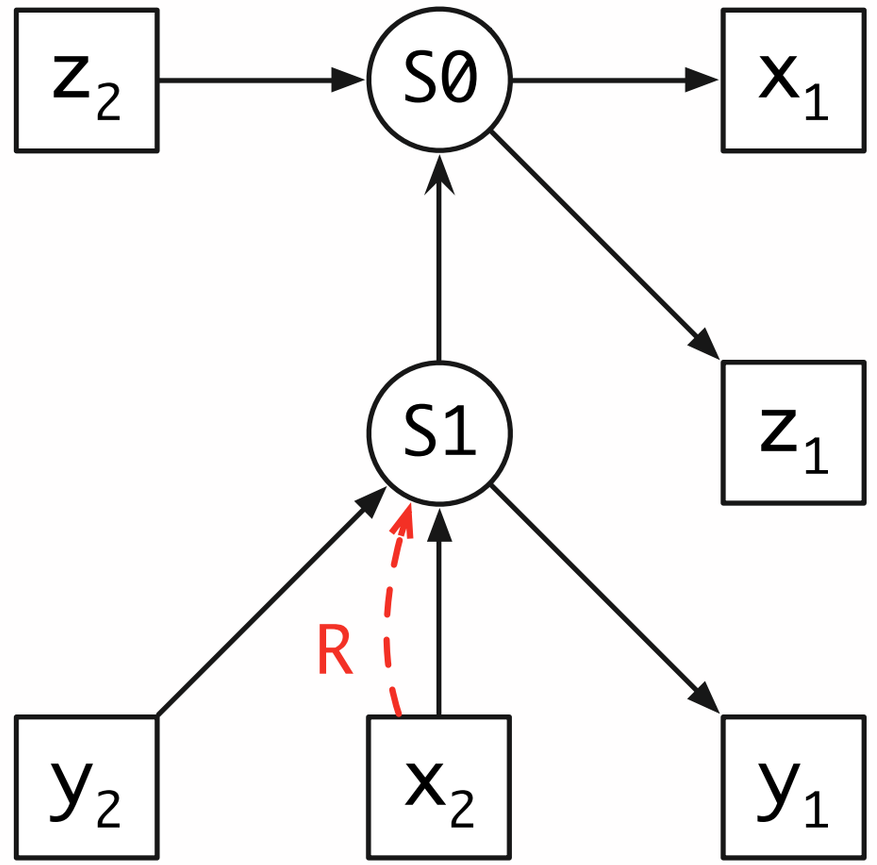


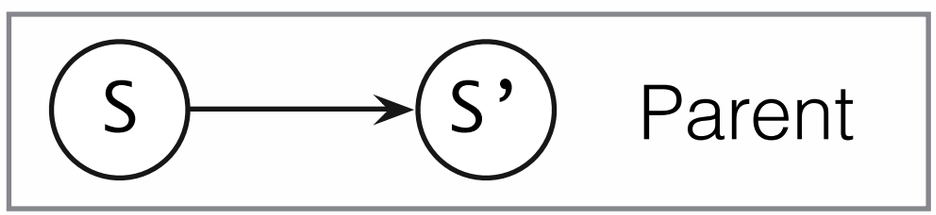


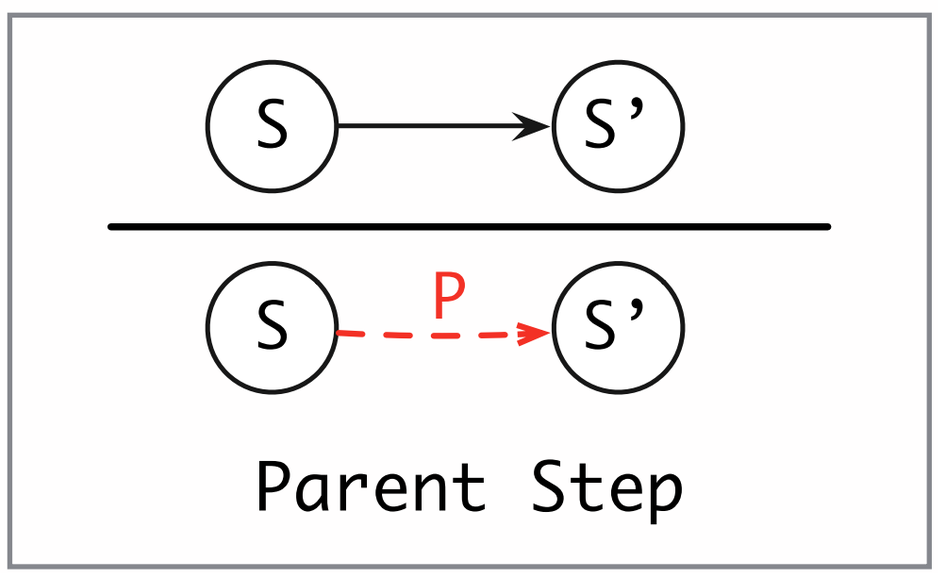


Lexical Scoping

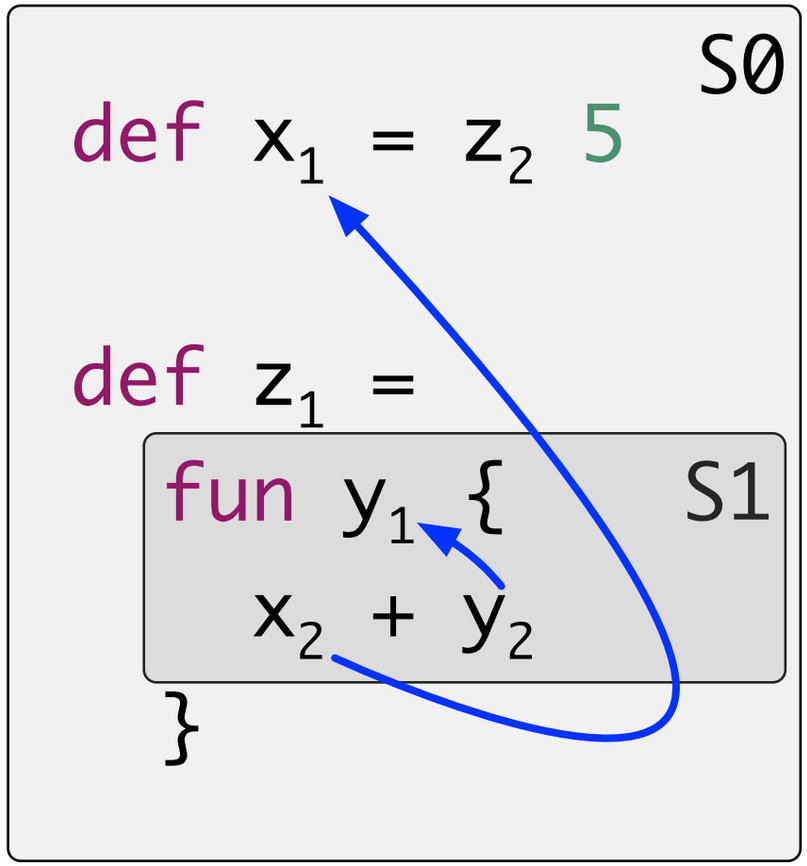


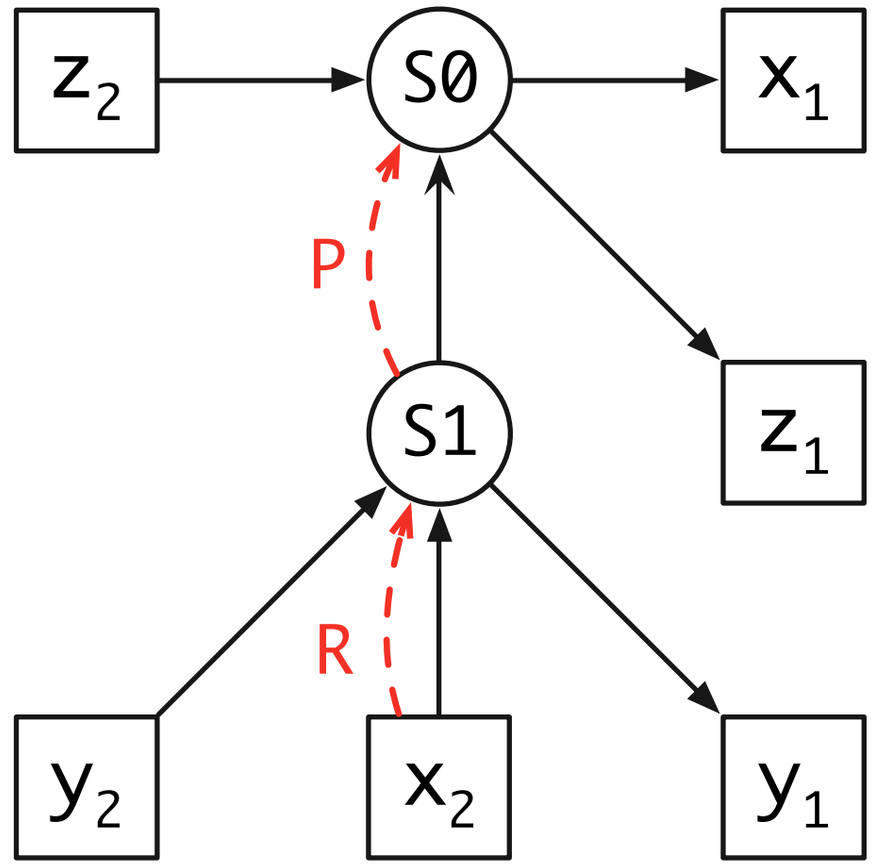


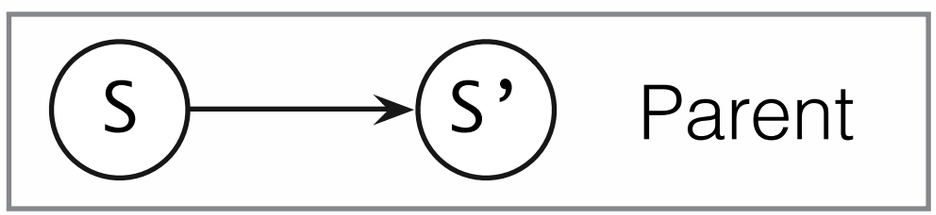


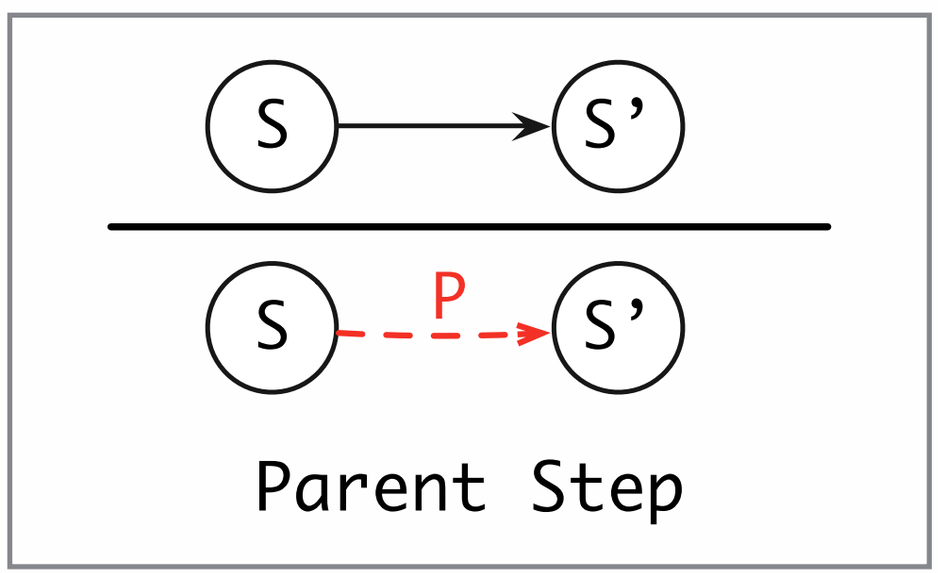


Lexical Scoping

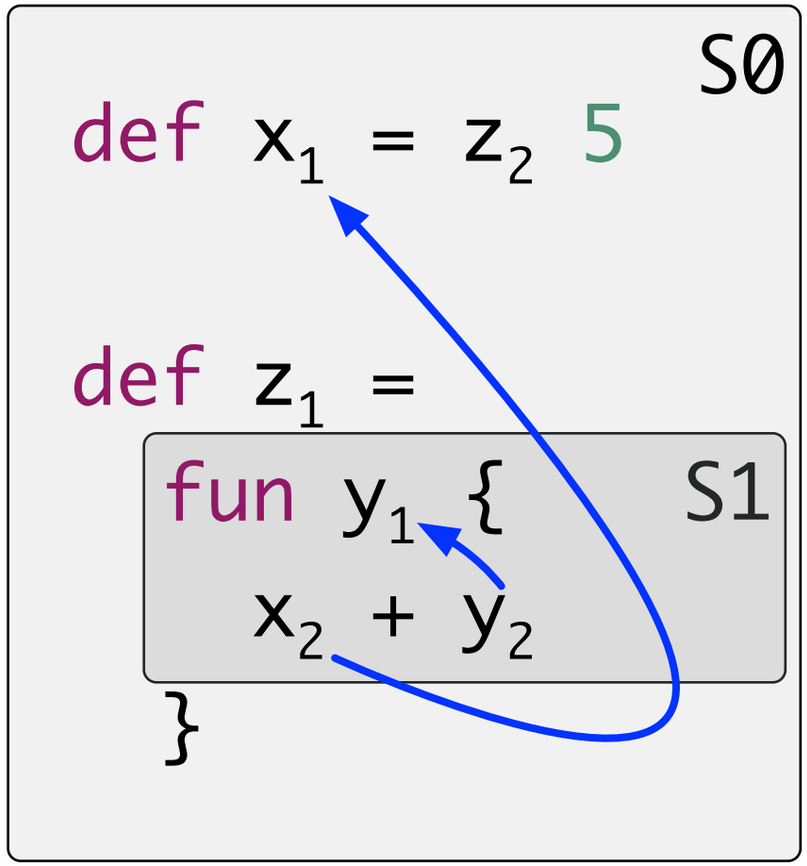


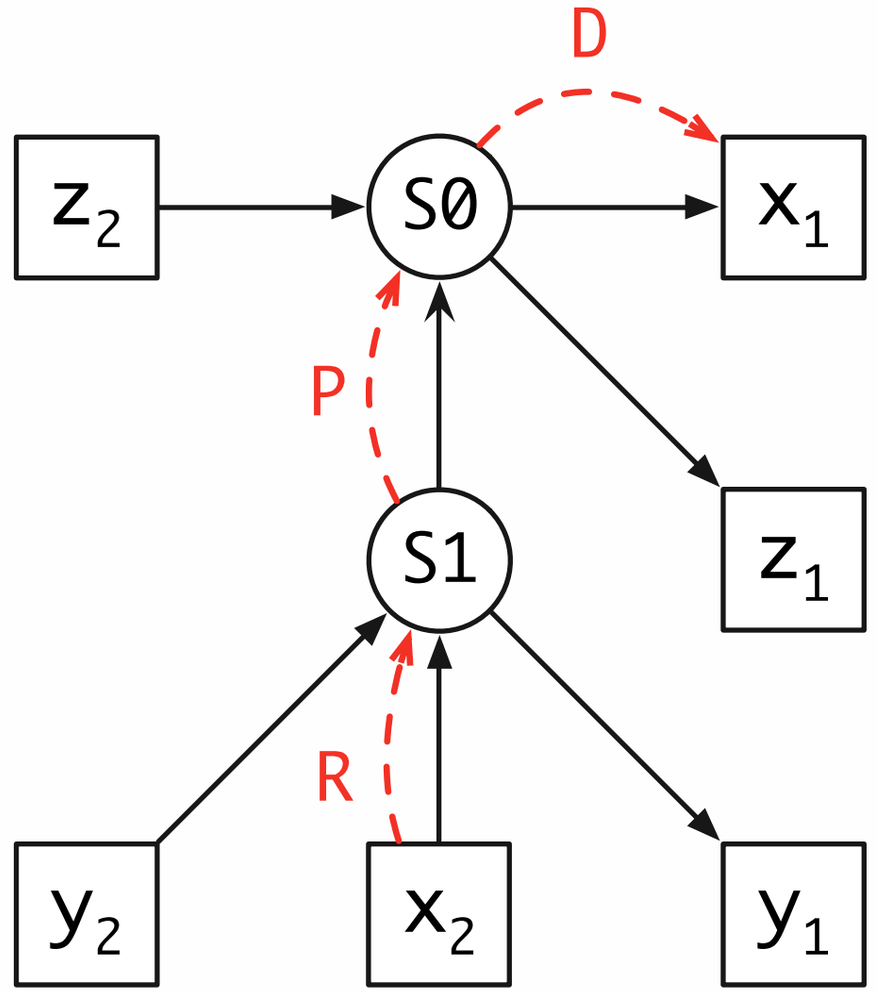


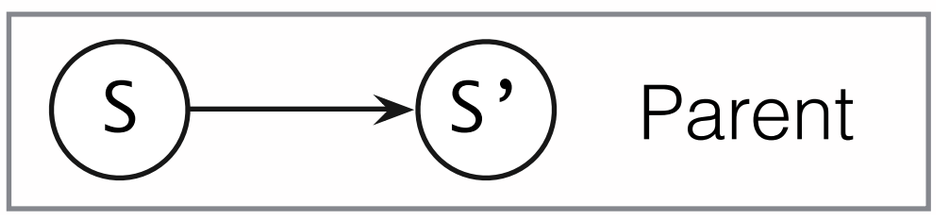


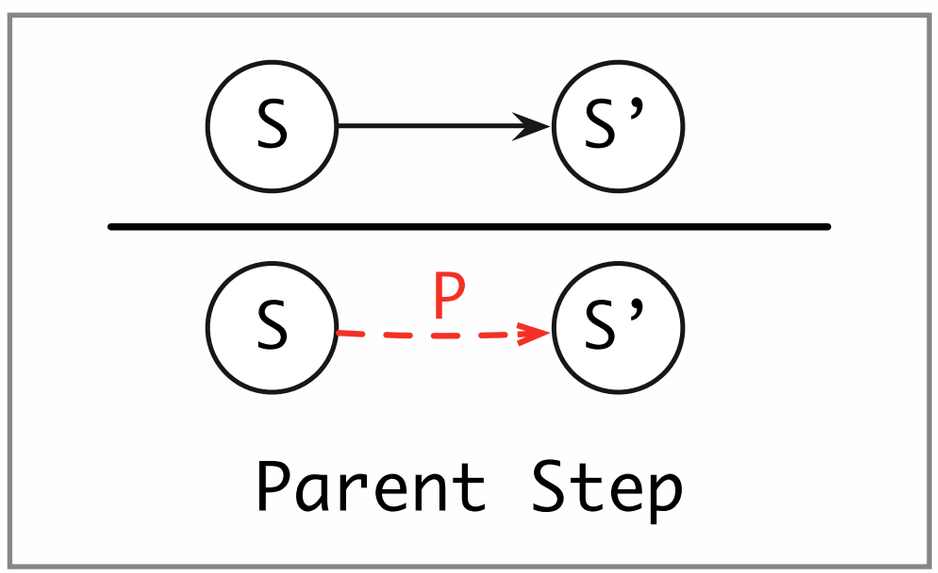


Lexical Scoping

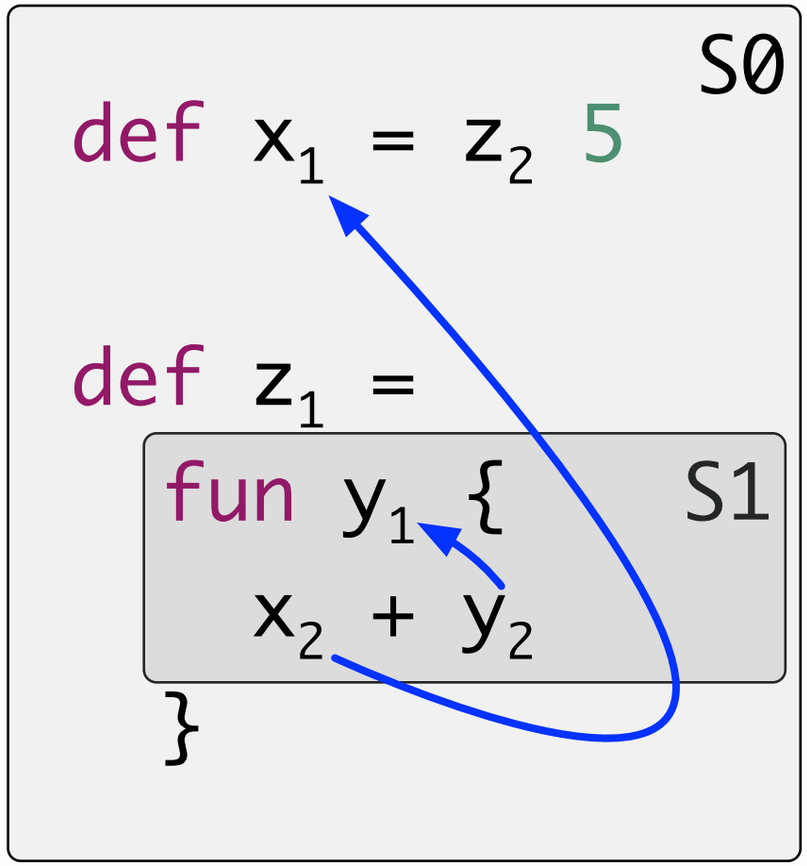


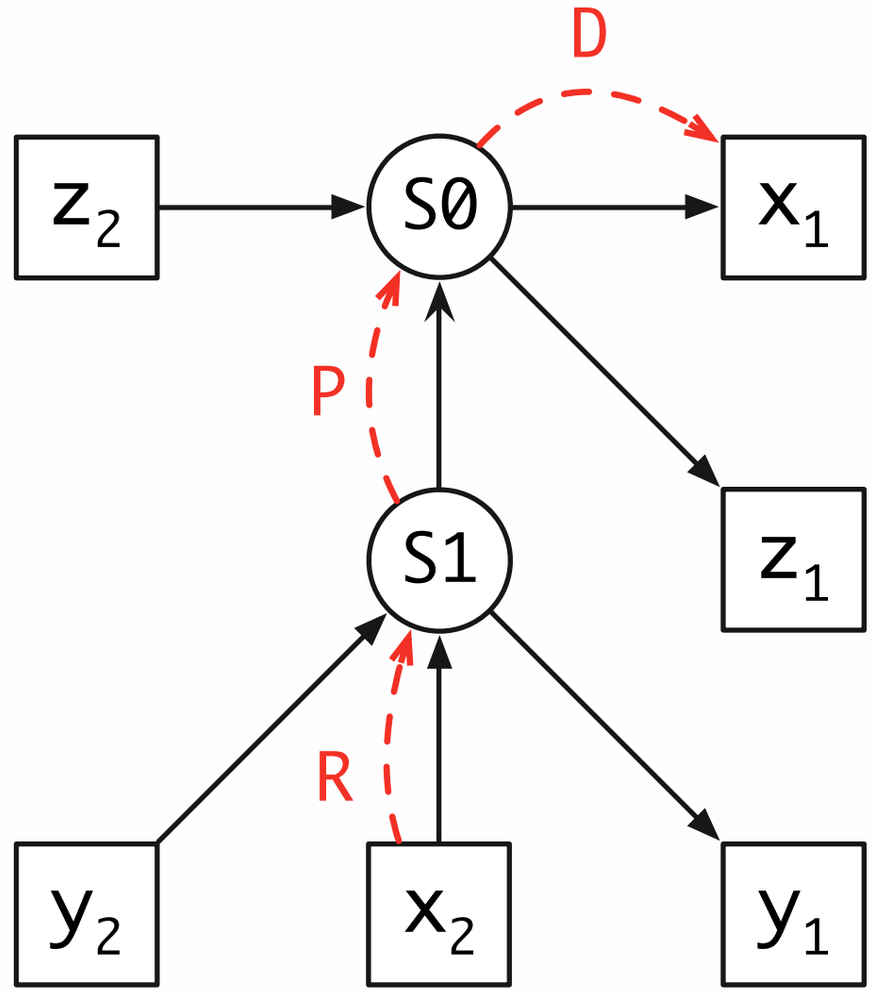


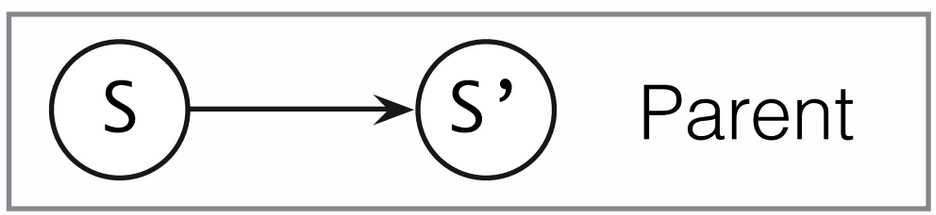




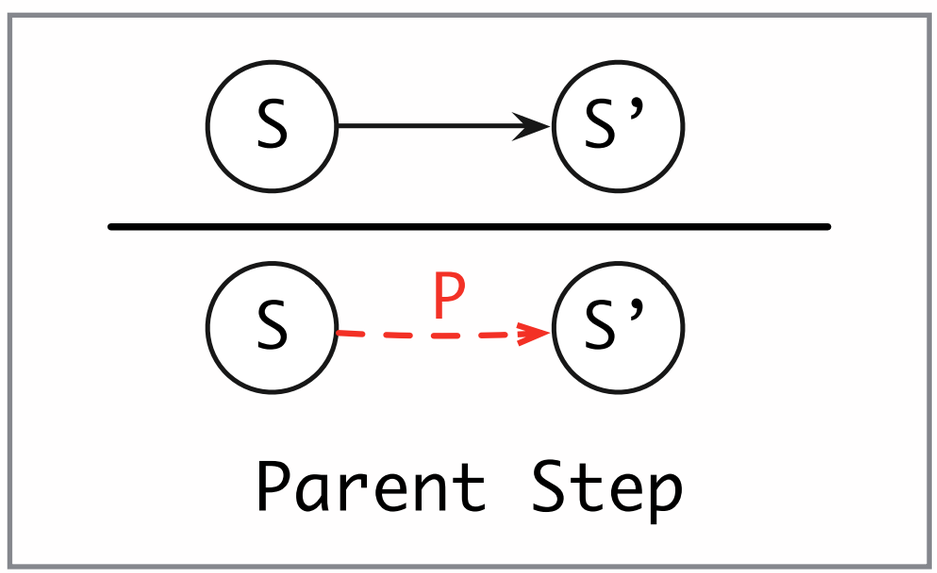
Lexical Scoping







Well formed path: R.P*.D



Shadowing

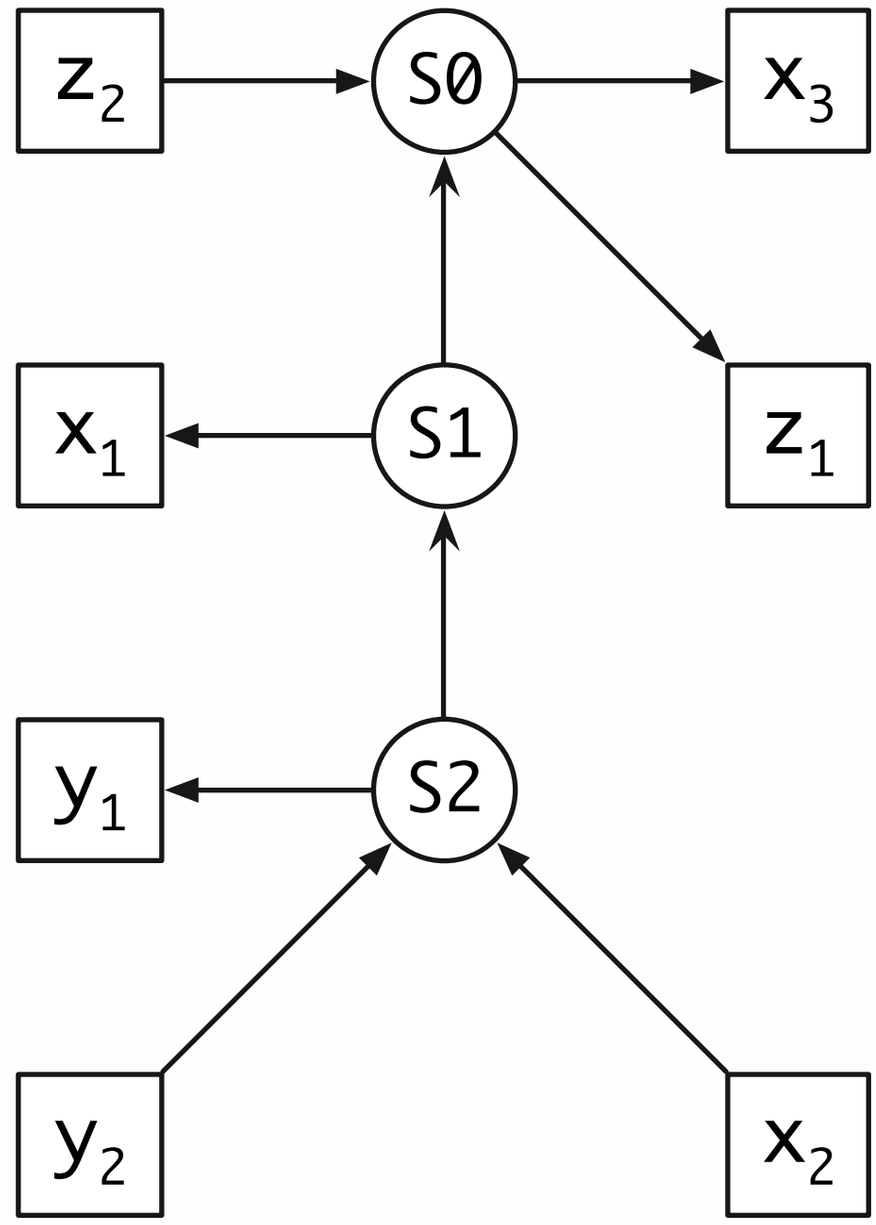
```
def x3 = z2 5 7

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
    }
  }
```

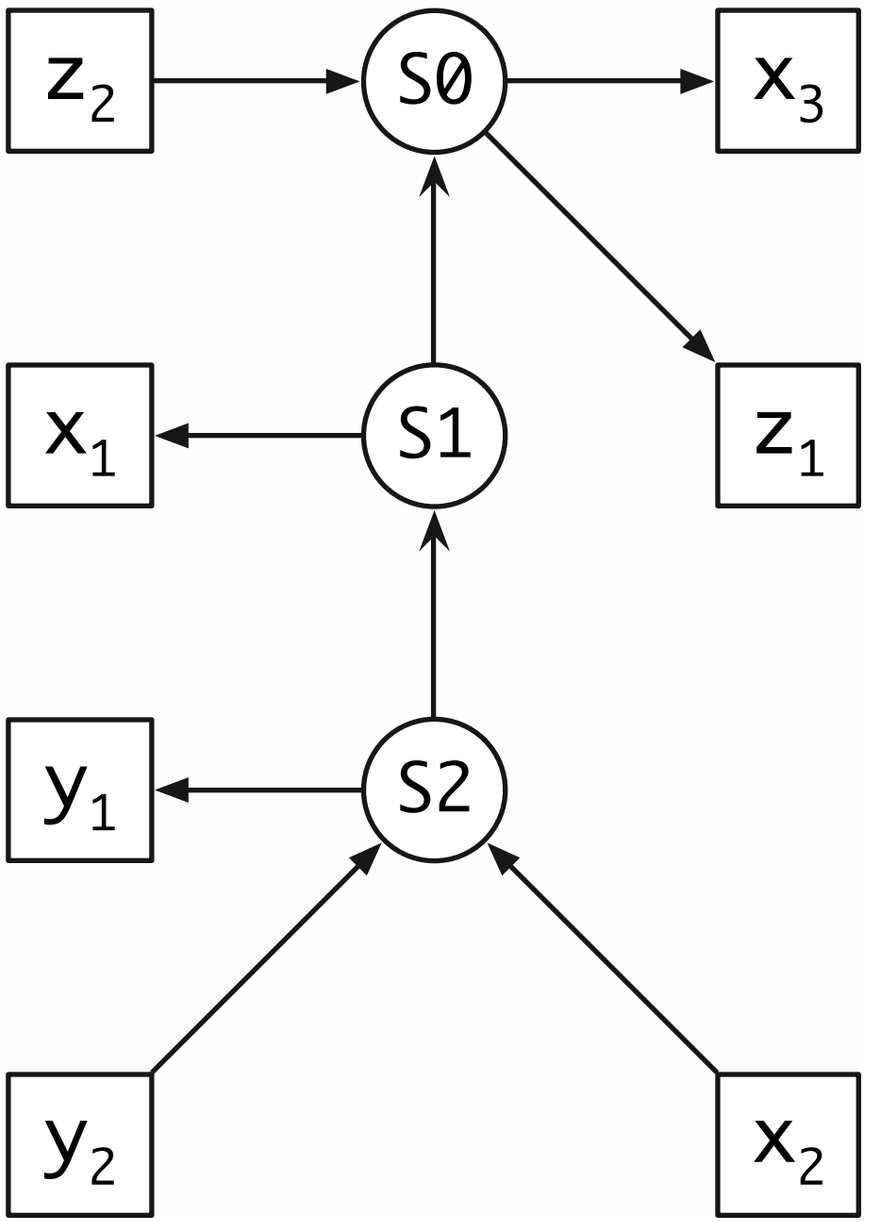
Shadowing

```
def x3 = z2 5 7                    S0

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



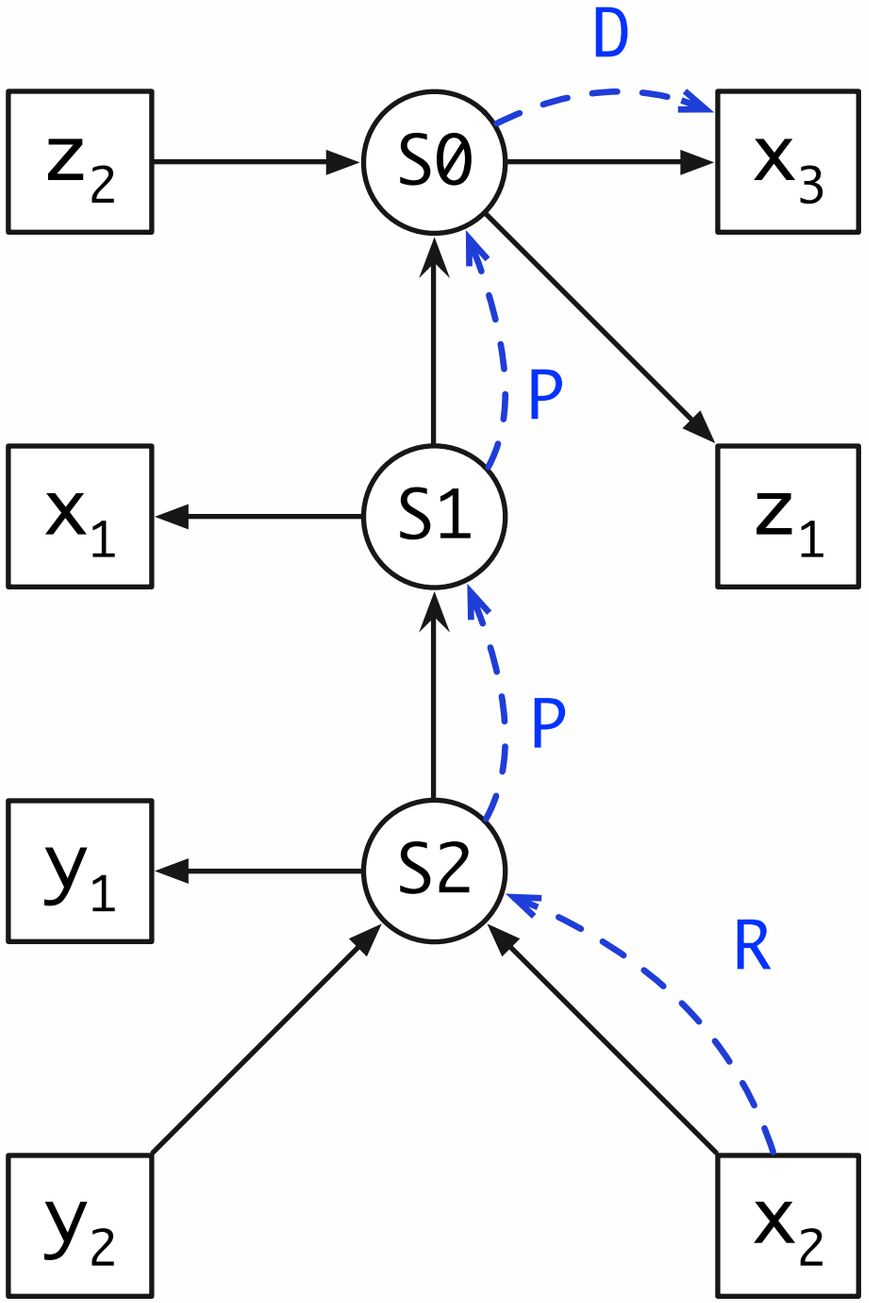
Shadowing

```
def x3 = z2 5 7                    S0

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



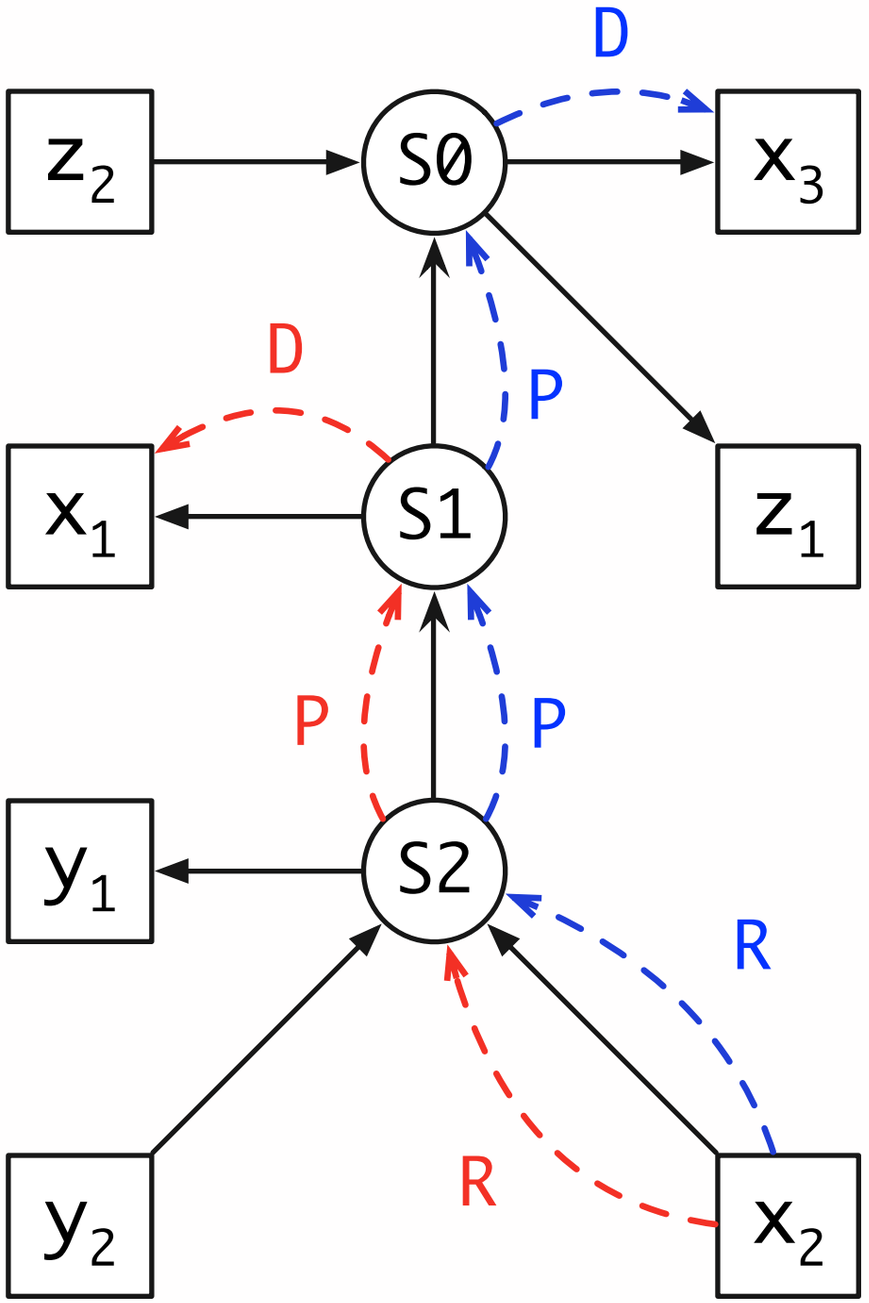
Shadowing

```
def x3 = z2 5 7                    S0

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



Shadowing

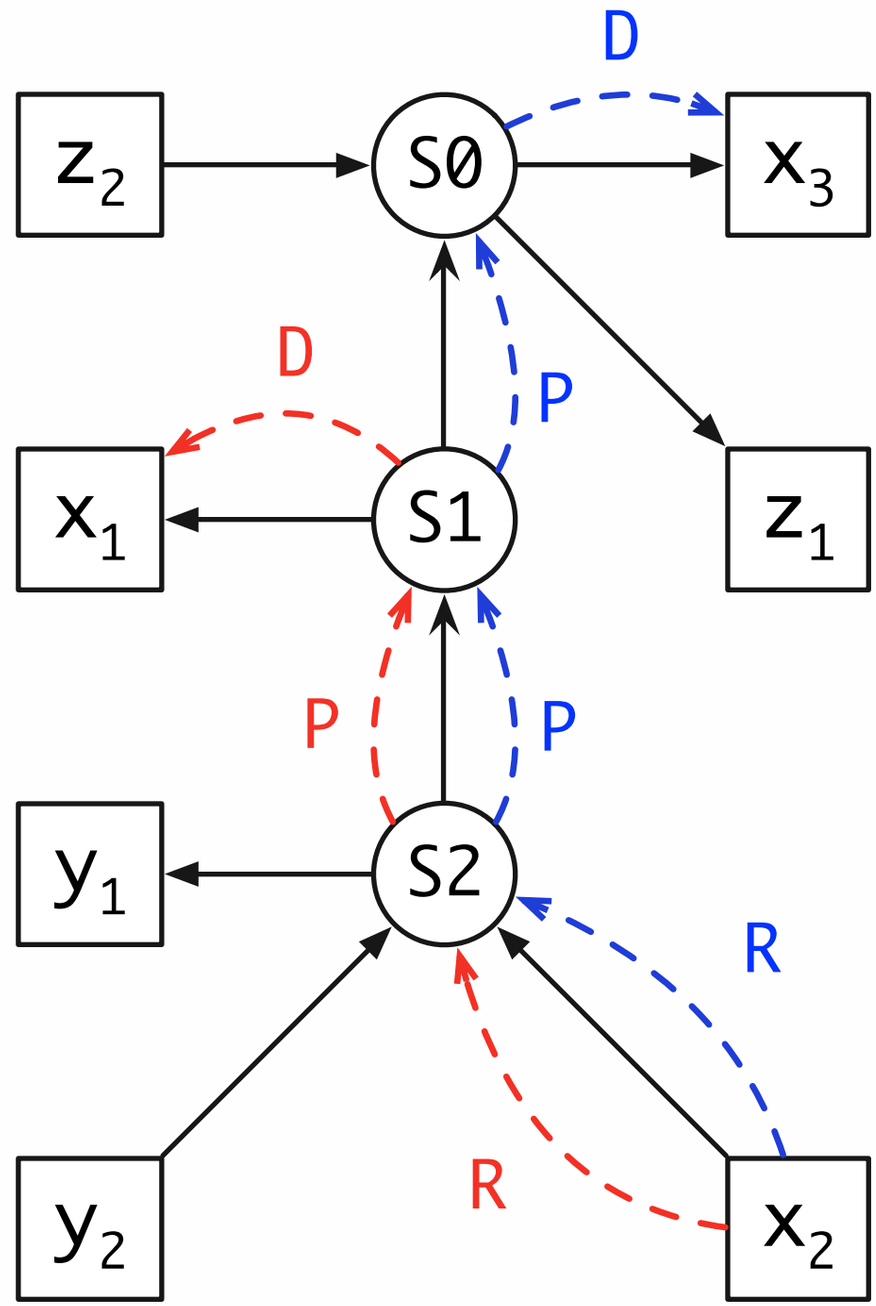
```
def x3 = z2 5 7                    S0

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



Shadowing

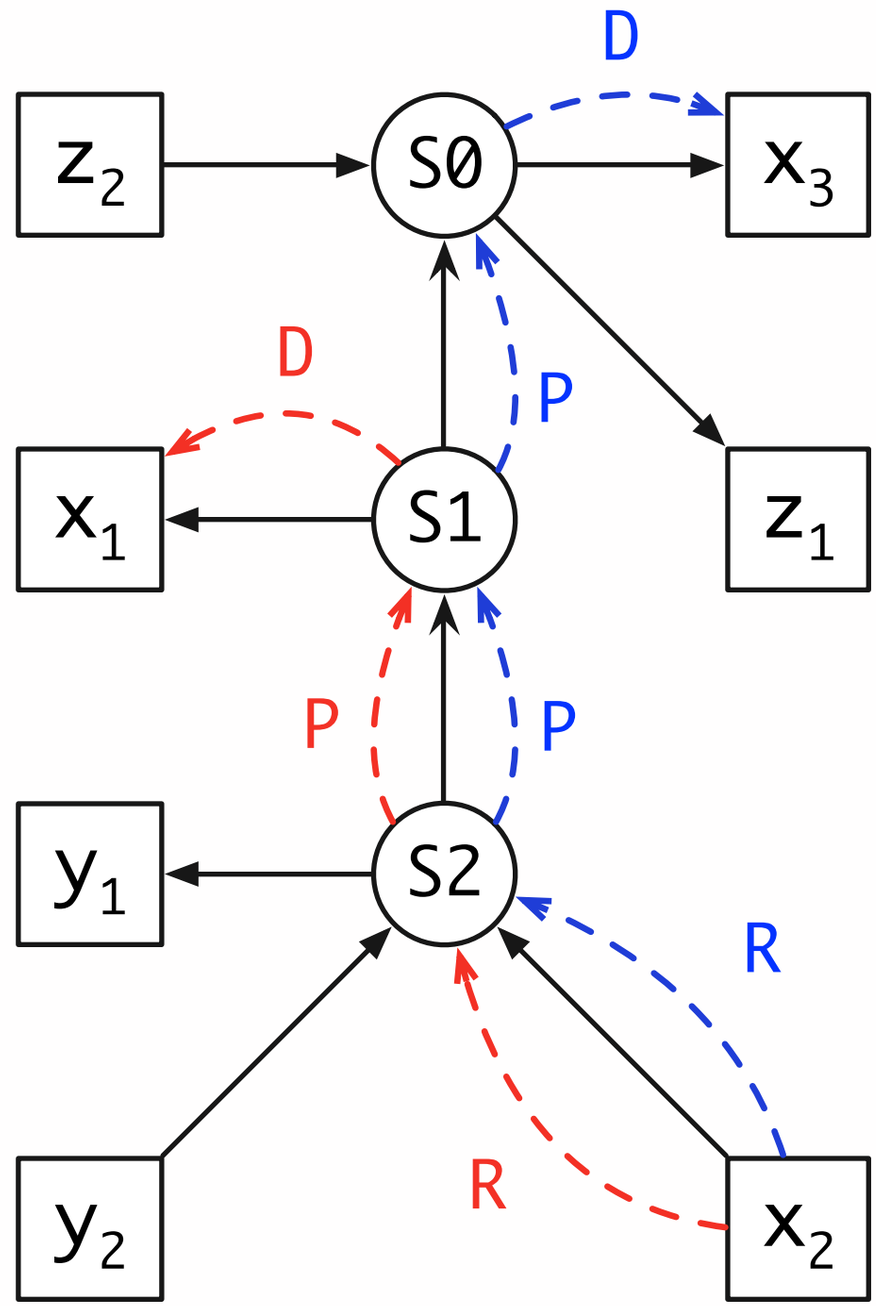
```
def x3 = z2 5 7                    S0

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
    }
  }
```

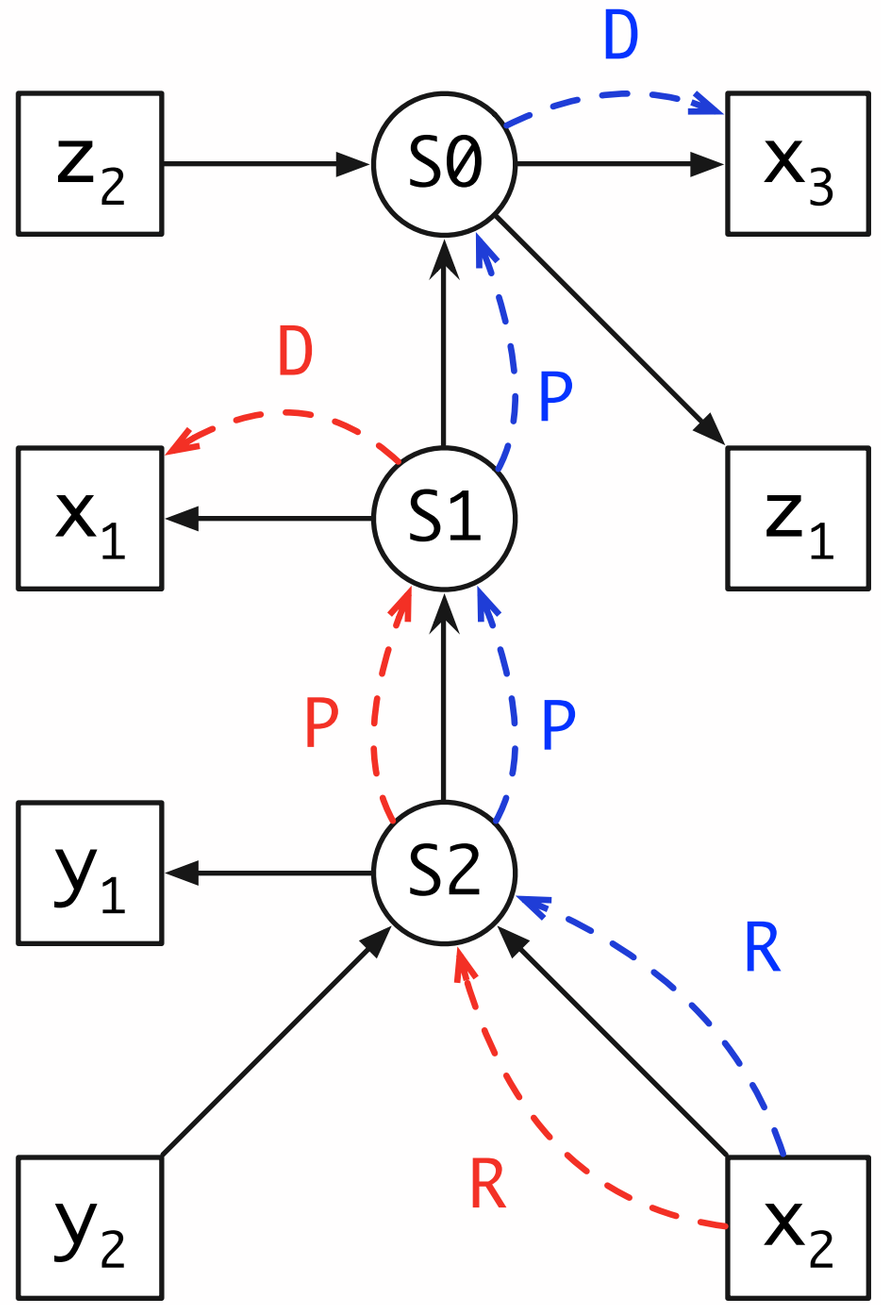


$$\frac{}{D < P.p}$$

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

Shadowing

```
def x3 = z2 5 7                                   S0

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



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

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

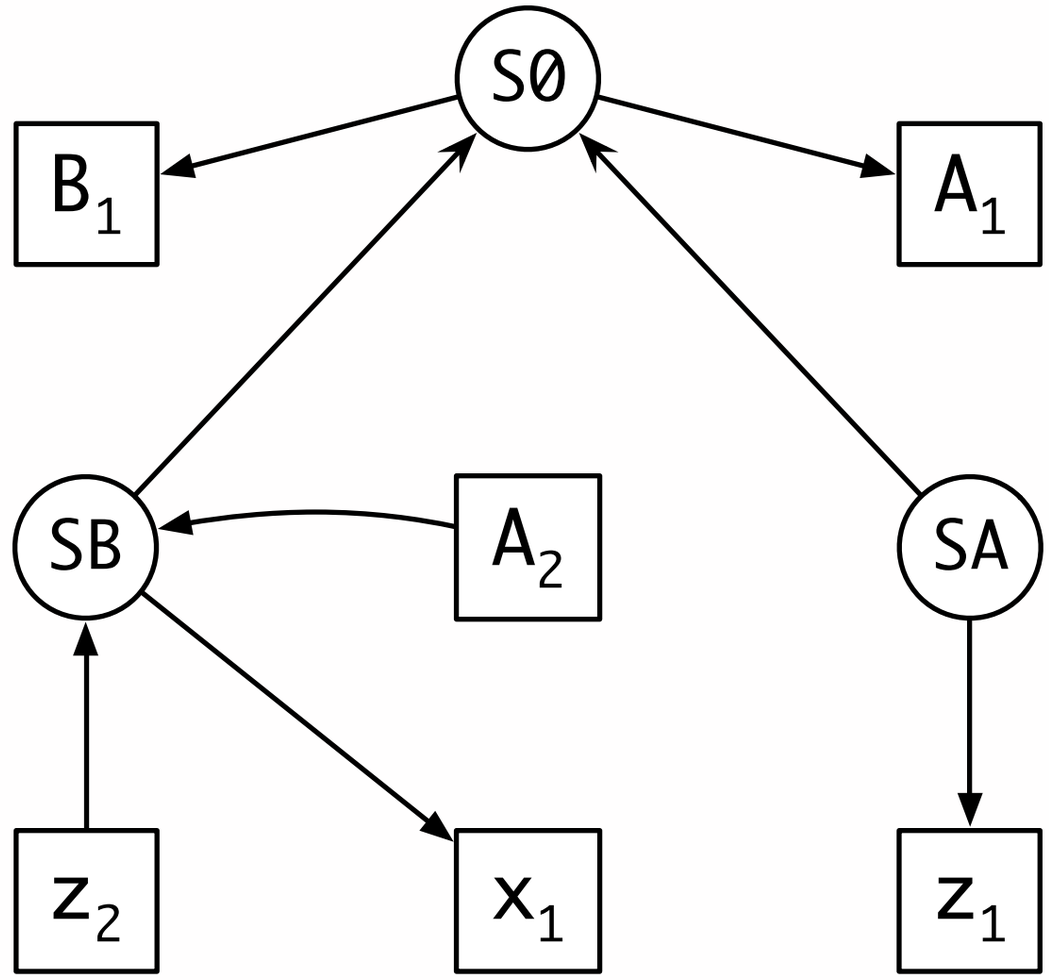
$$R.P.D < R.P.P.D$$

Imports

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

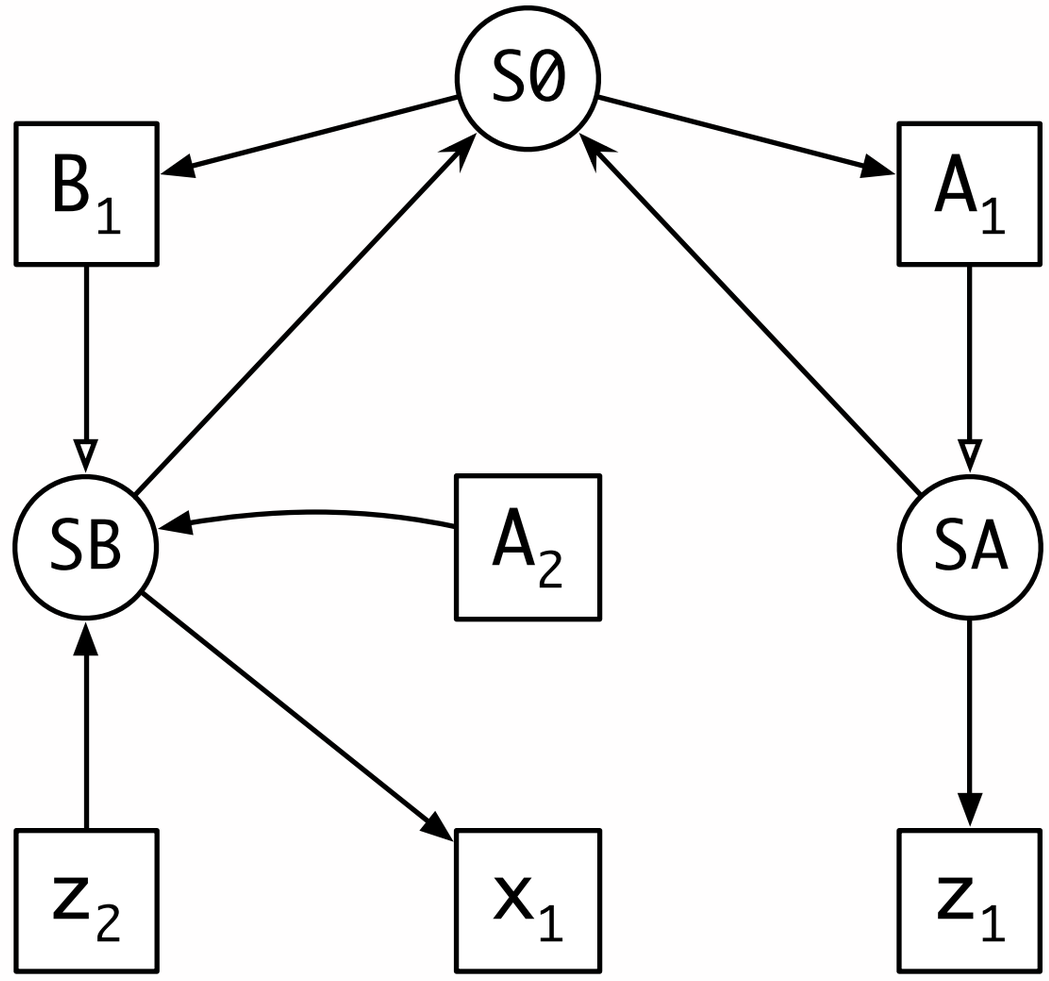
Imports

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



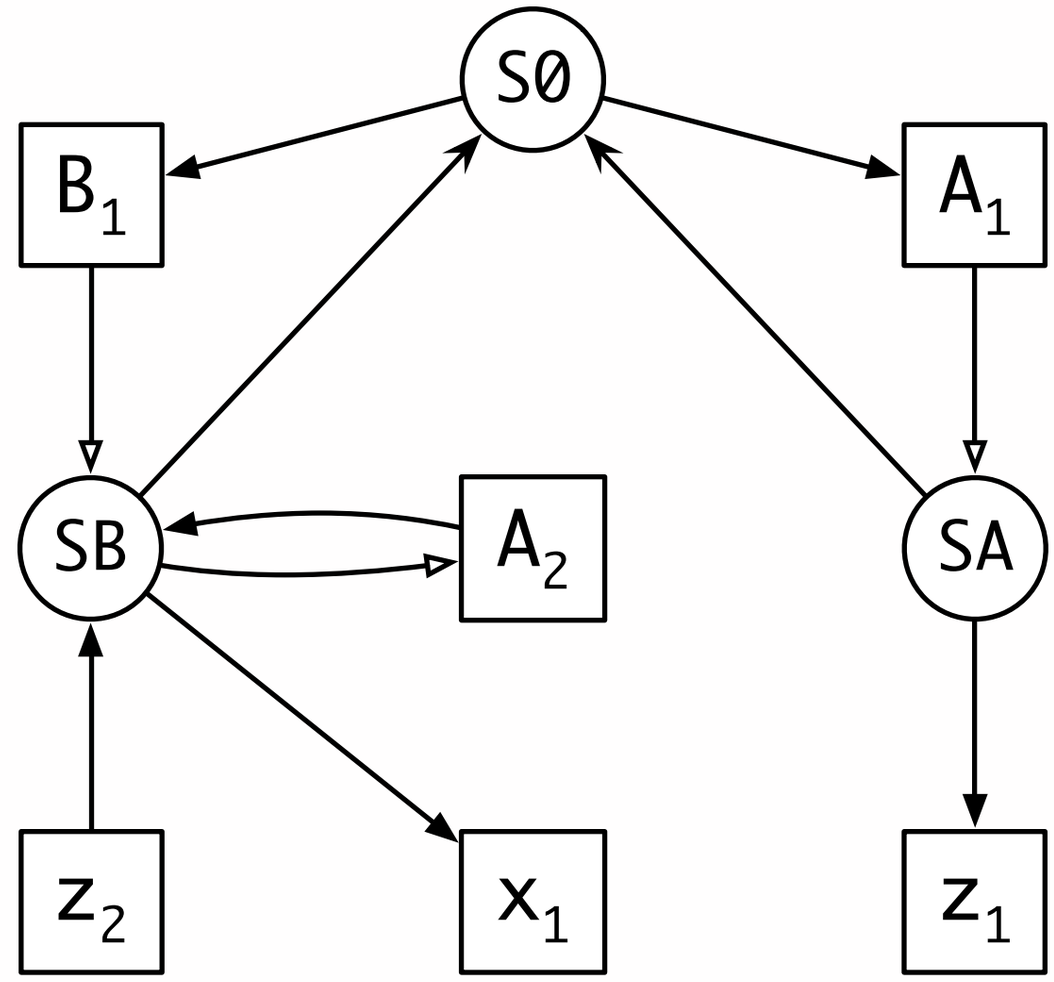
Imports

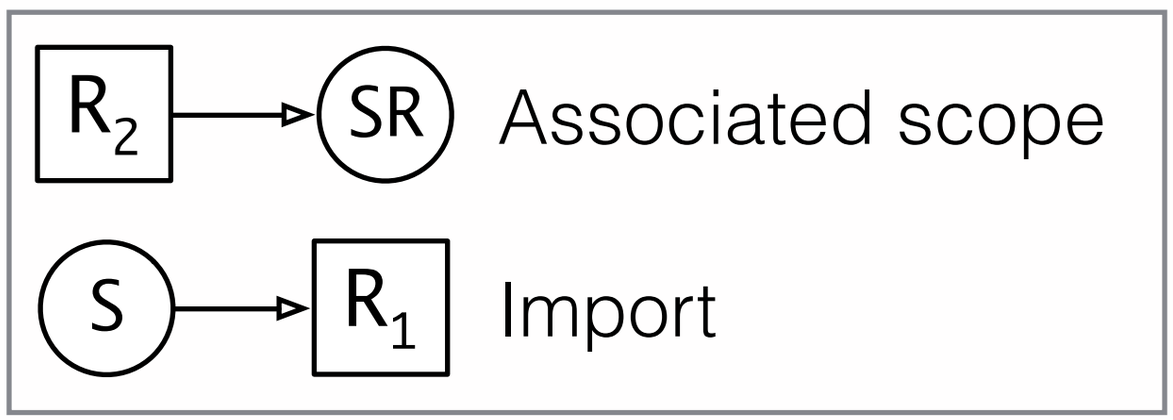
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```



Imports

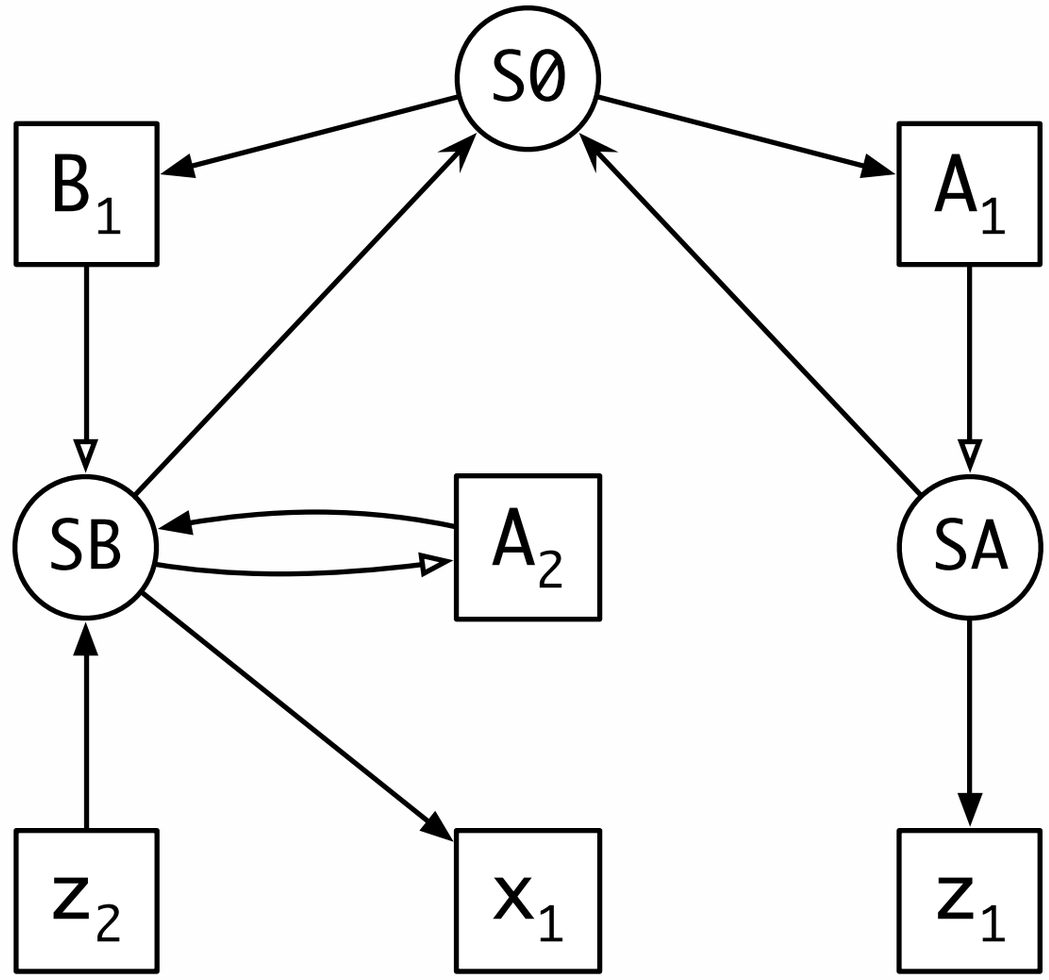
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

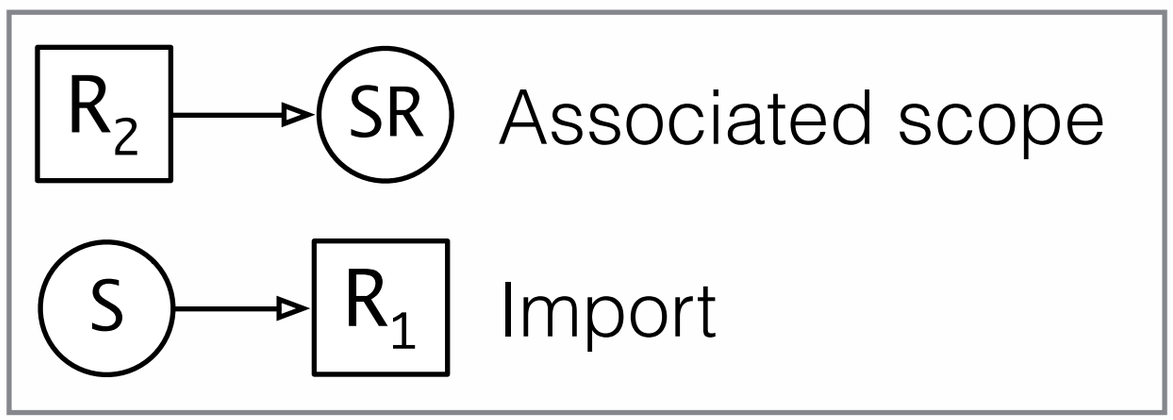




Imports

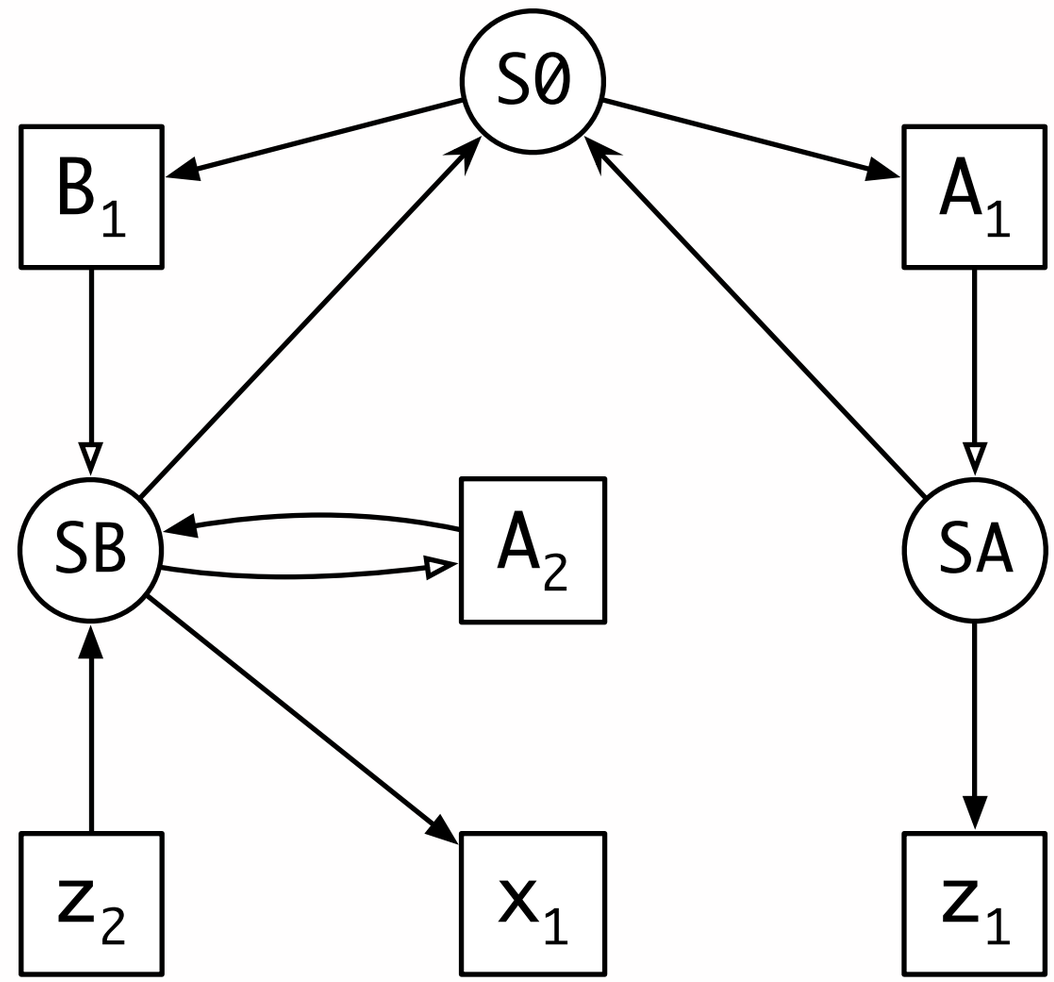
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

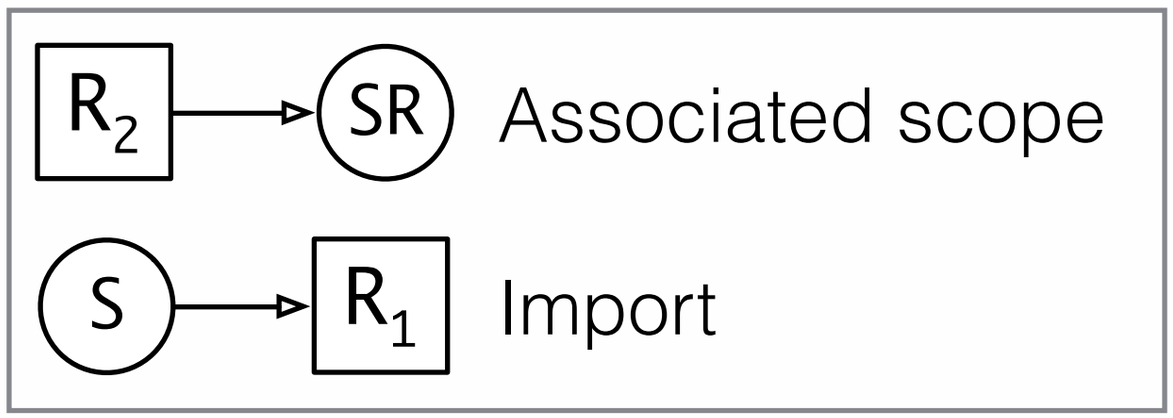




Imports

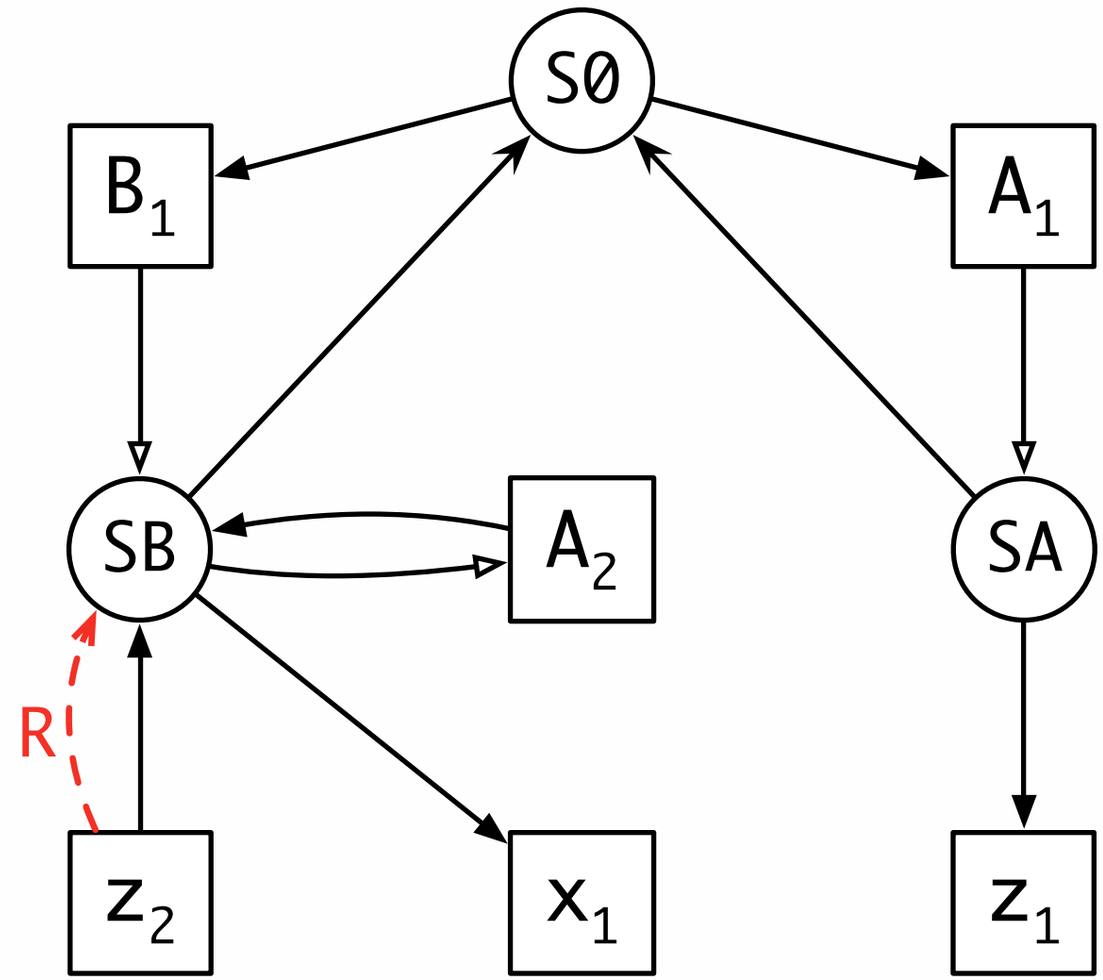
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

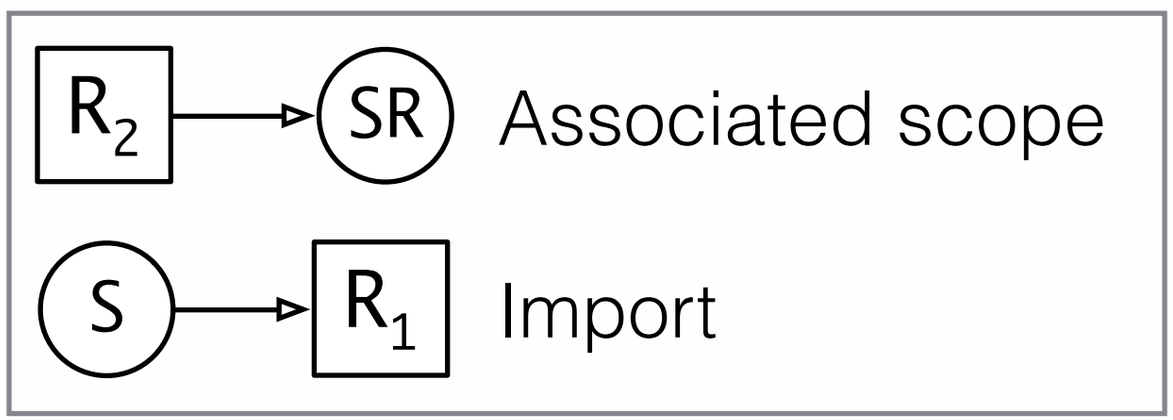




Imports

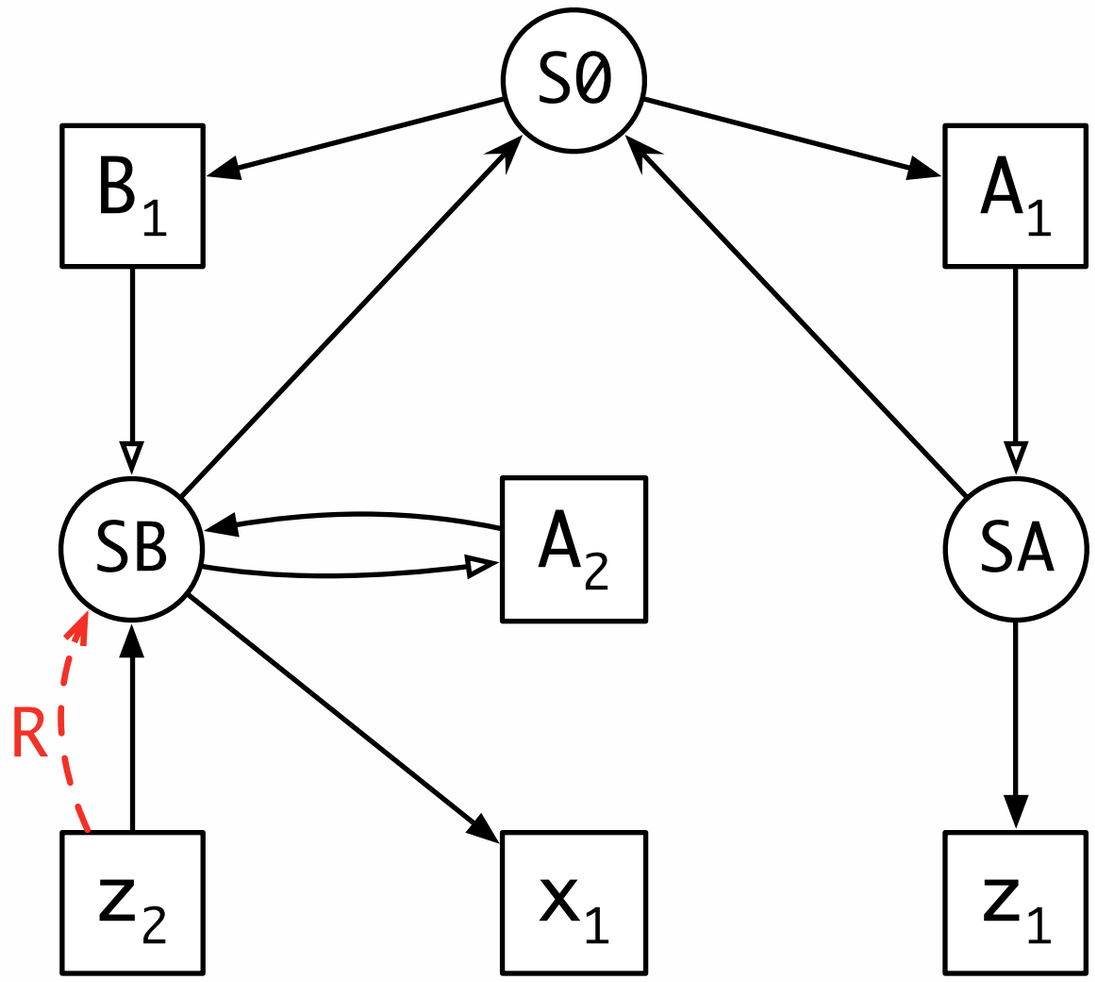
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

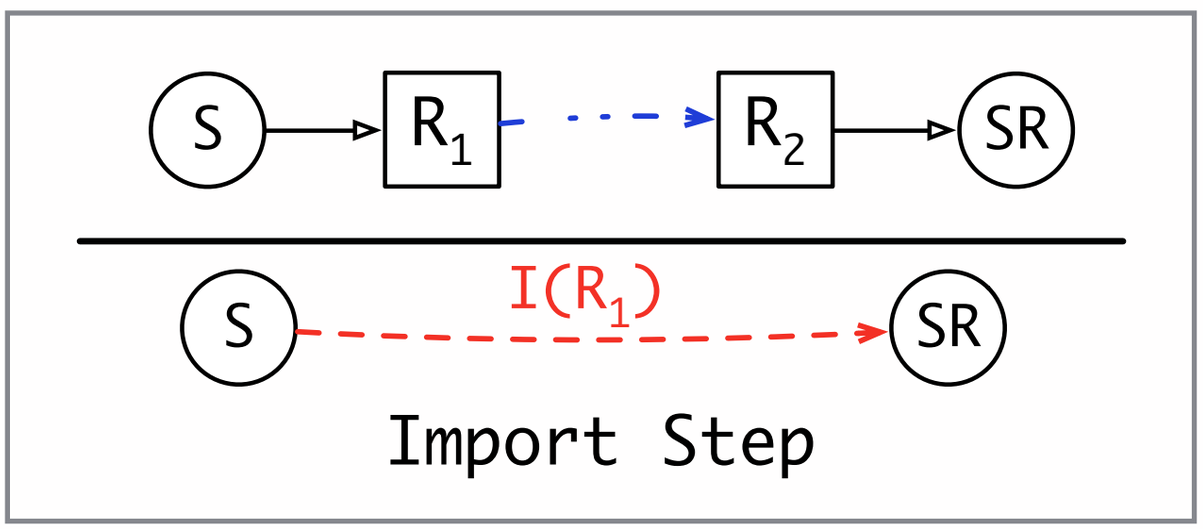




Imports

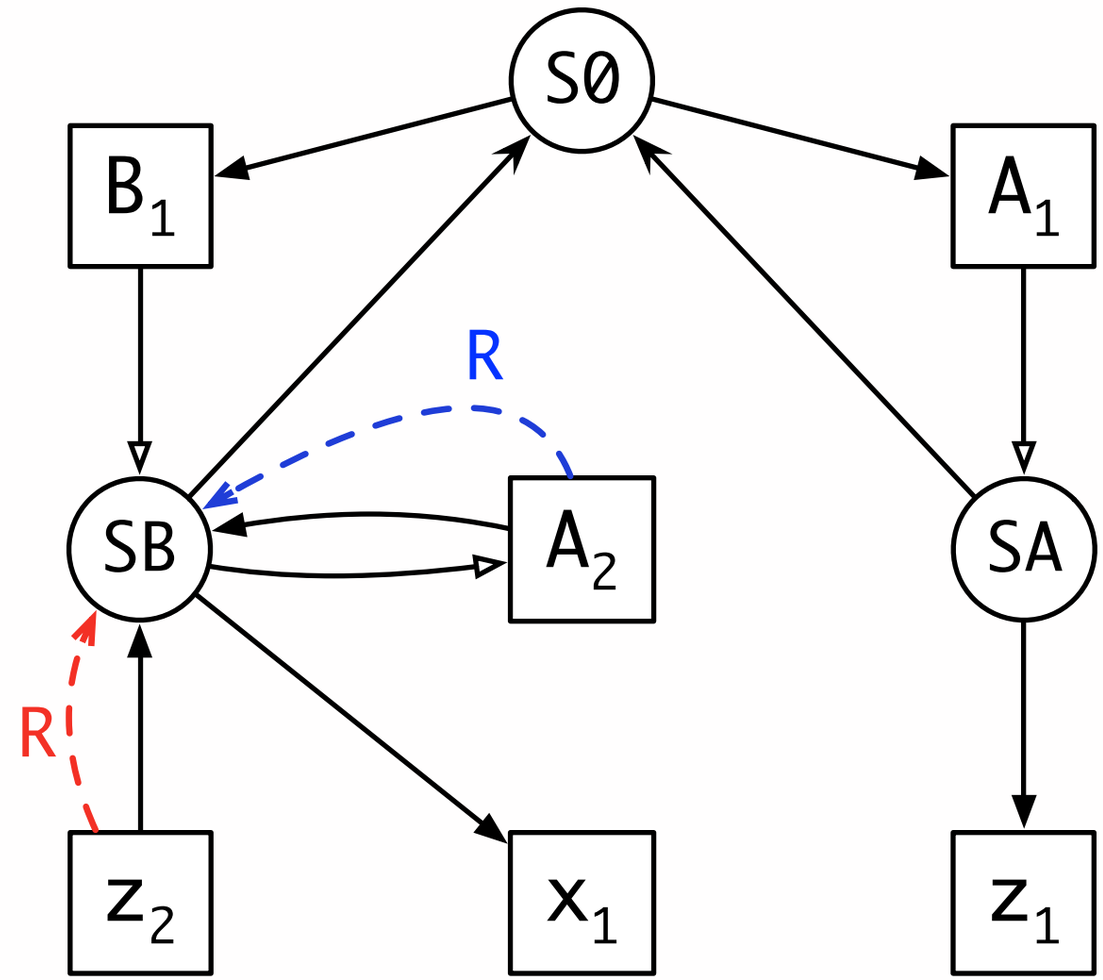
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

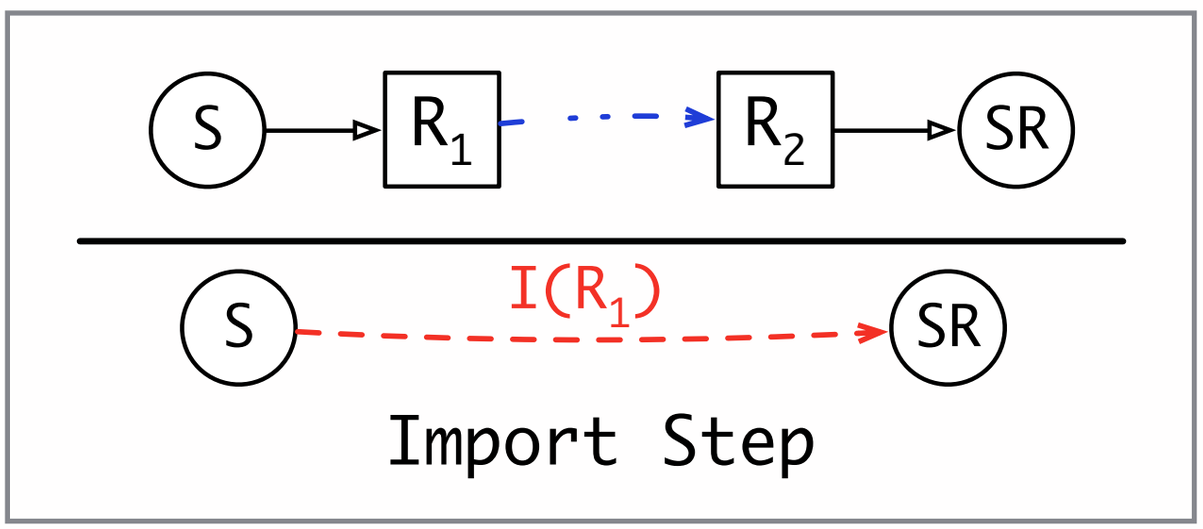




Imports

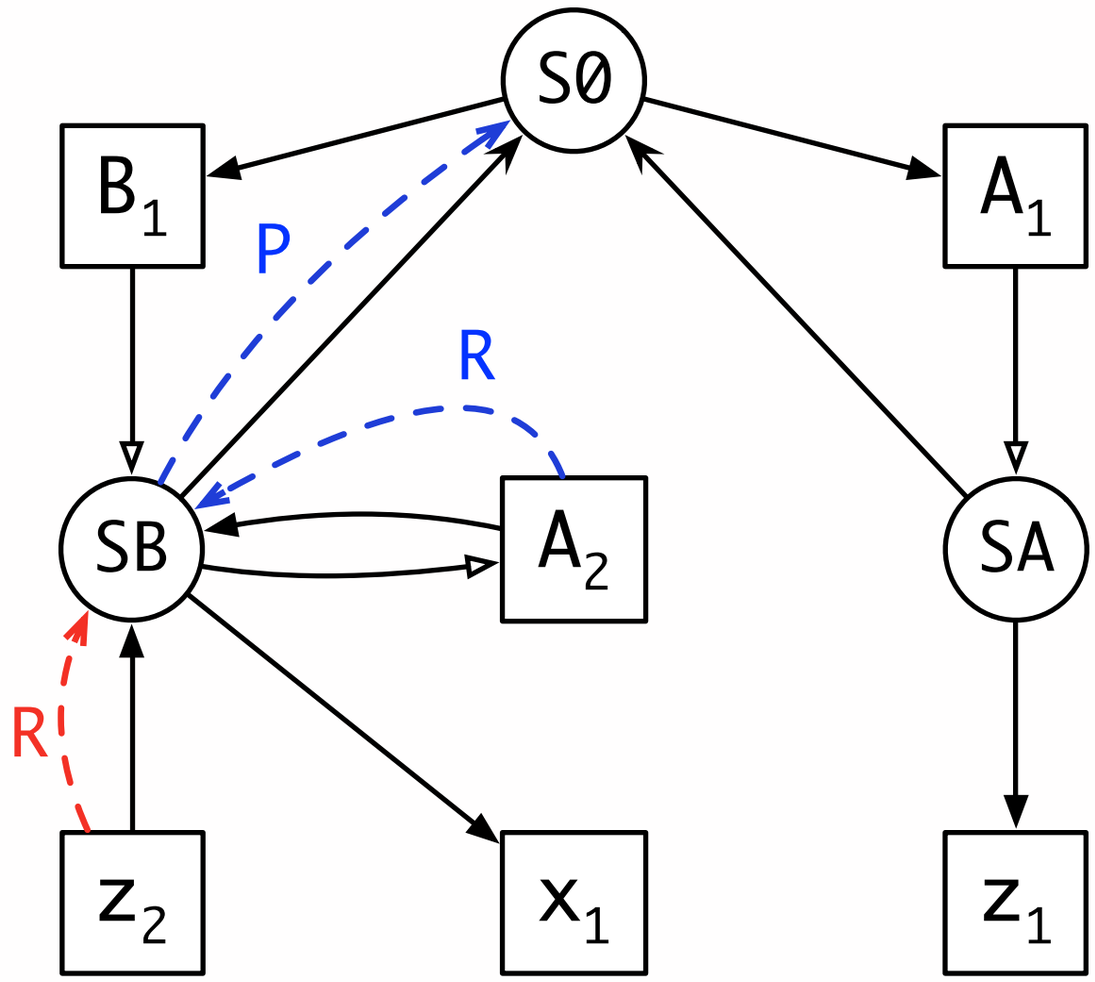
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

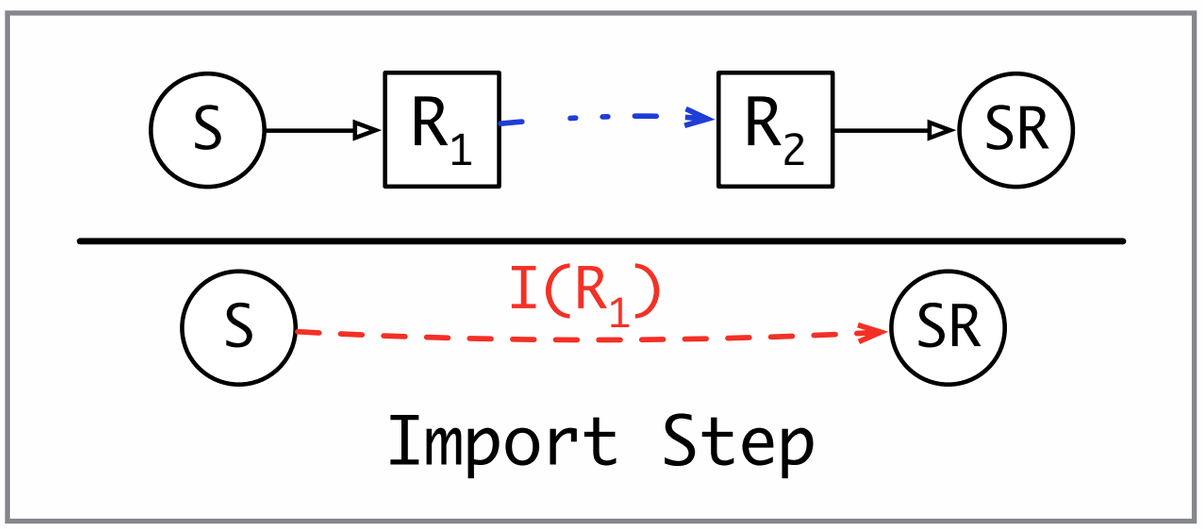




Imports

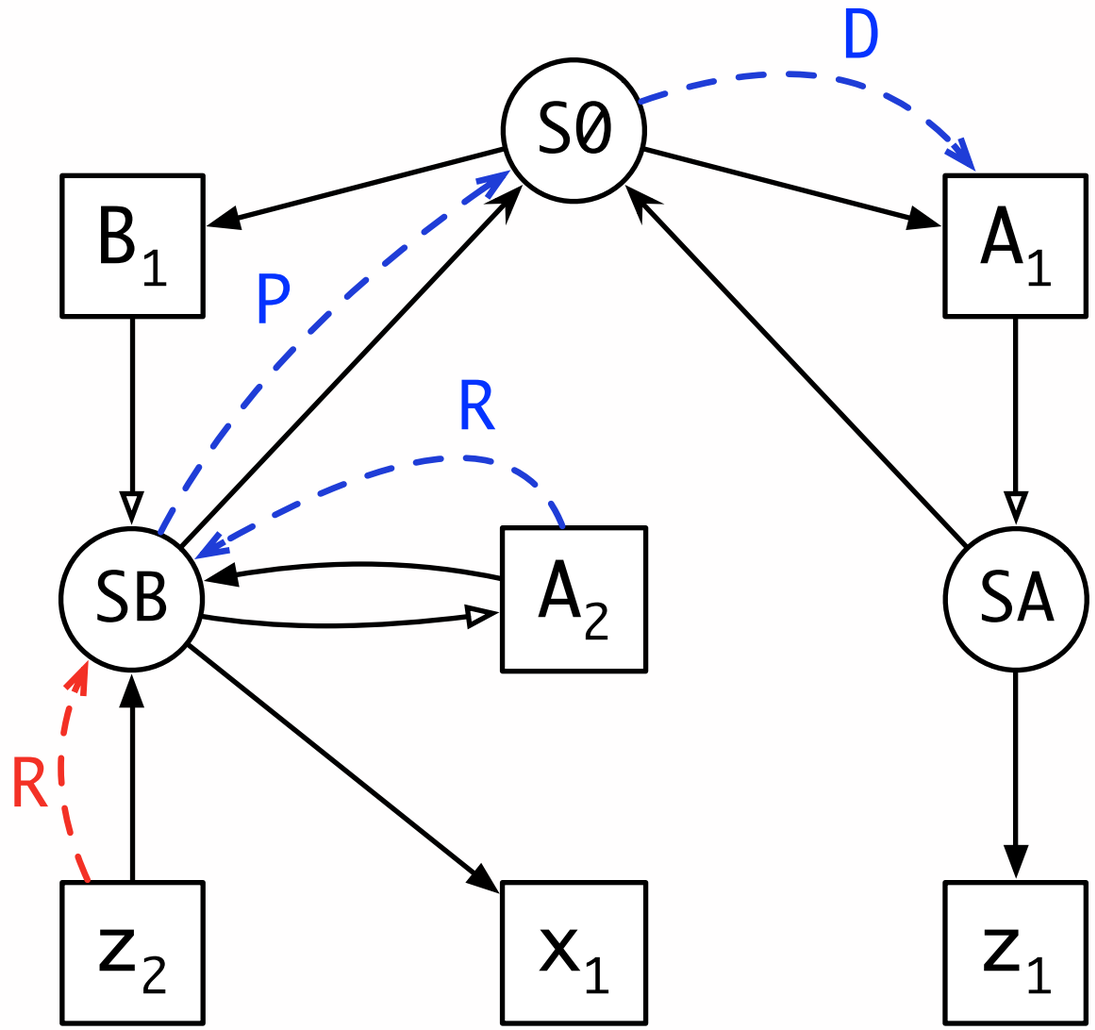
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

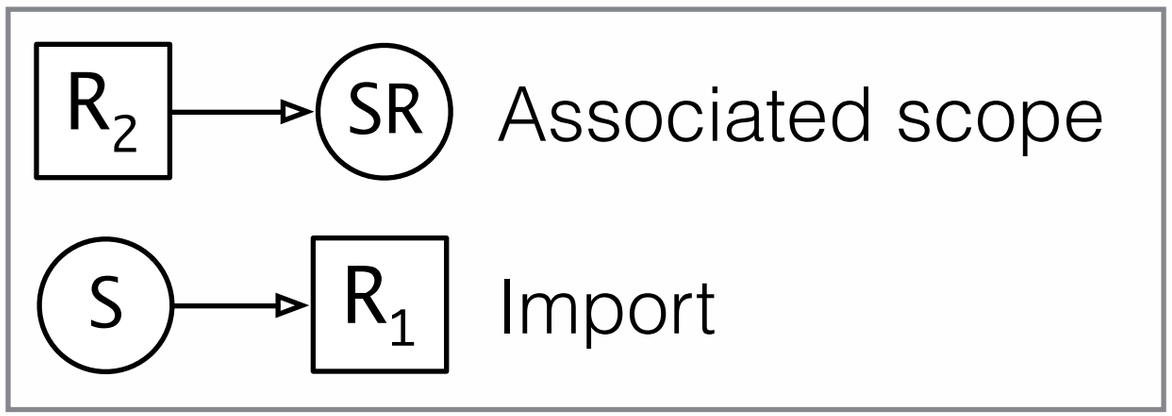


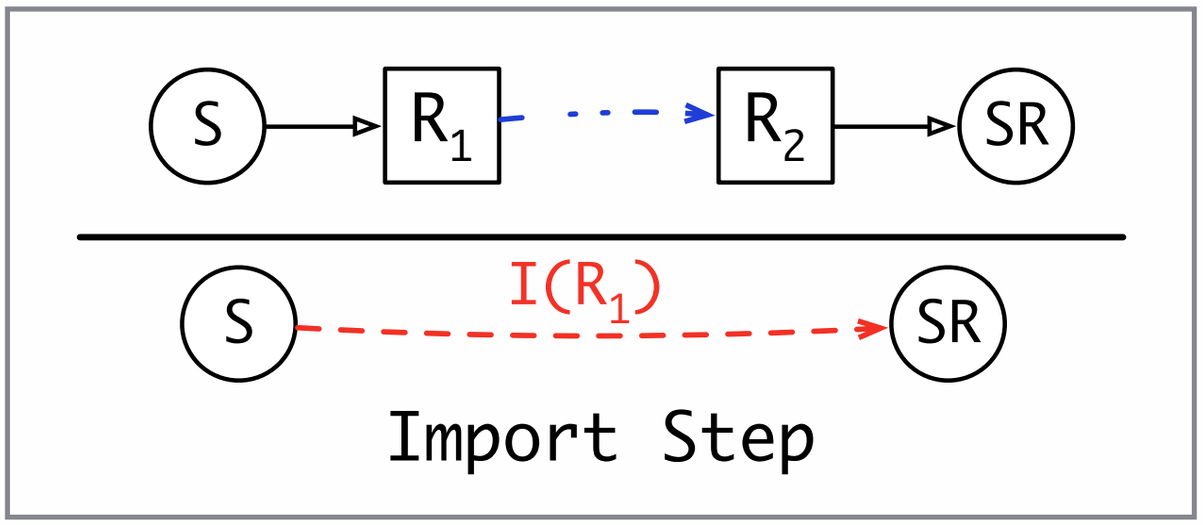


Imports

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

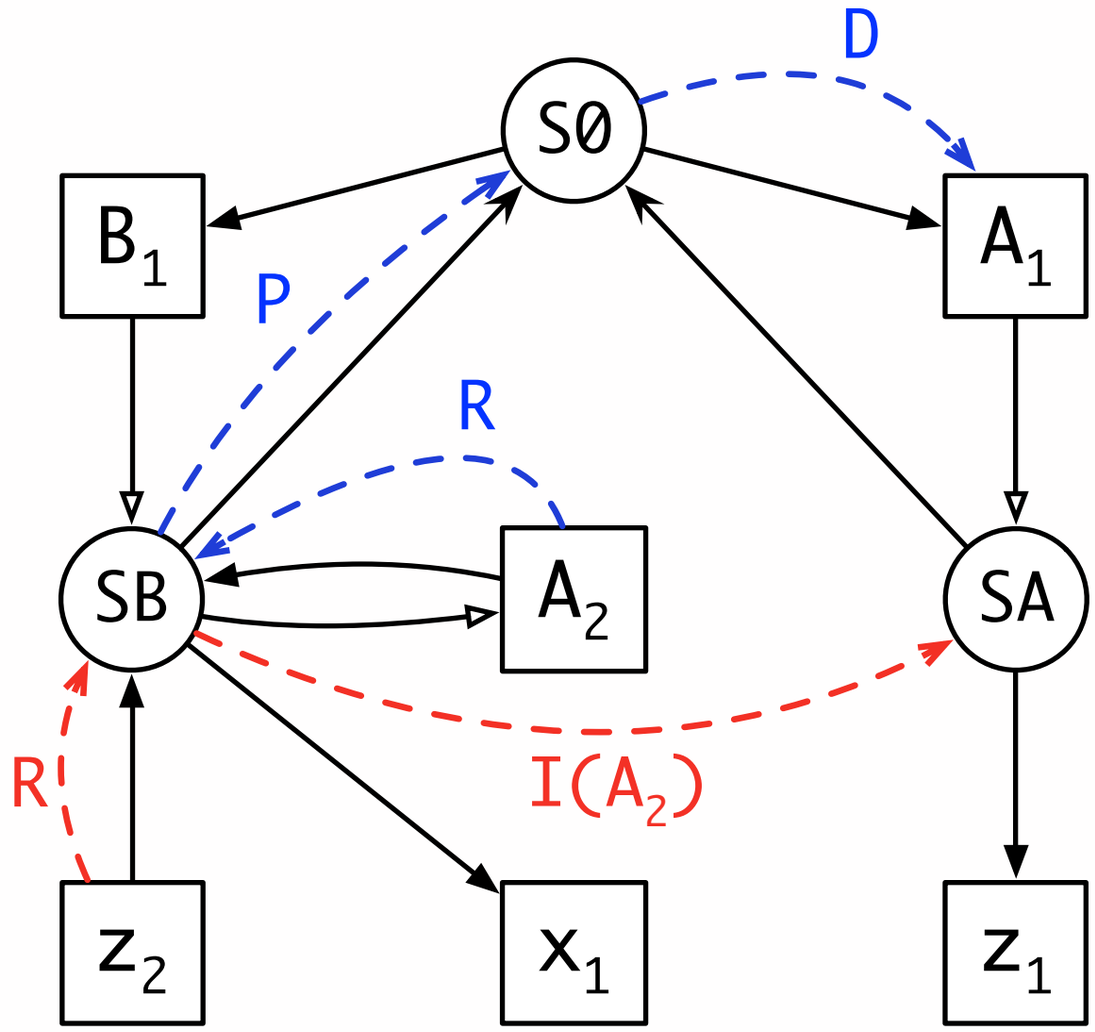


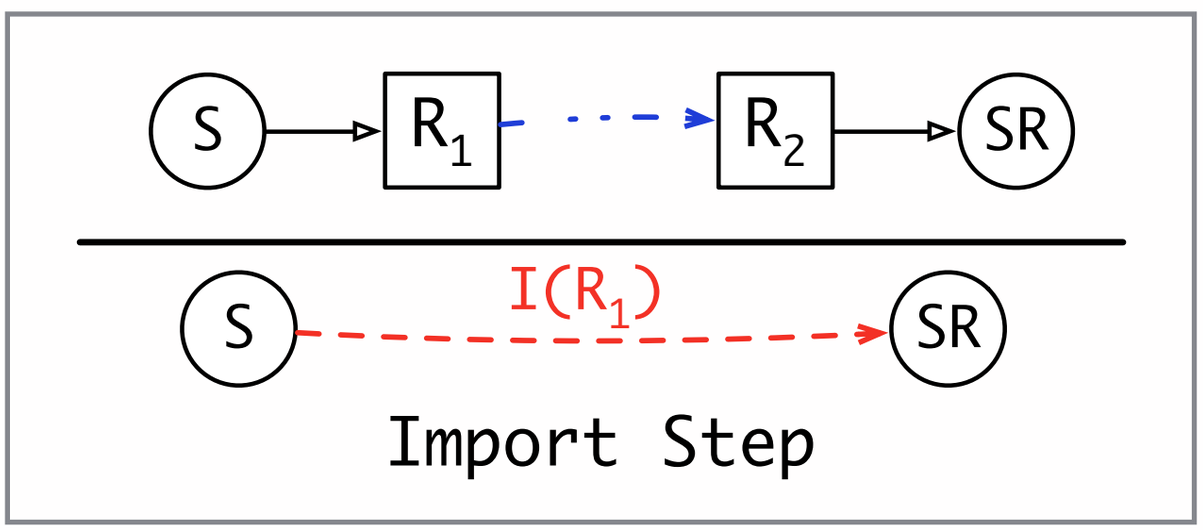




Imports

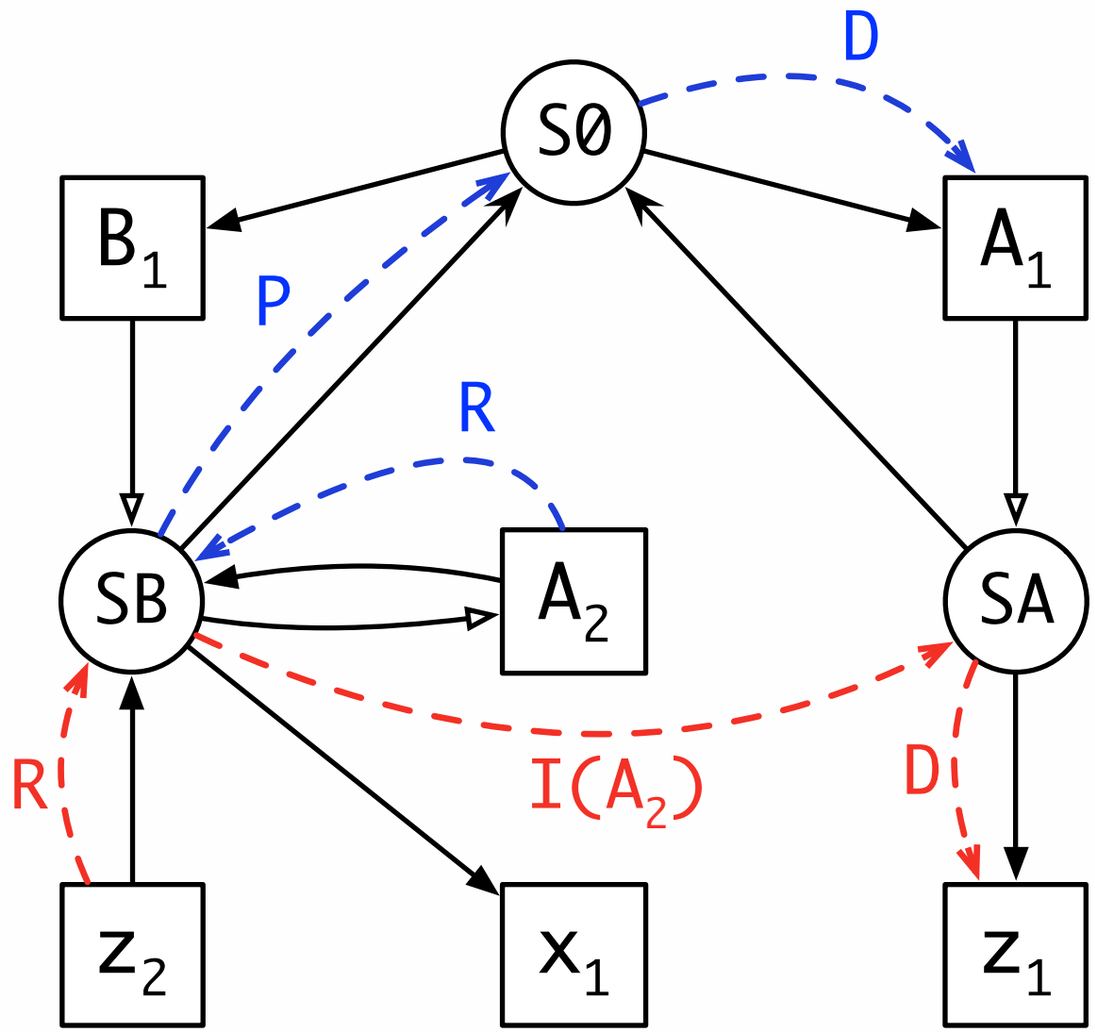
```
module A1 {                    S0
  def z1 = 5                   SA
}
module B1 {
  import A2                    SB
  def x1 = 1 + z2
}
```

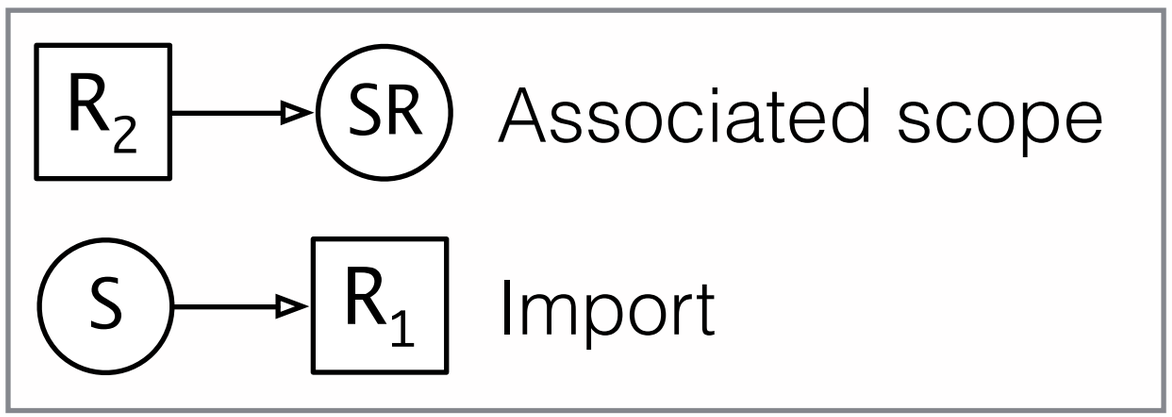


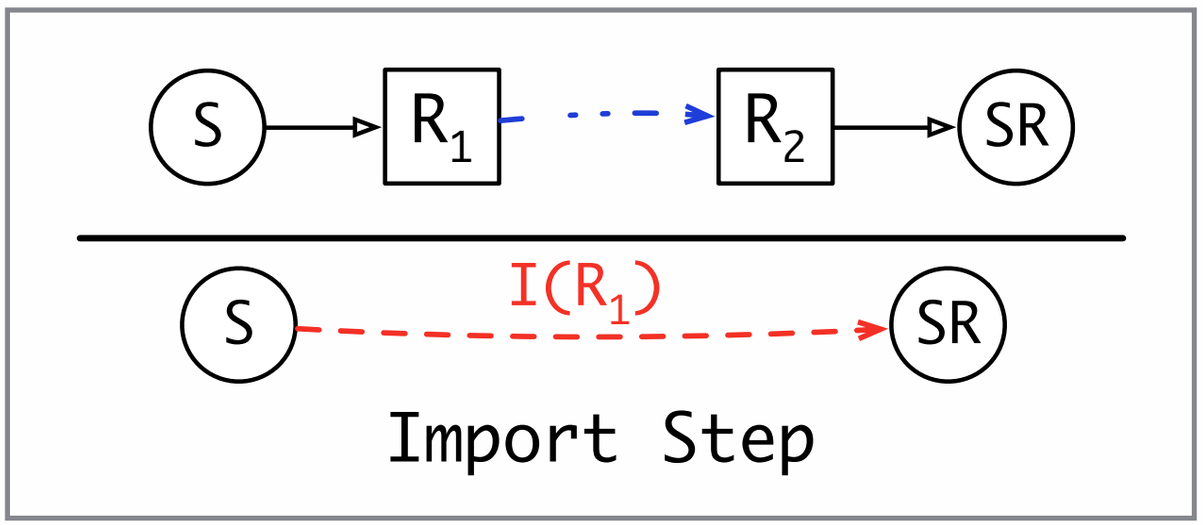


Imports

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





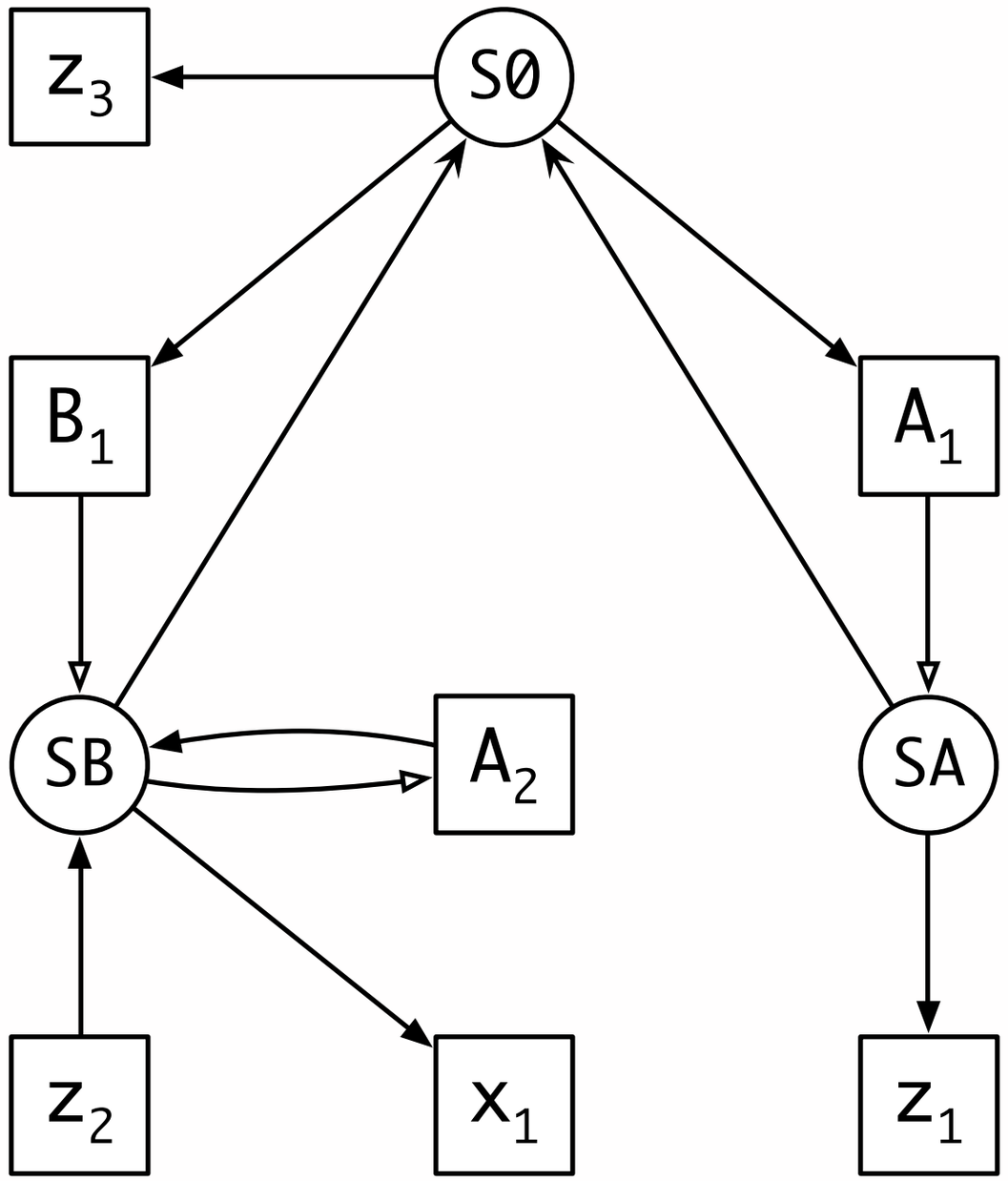


Imports shadow Parents

```
def z3 = 2                                  S0

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

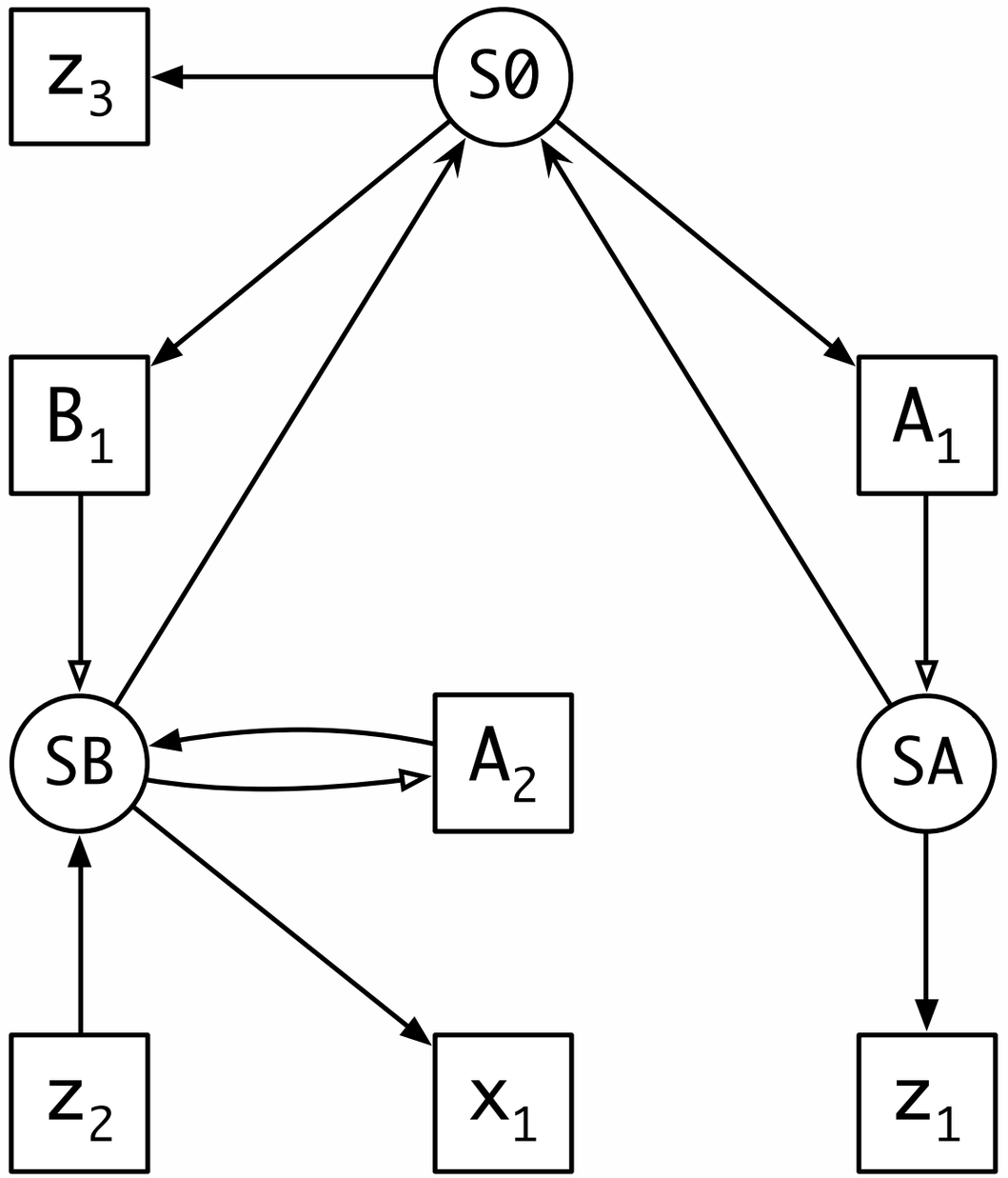
Imports shadow Parents

```
def z₃ = 2                                  S0

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



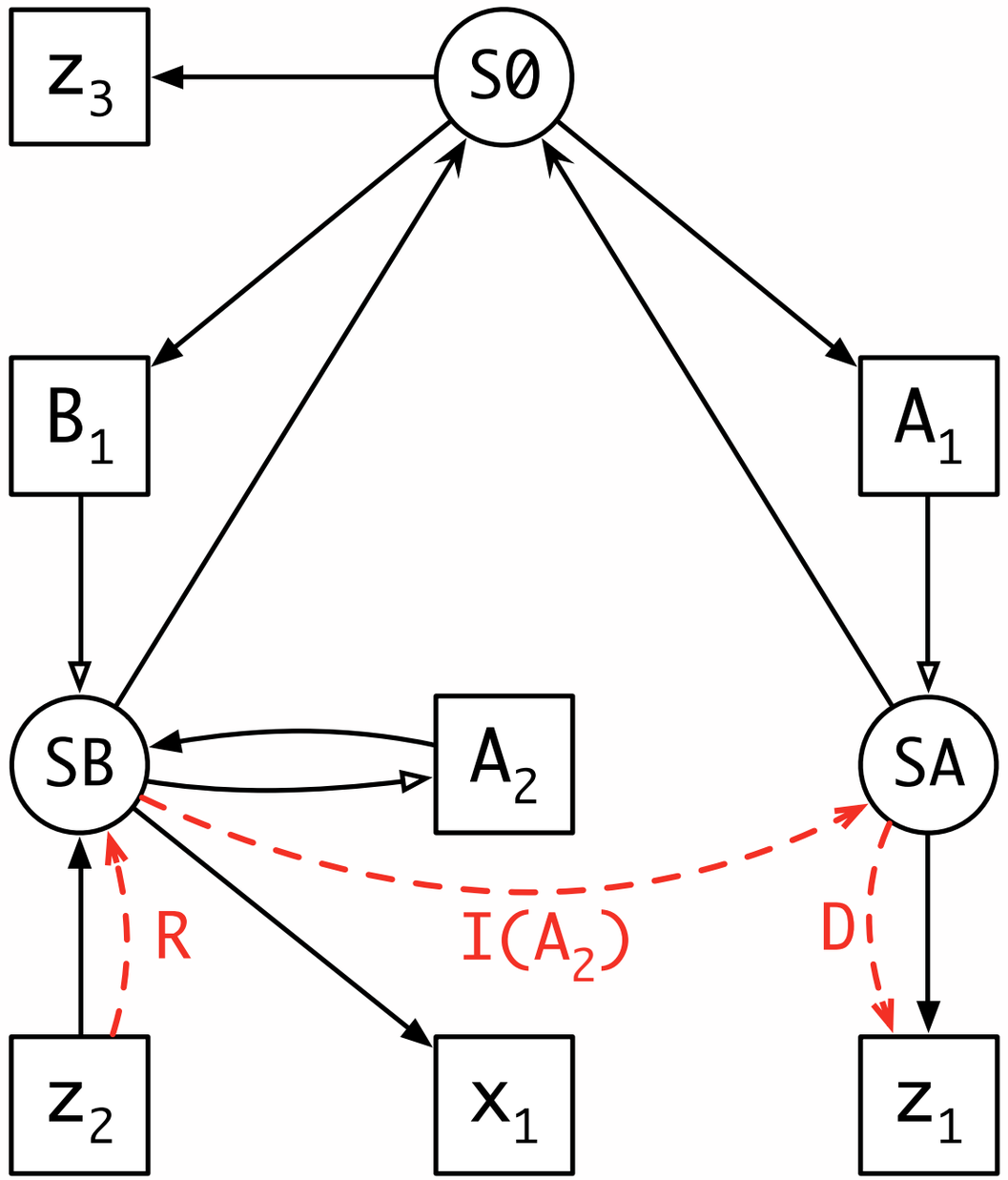
Imports shadow Parents

```
def z₃ = 2                                  S0

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



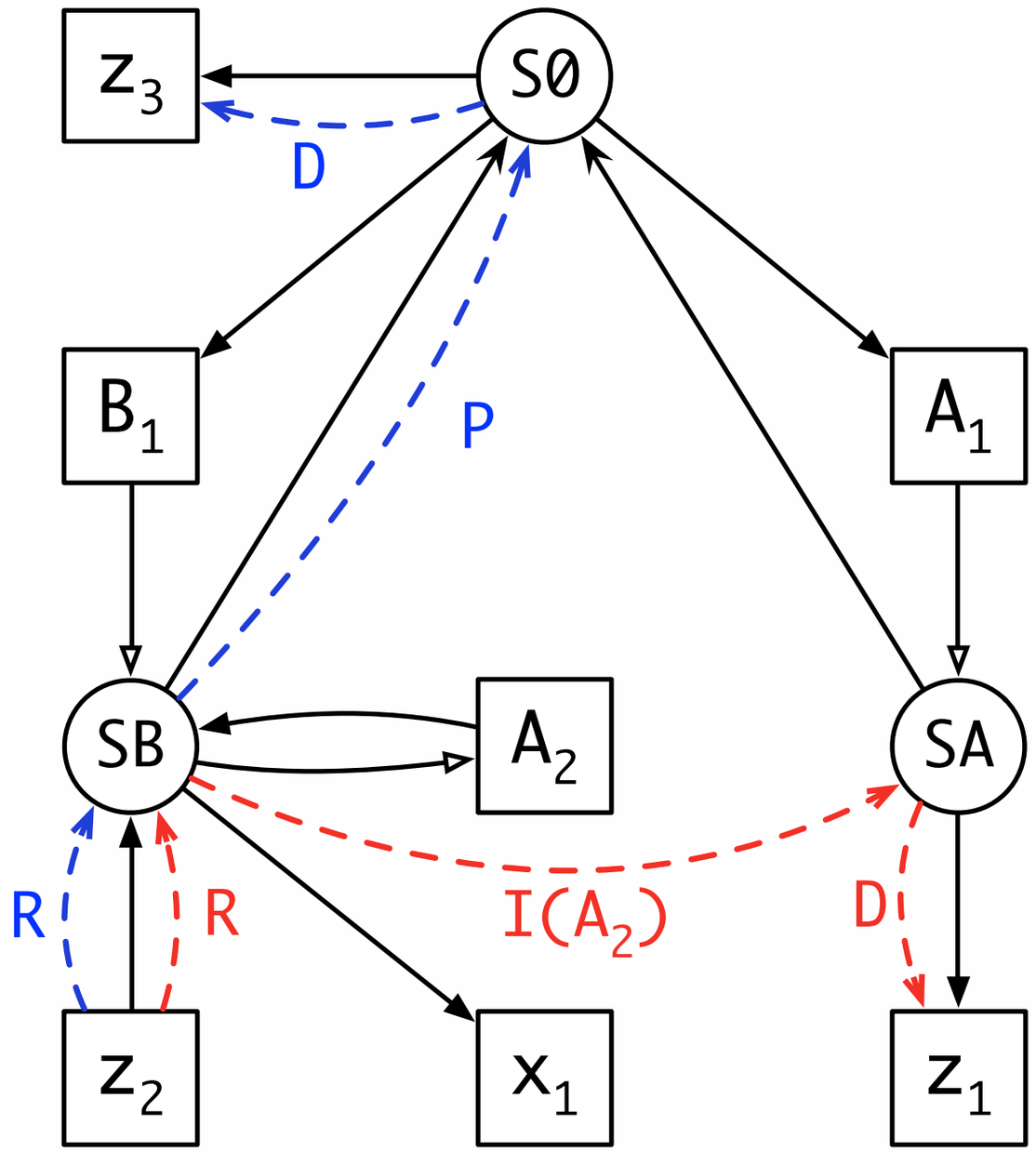
Imports shadow Parents

```
def z3 = 2                                  S0

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



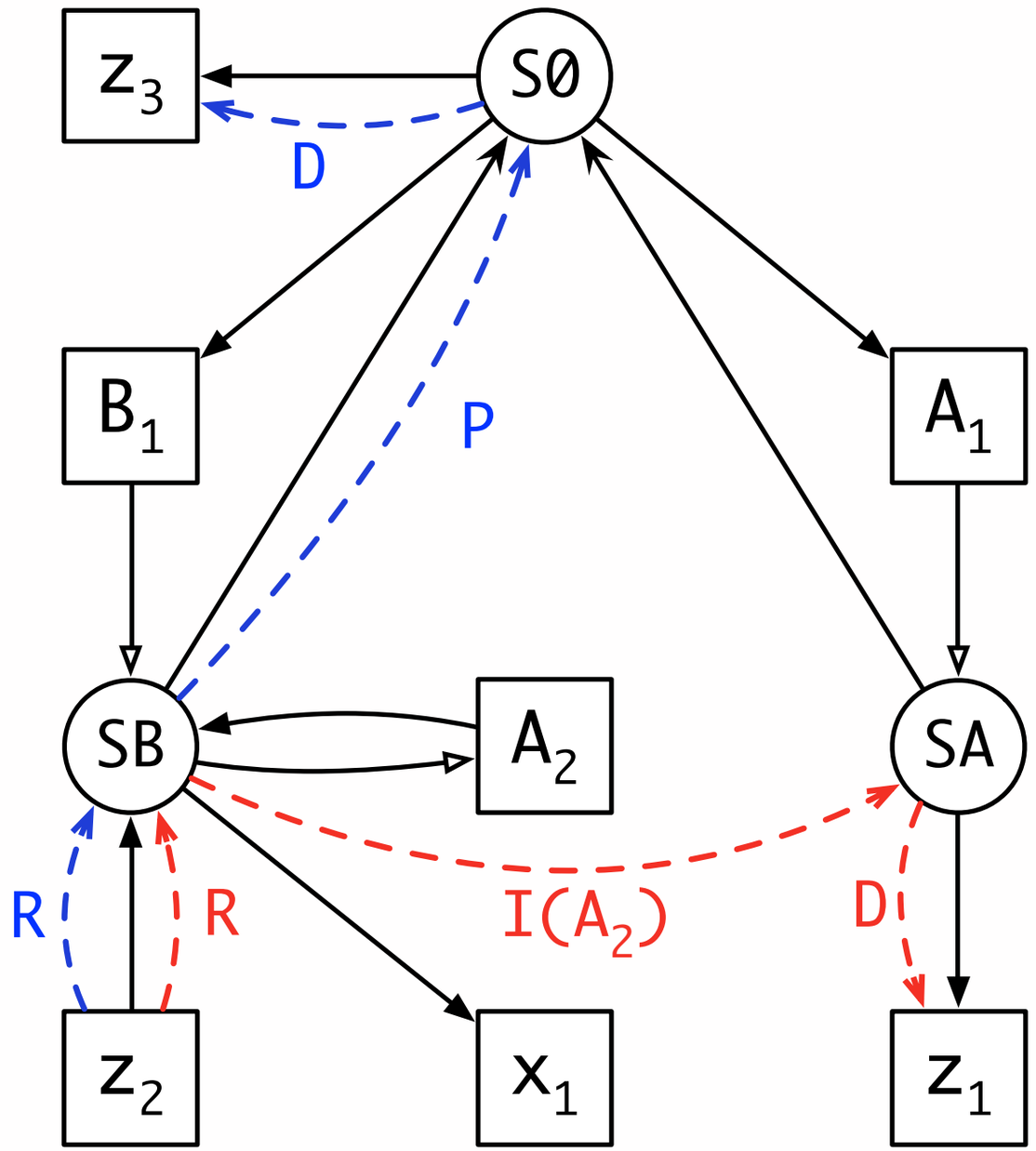
Imports shadow Parents

```
def z₃ = 2                                  S0

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



Imports shadow Parents

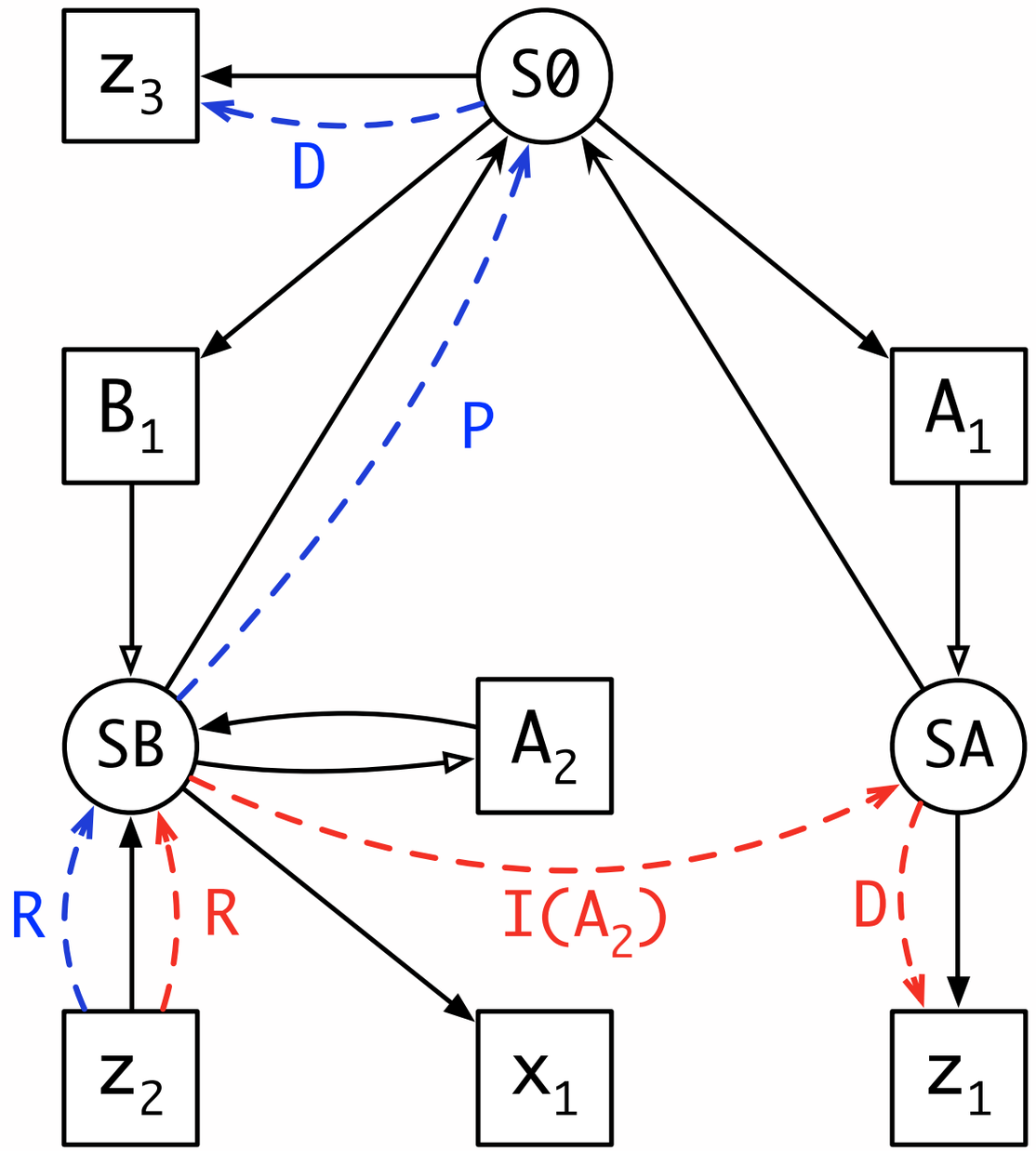
```
def z3 = 2                                  S0

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



I(_).p' < P.p

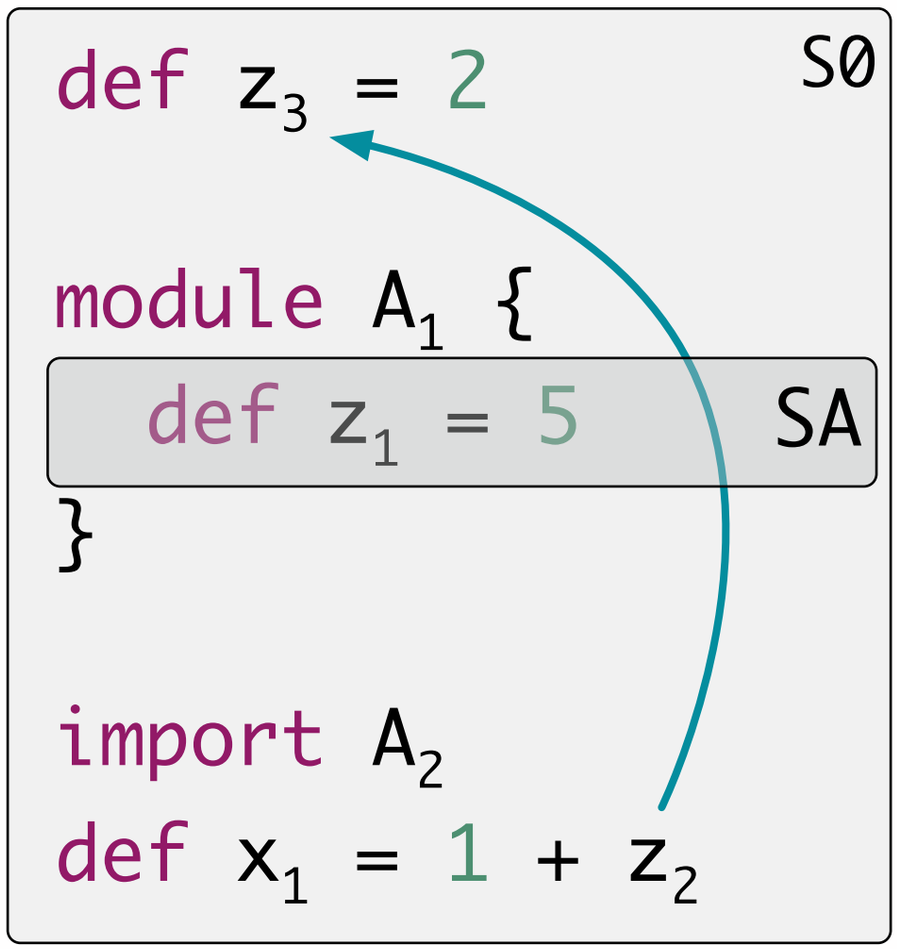
Imports shadow Parents

```
def z3 = 2                                  S0

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



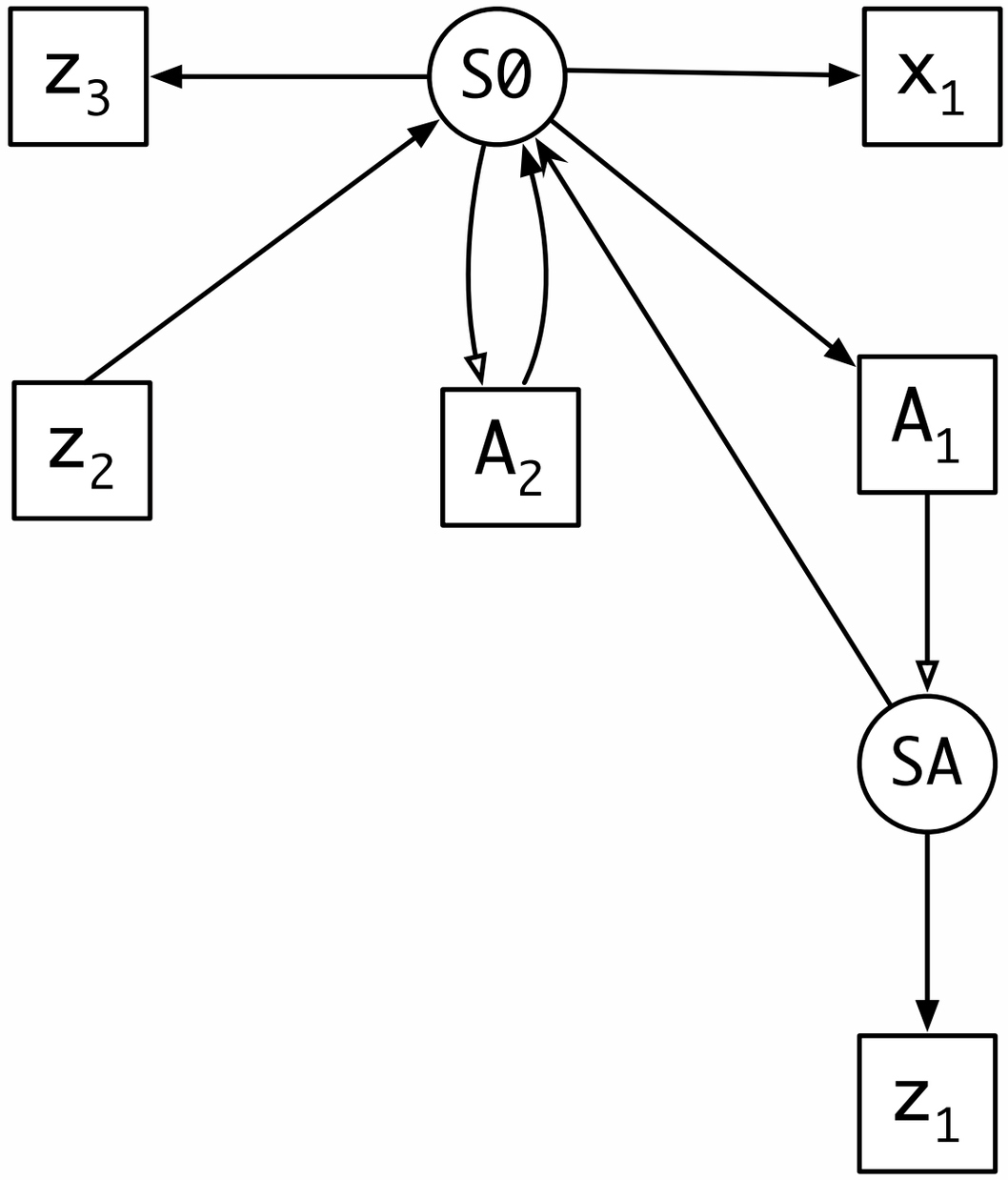
$$\frac{}{I(\_).p' < P.p} \quad \Rightarrow \quad R.I(A_2).D < R.P.D$$

Imports vs. Includes



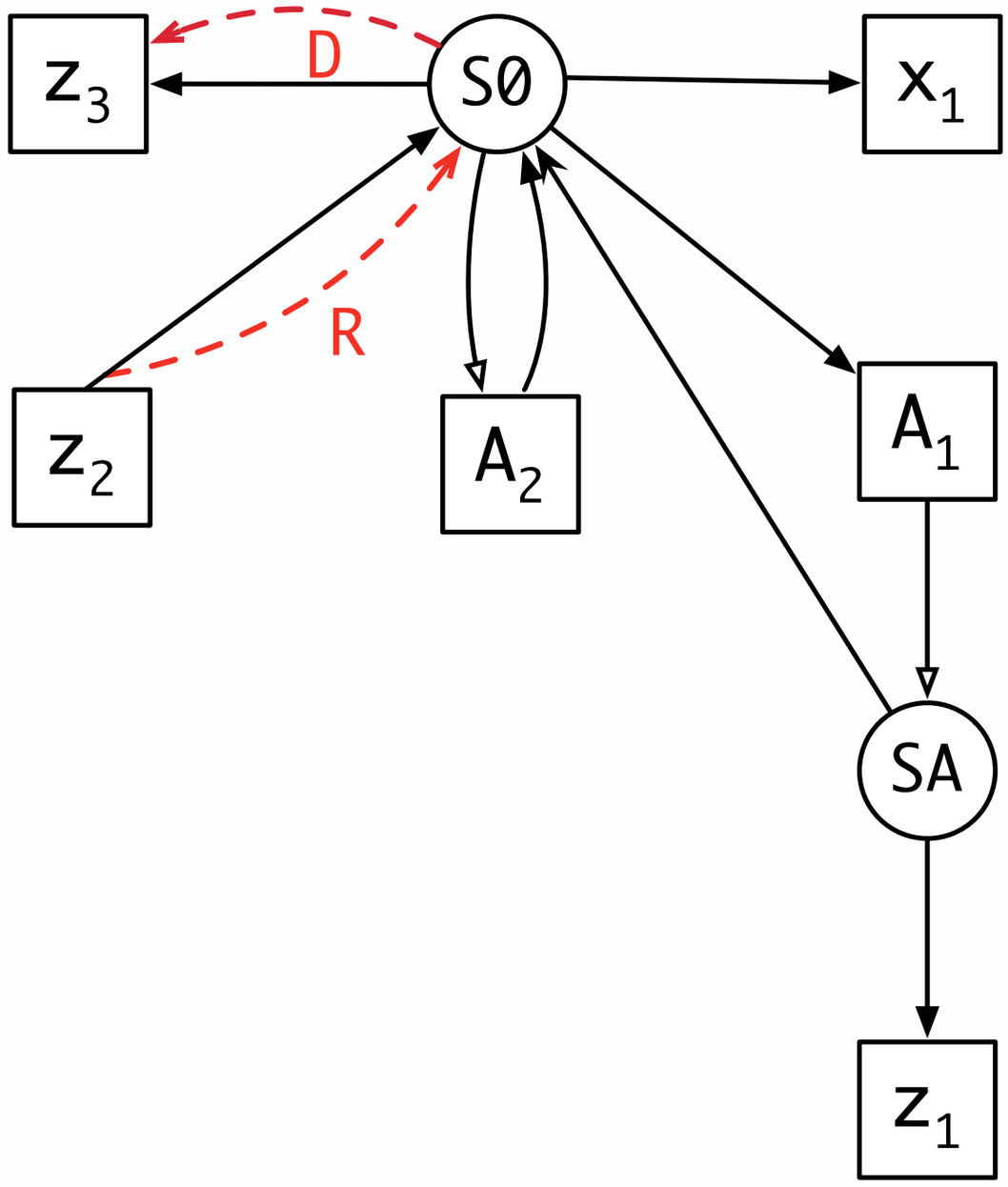
Imports vs. Includes

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

import A2
def x1 = 1 + z2
```



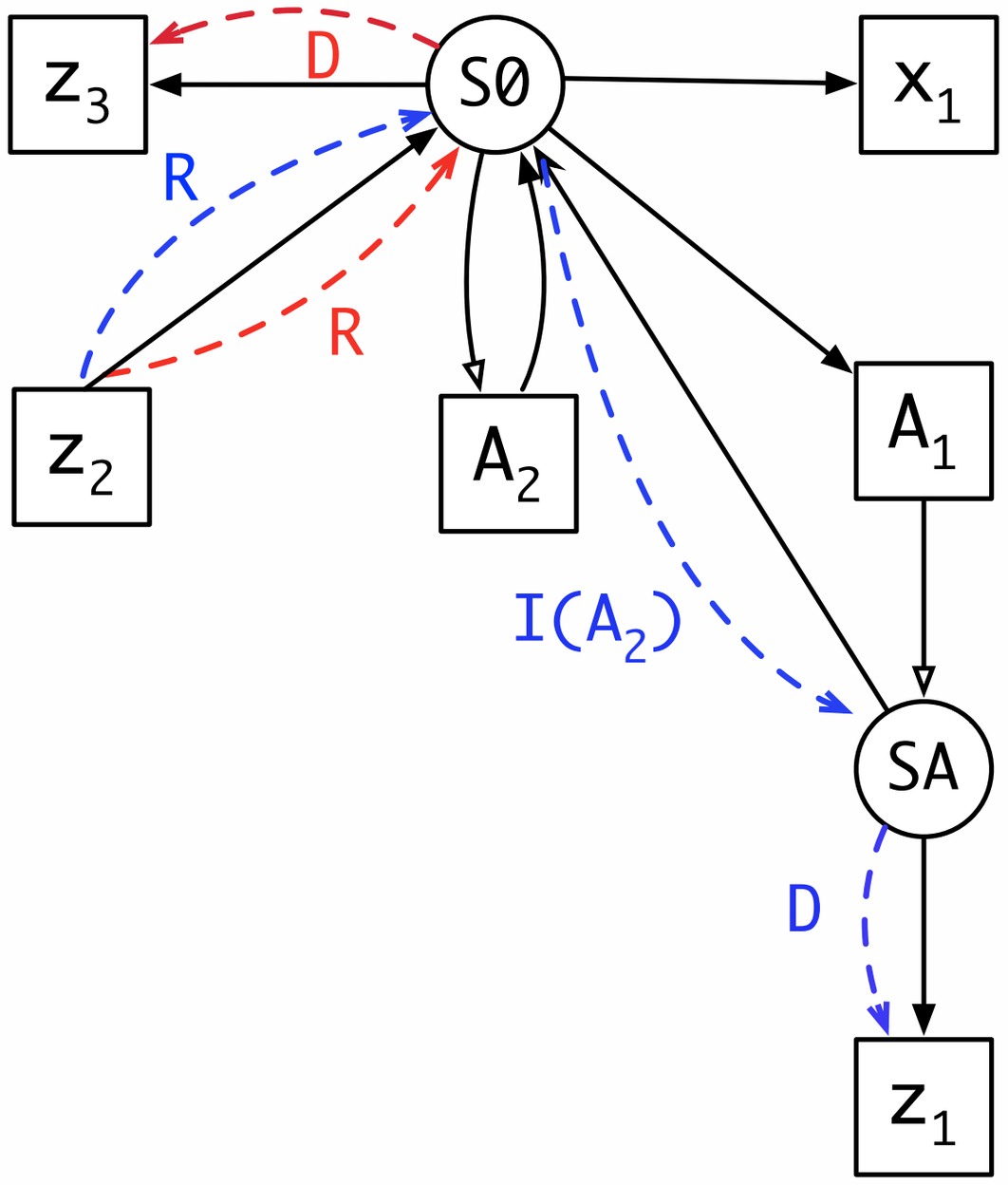
Imports vs. Includes

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

import A2
def x1 = 1 + z2
```



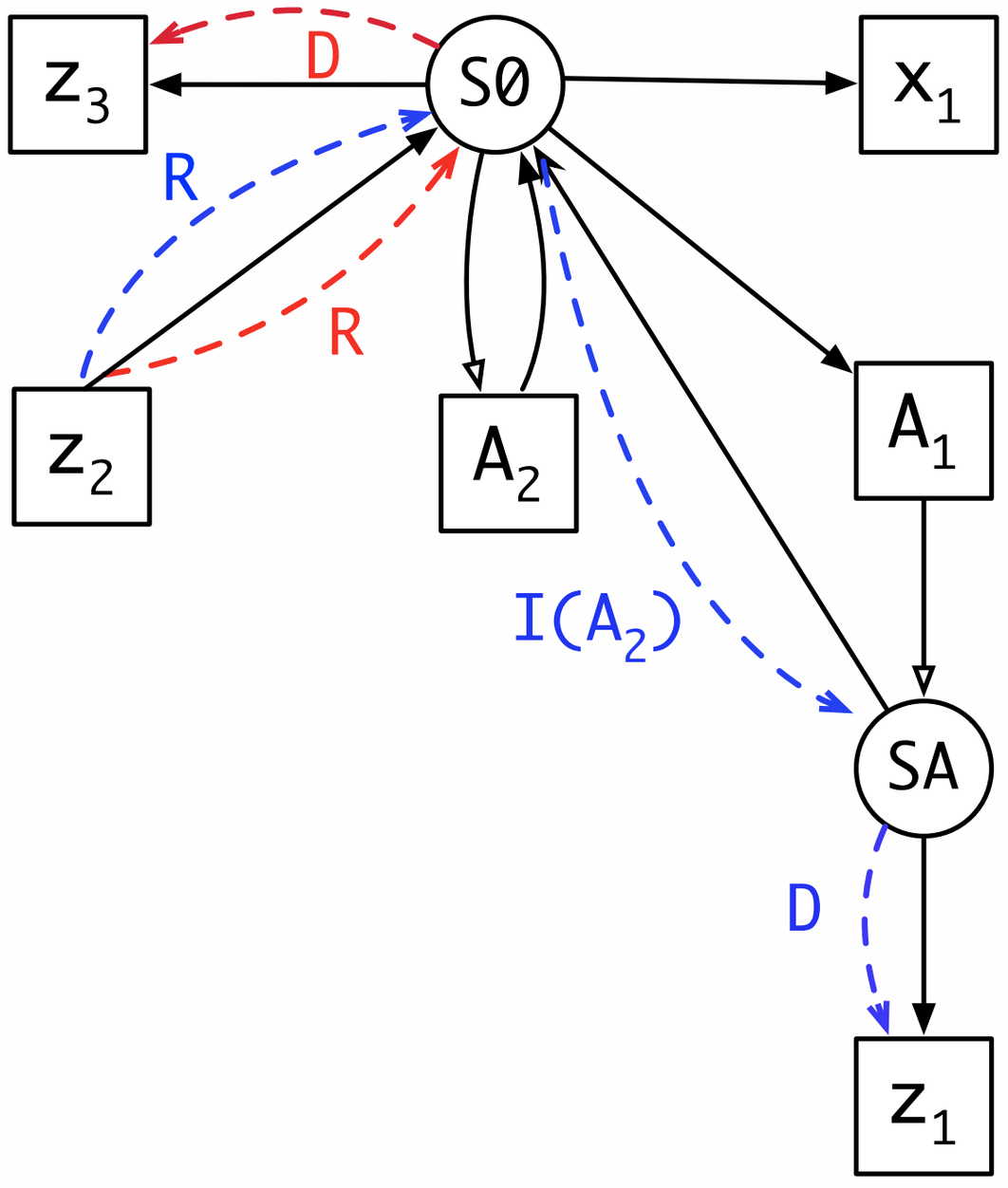
Imports vs. Includes

```
def z3 = 2                          S0

module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



Imports vs. Includes

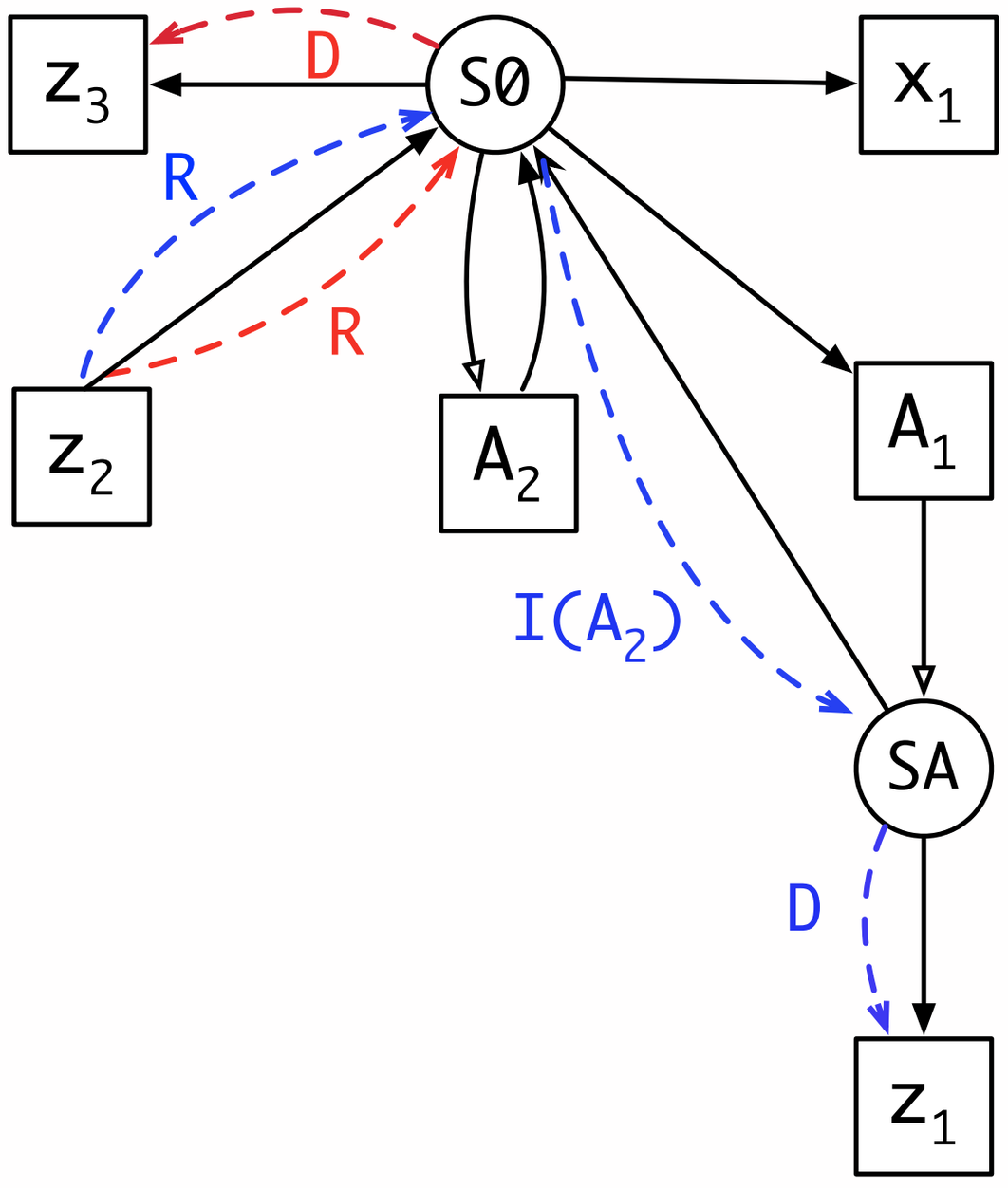
```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}

import A2
def x1 = 1 + z2
```



D < I(_).p’

Imports vs. Includes

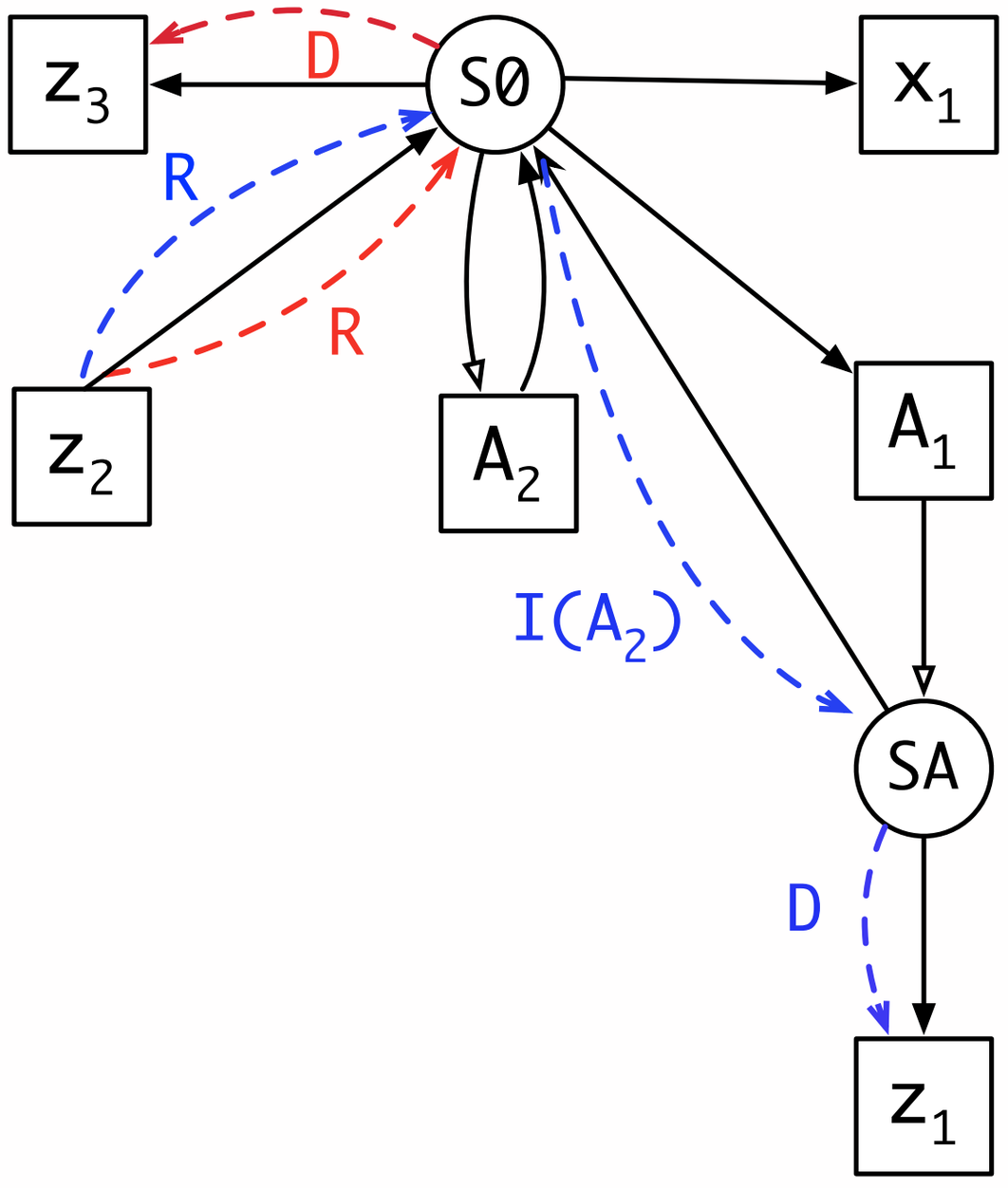
```
def z3 = 2                          S0
module A1 {
  def z1 = 5                        SA
}

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

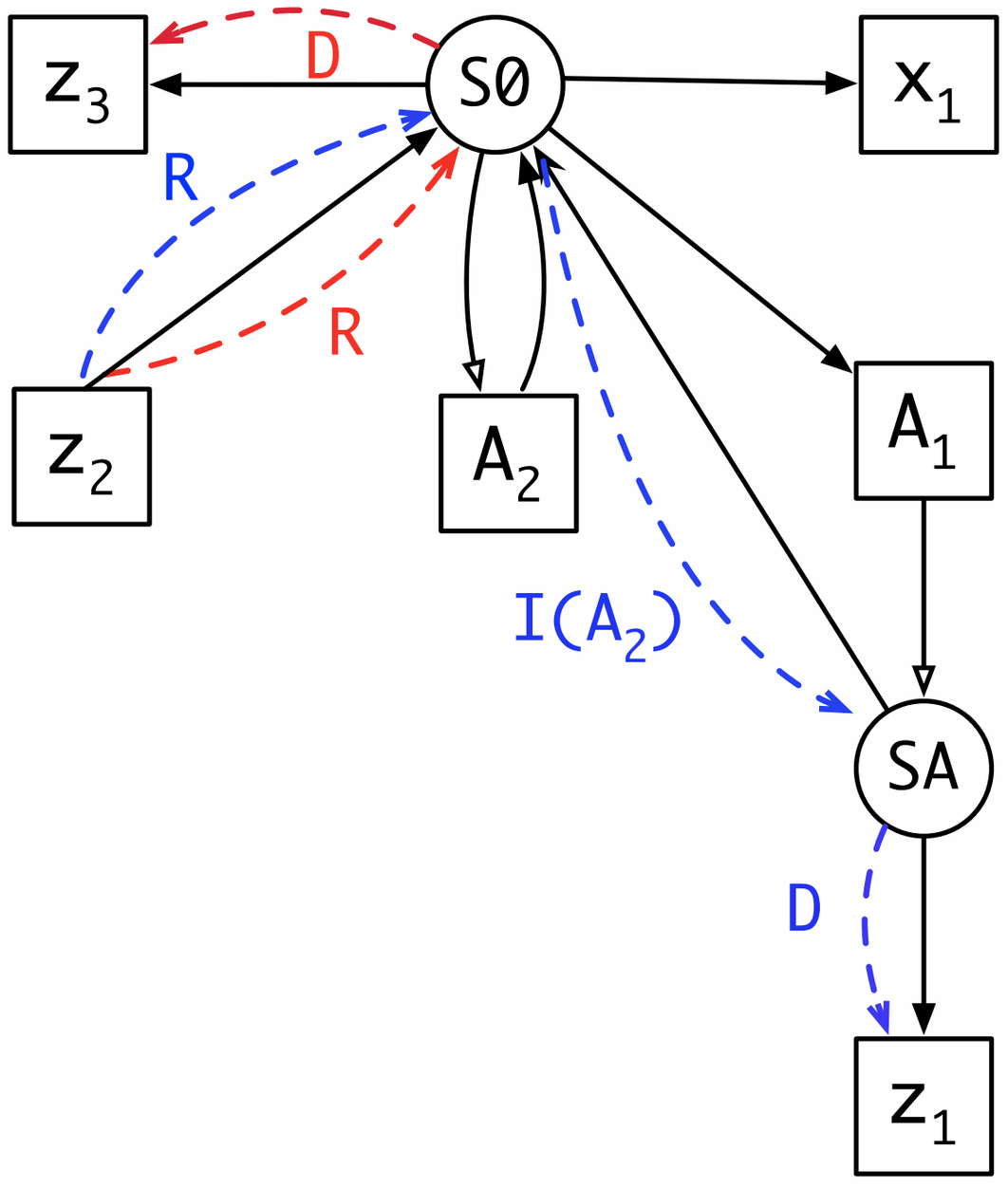
```
                                  S0
def z3 = 2
module A1 {
  def z1 = 5                      SA
}
include A2
def x1 = 1 + z2
```

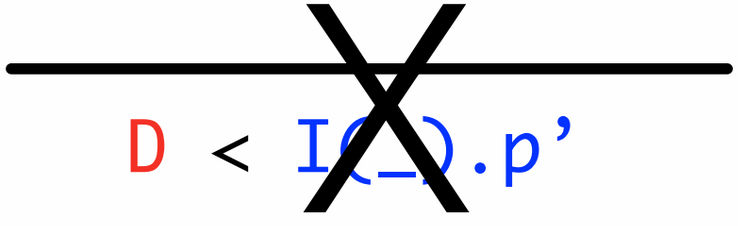


$$\frac{}{D < I(\_).p'} \quad \Rightarrow \quad R.D < R.I(A_2).D$$

Imports vs. Includes

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

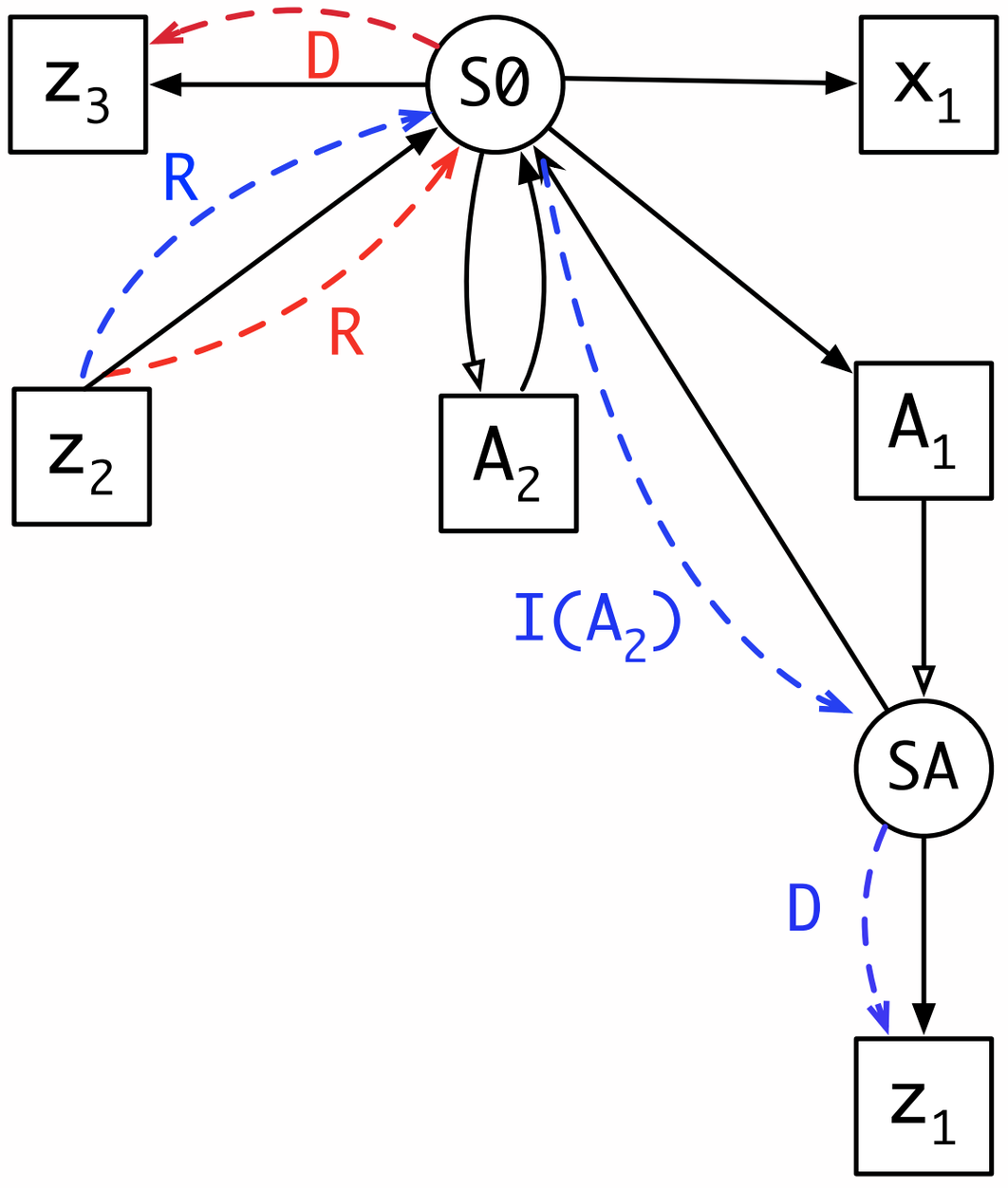


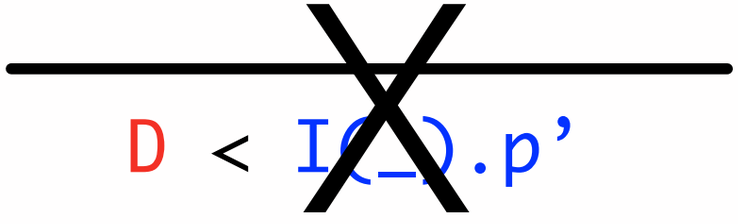


$\Rightarrow$ $R.D < R.I(A_2).D$

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
}
```

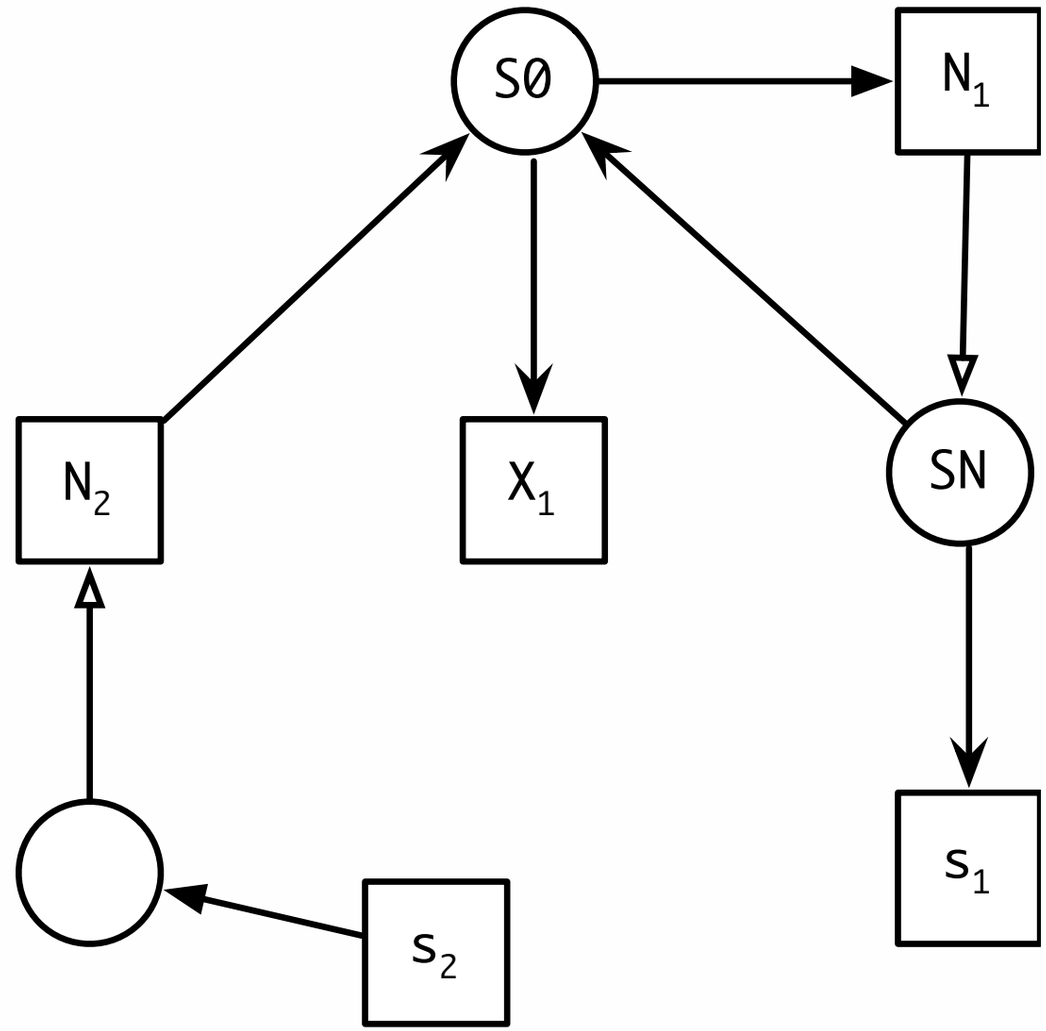
Qualified Names

S0

```
module N₁ {
    def s₁ = 5
}

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

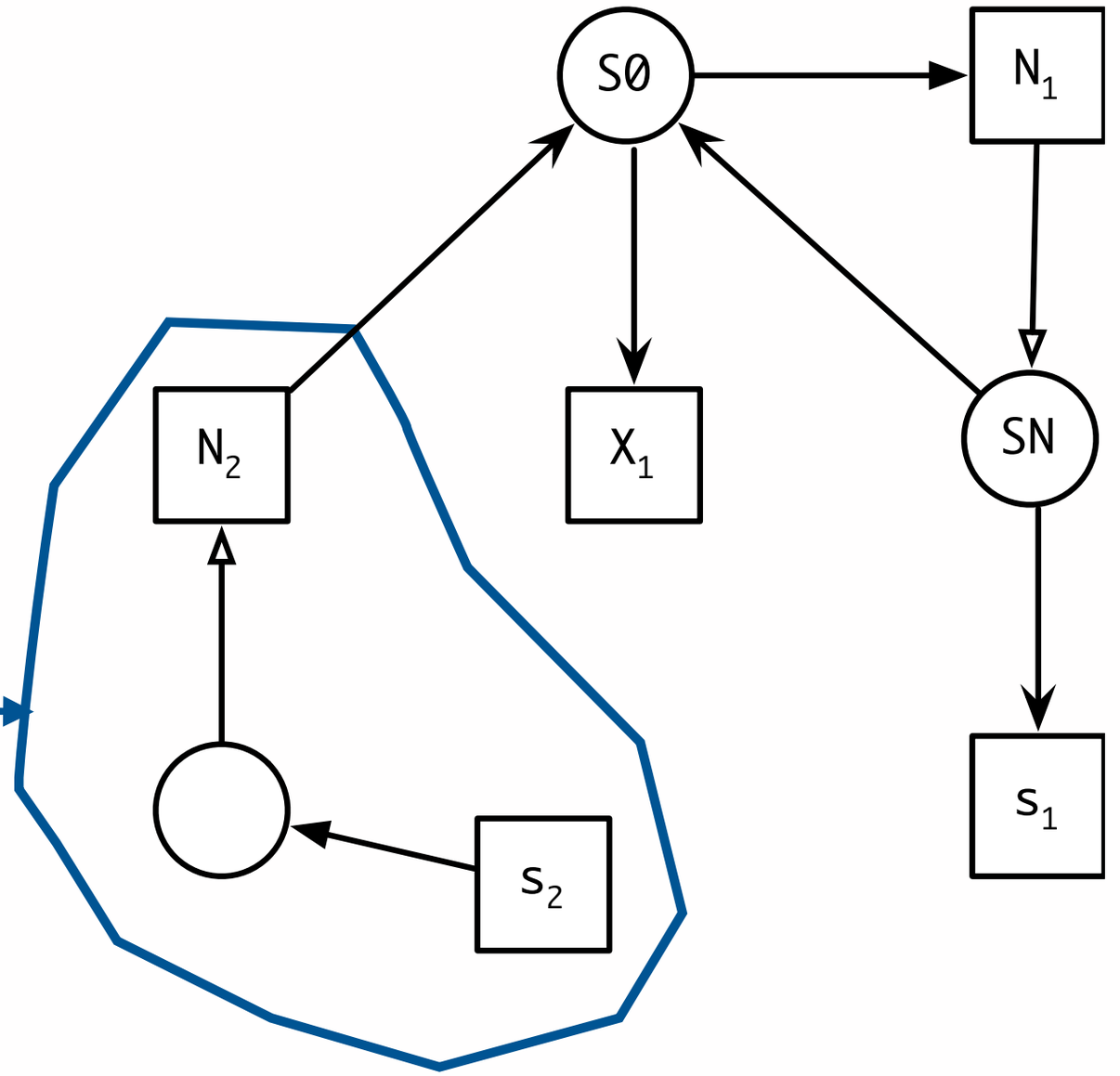
Qualified Names

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

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



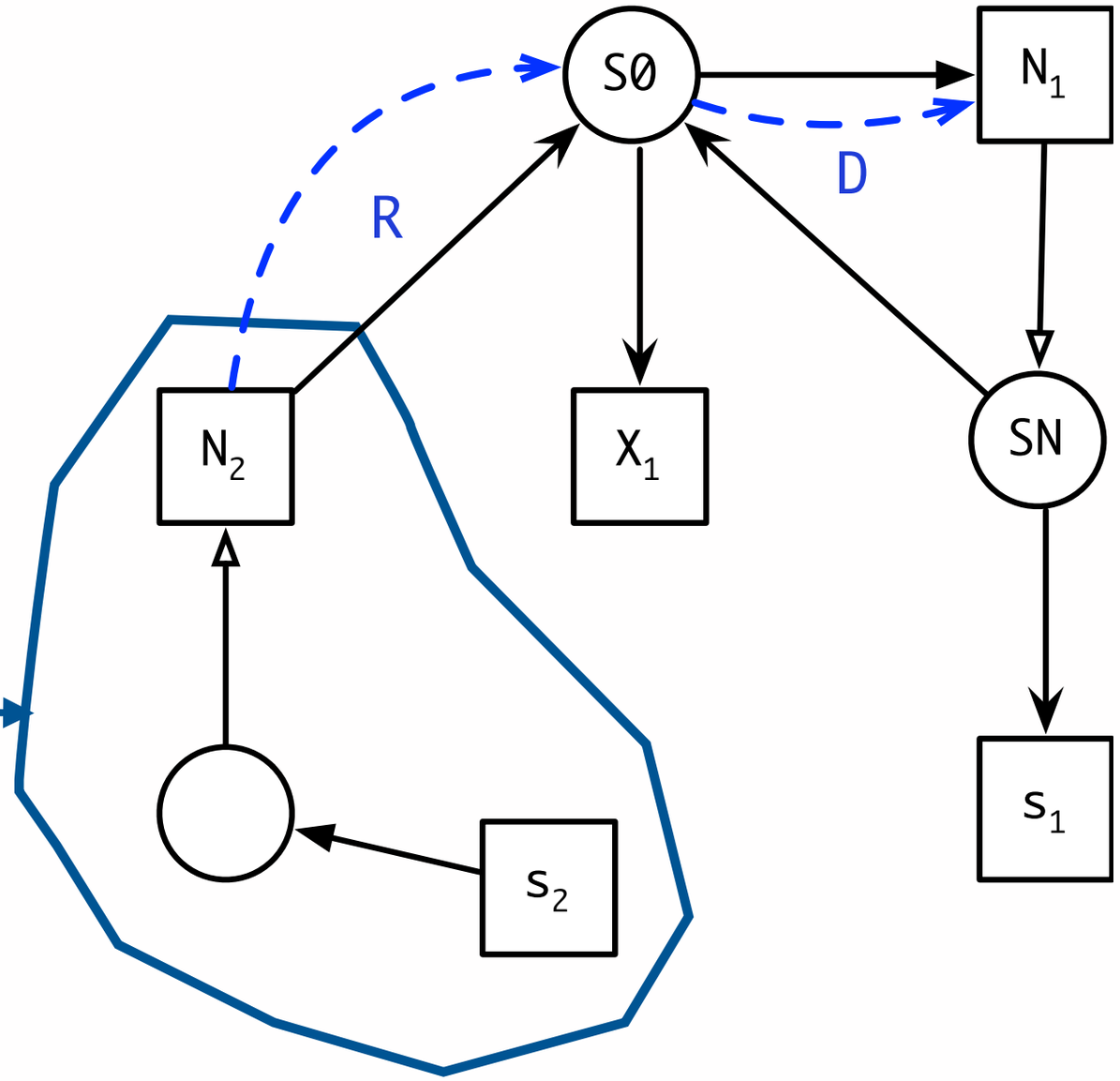
Qualified Names

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

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



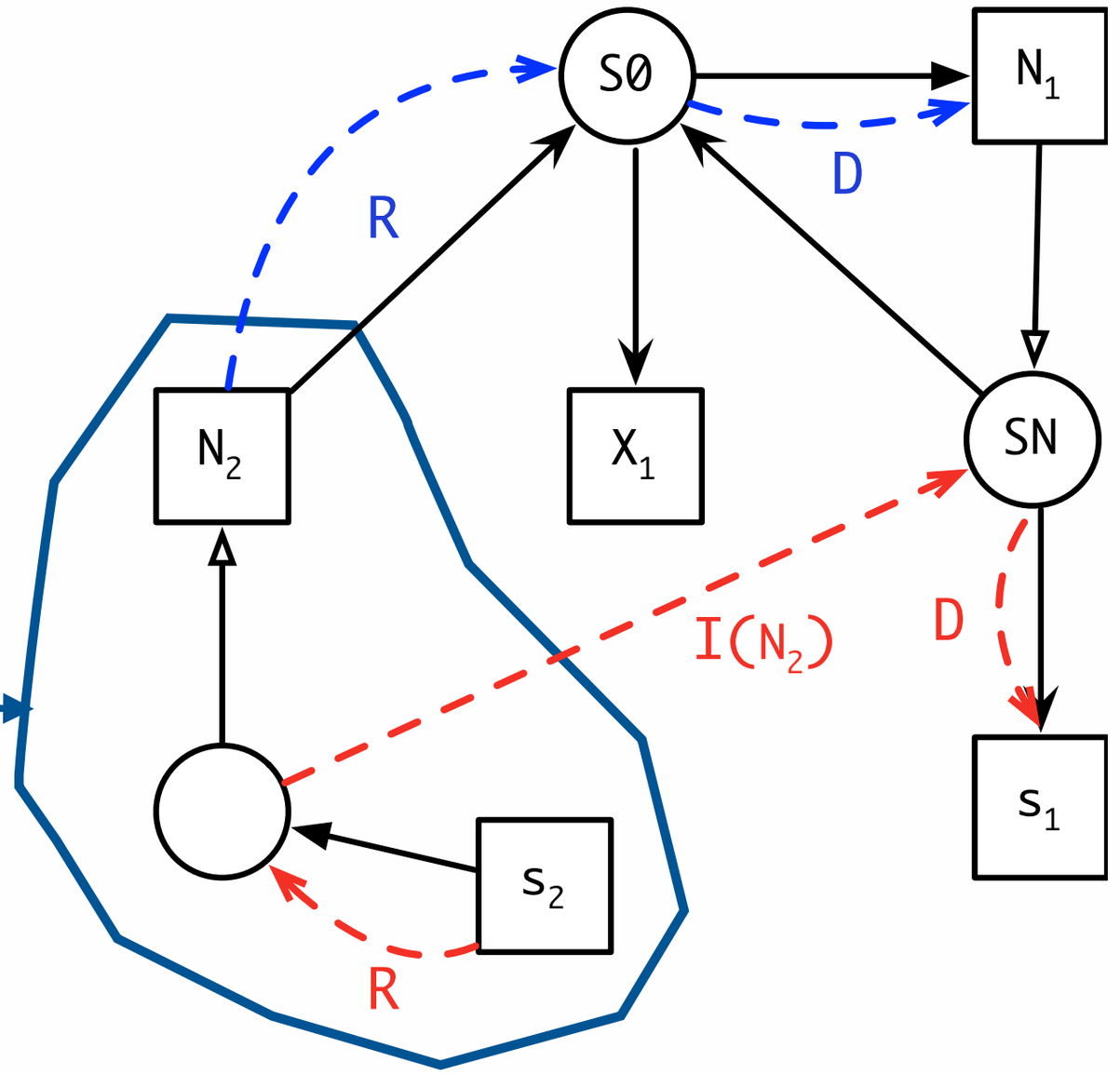
Qualified Names

```
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
}
```

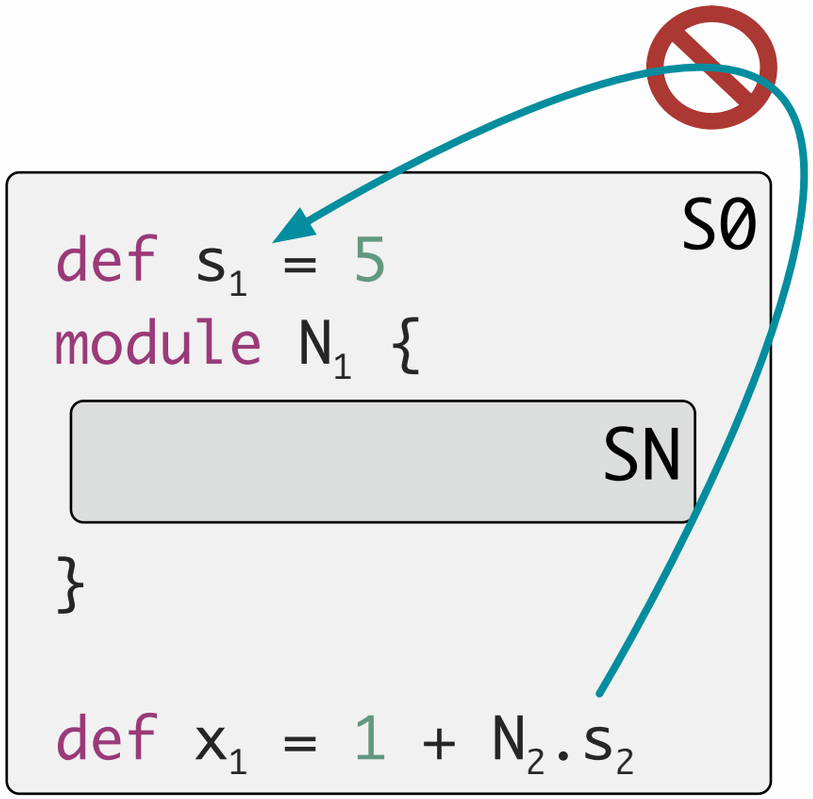


Well-Formedness

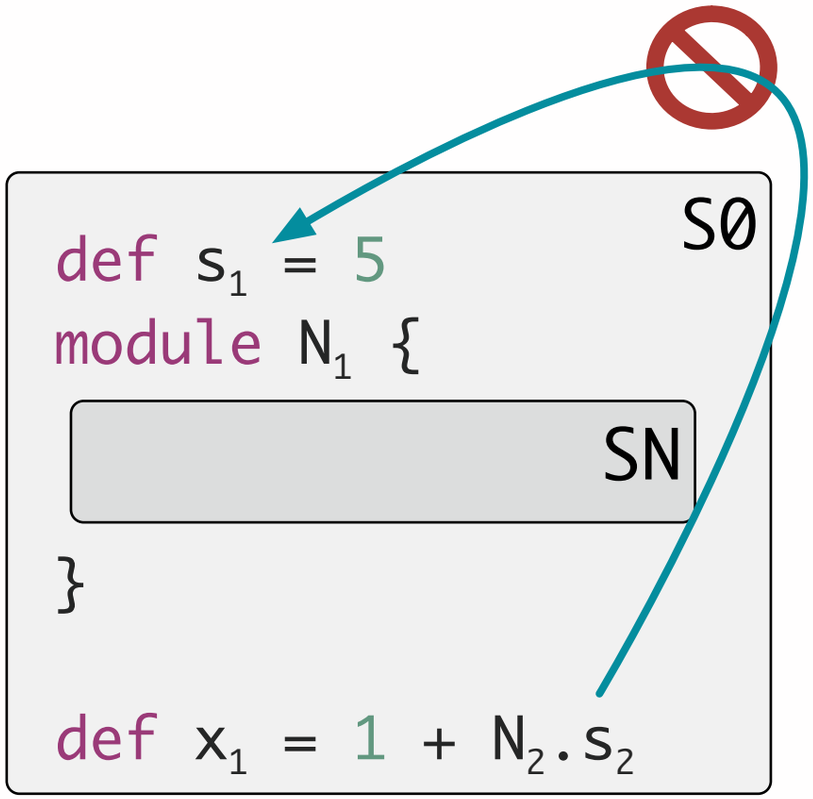
S0

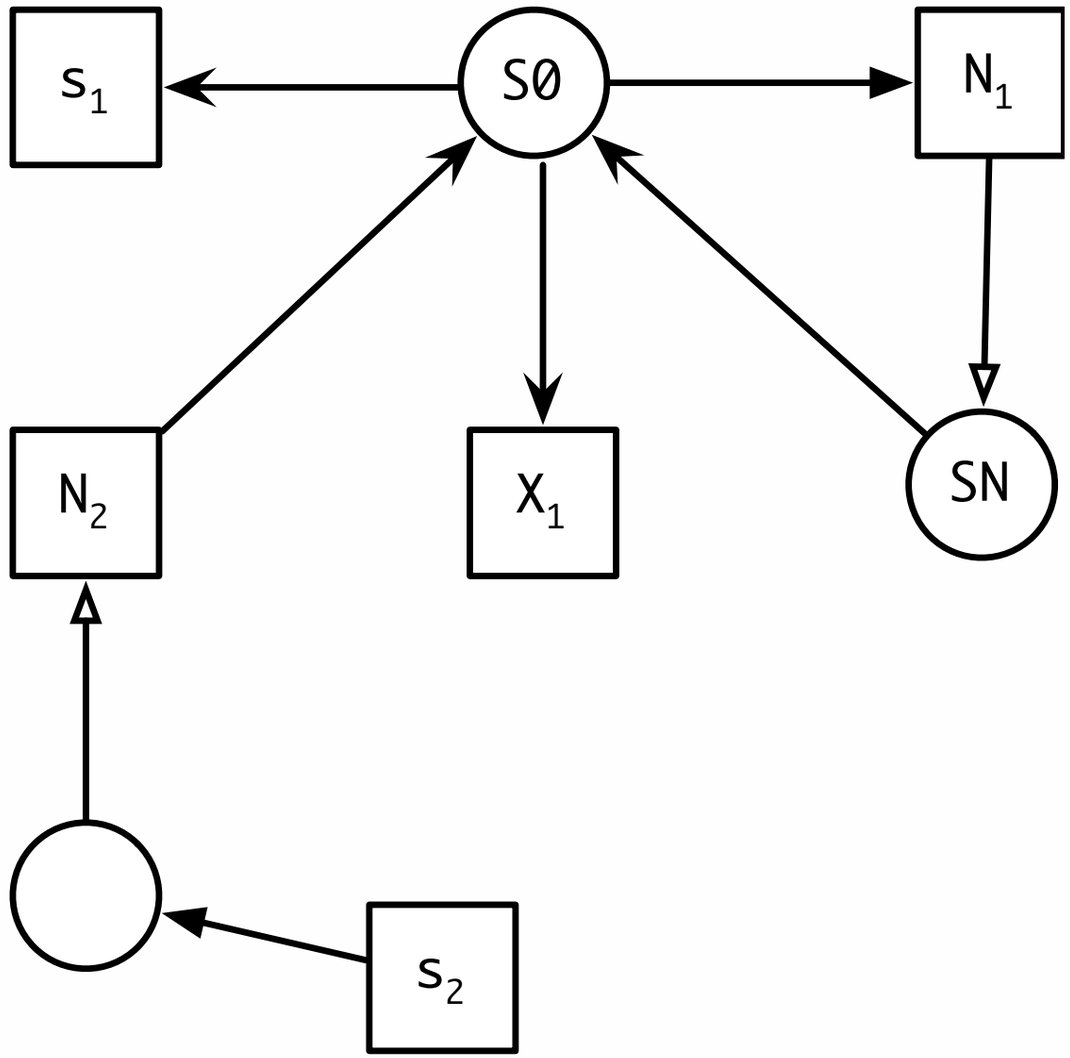
```
def s₁ = 5
module N₁ {
  SN
}

def x₁ = 1 + N₂.s₂
```

Well-Formedness

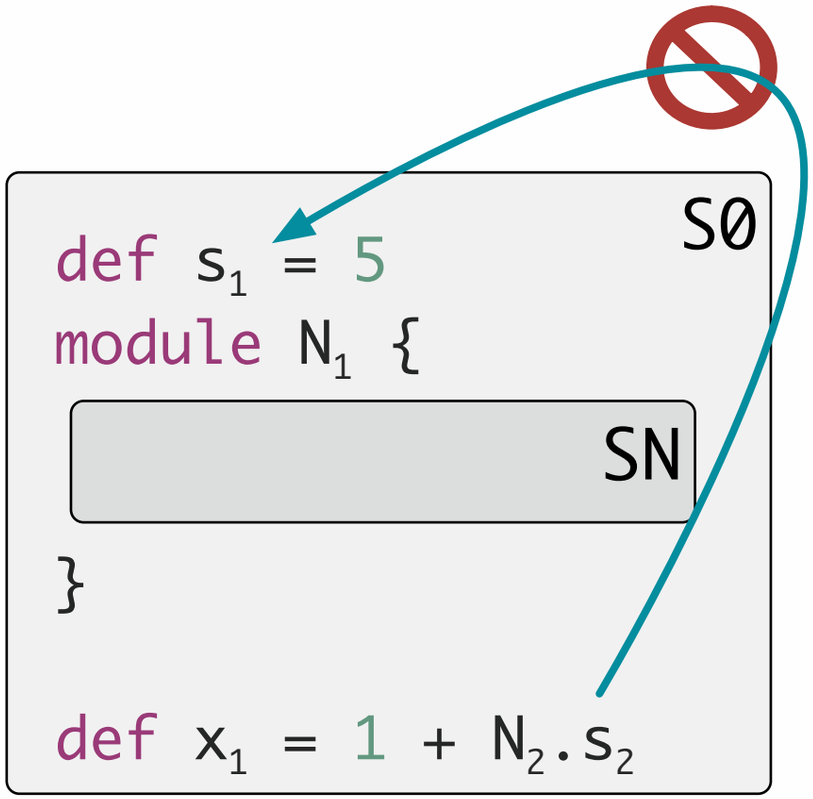


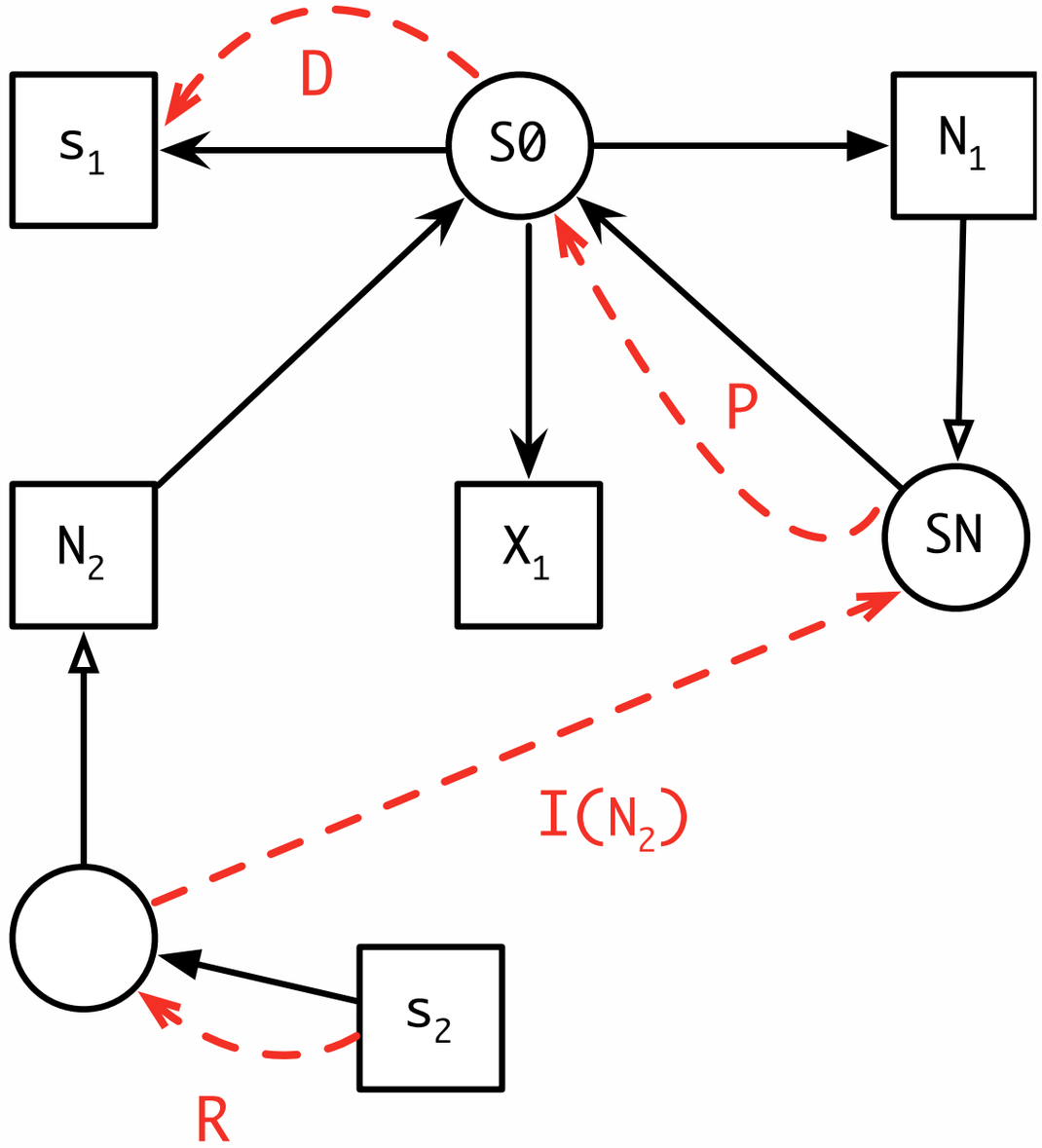
Well-Formedness



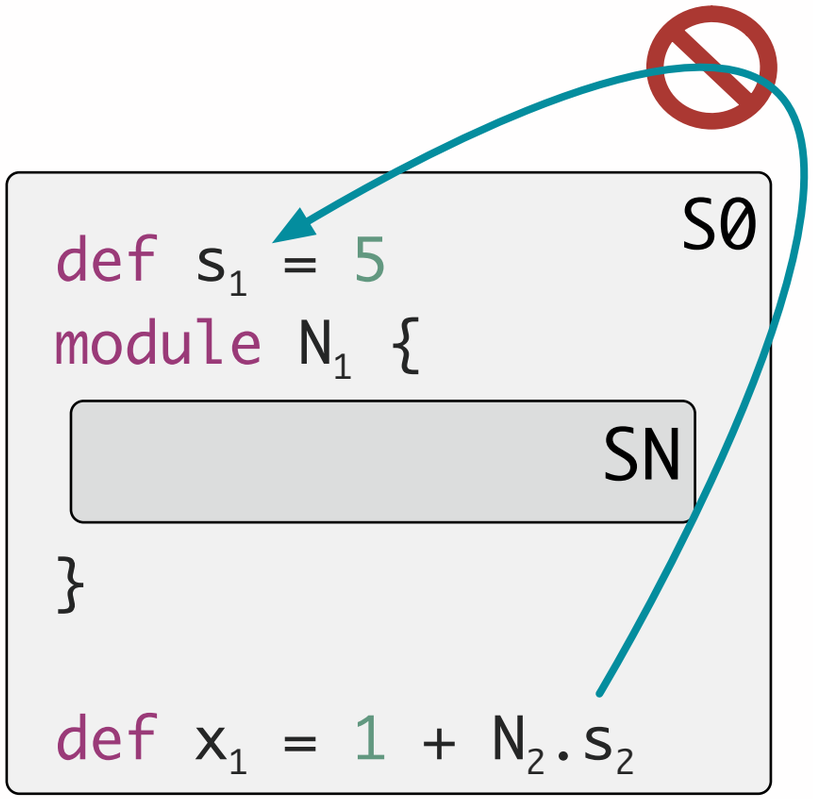


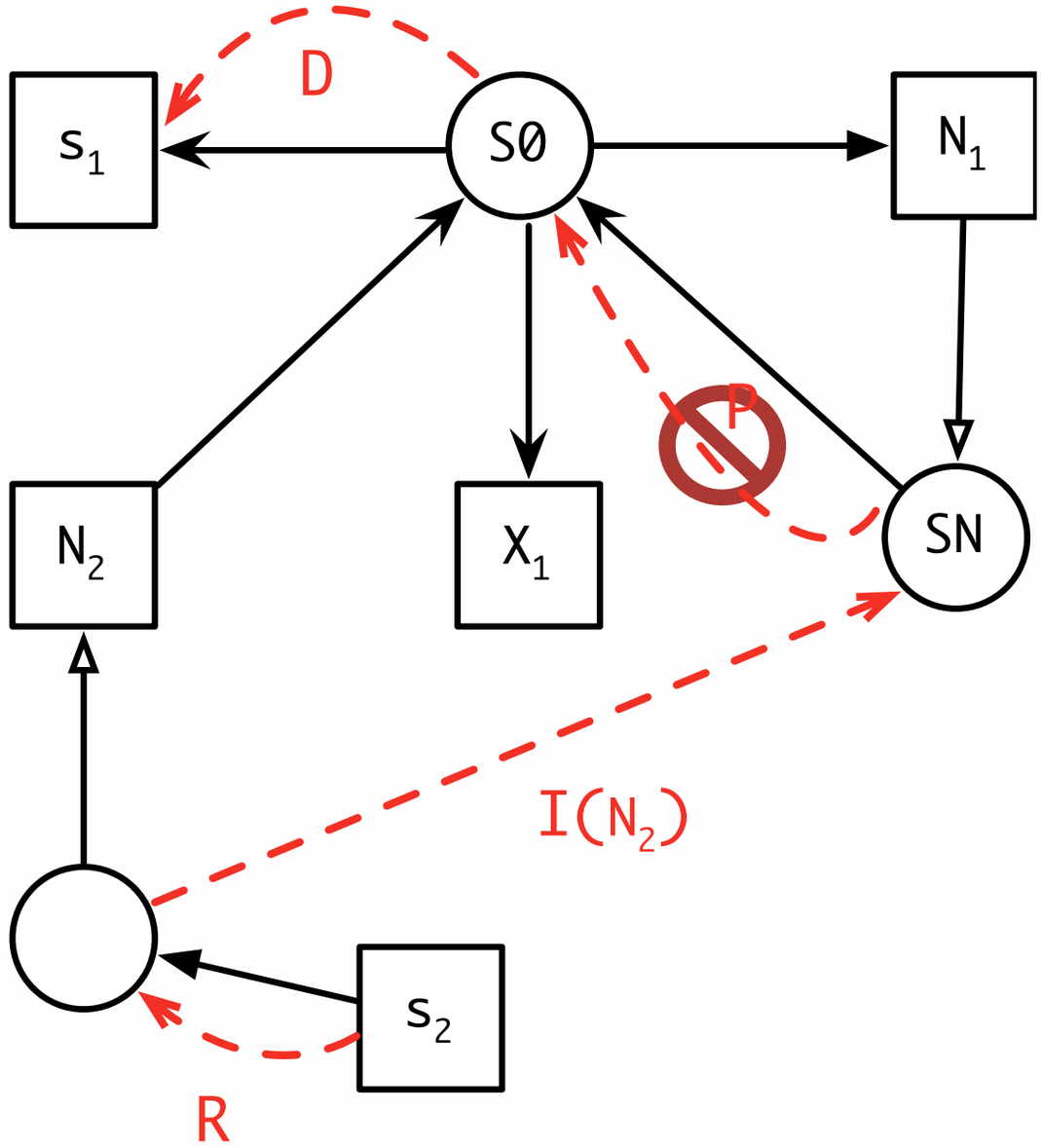
Well-Formedness



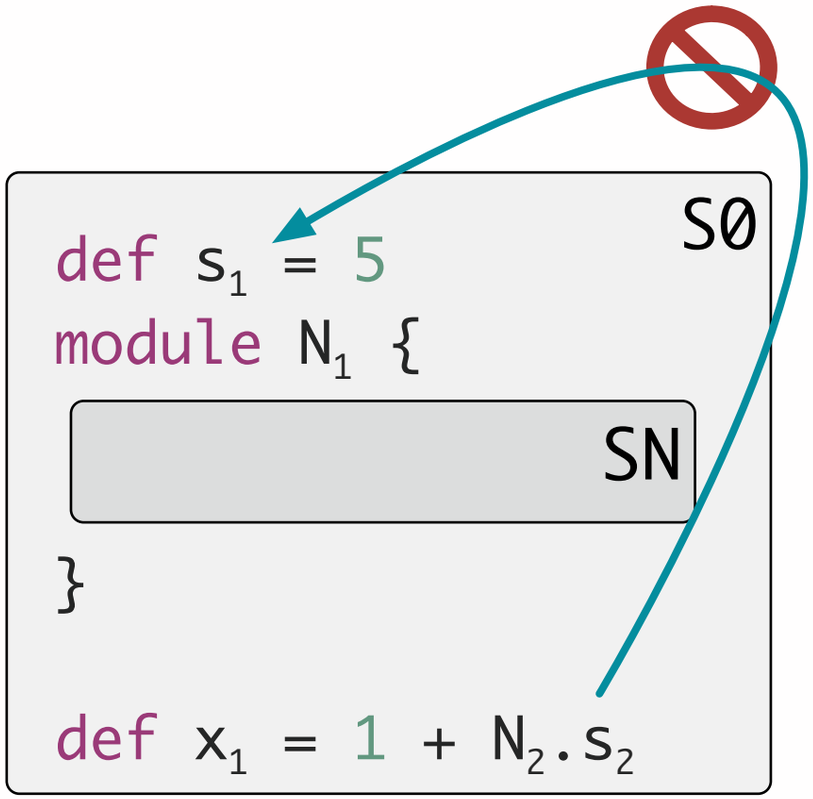


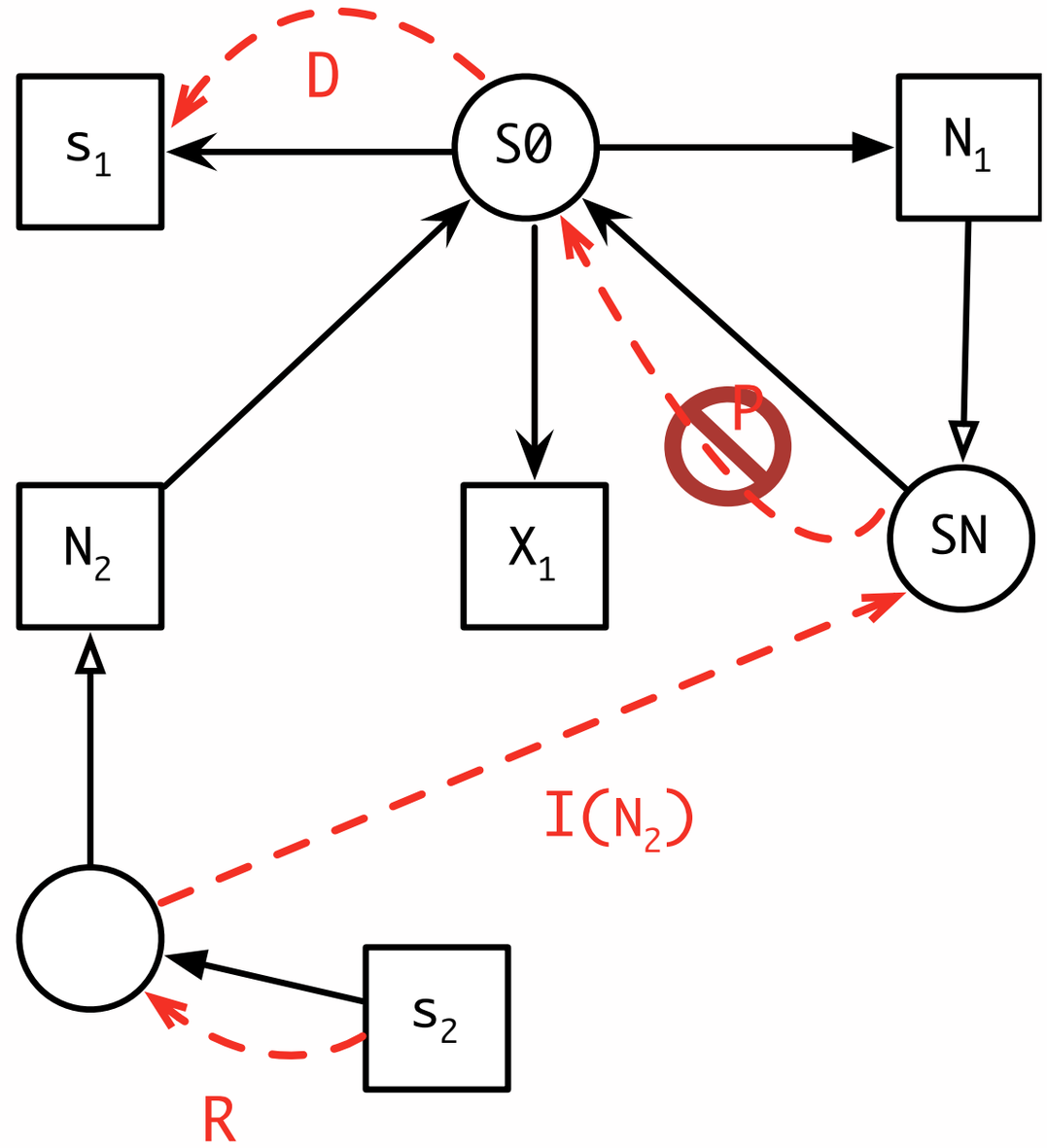
Well-Formedness





Well-Formedness



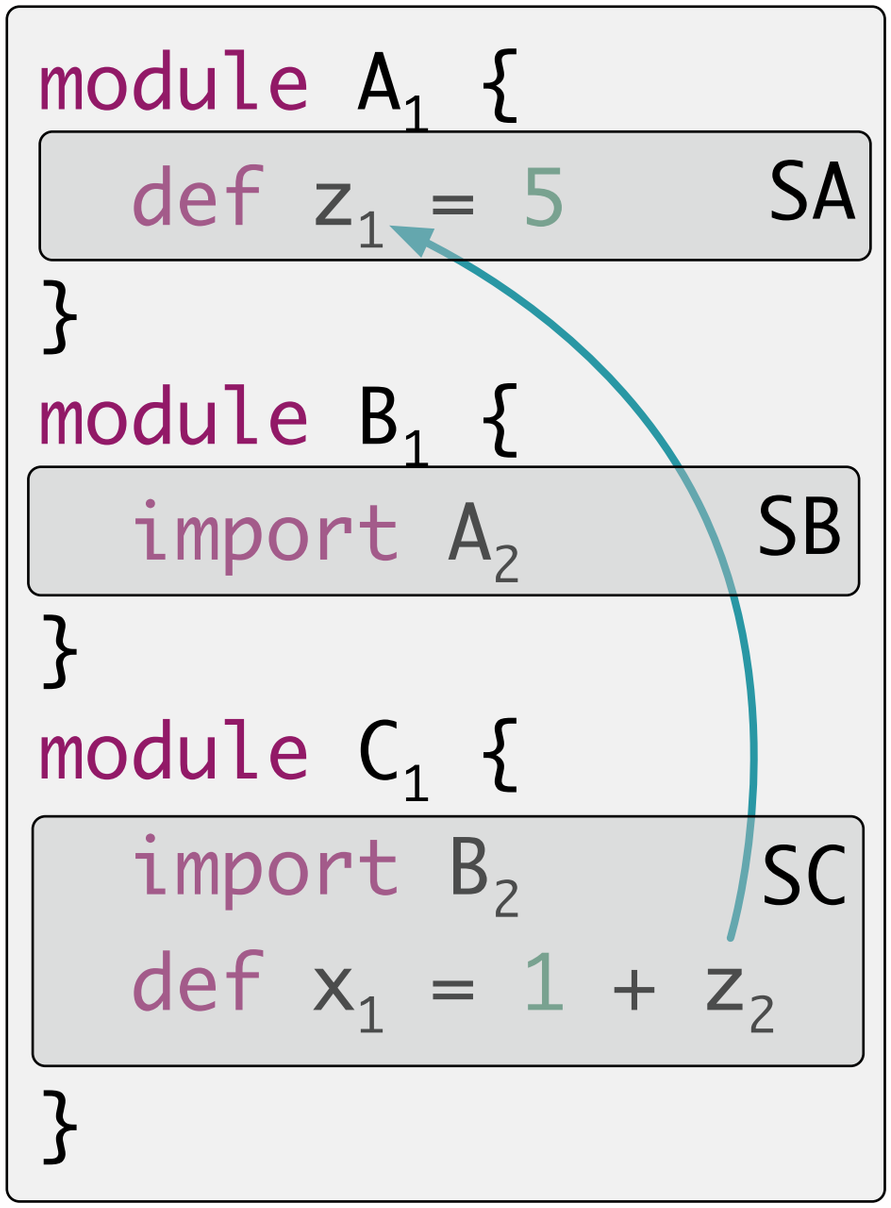


Well formed path: R.P*.I(_)*.D

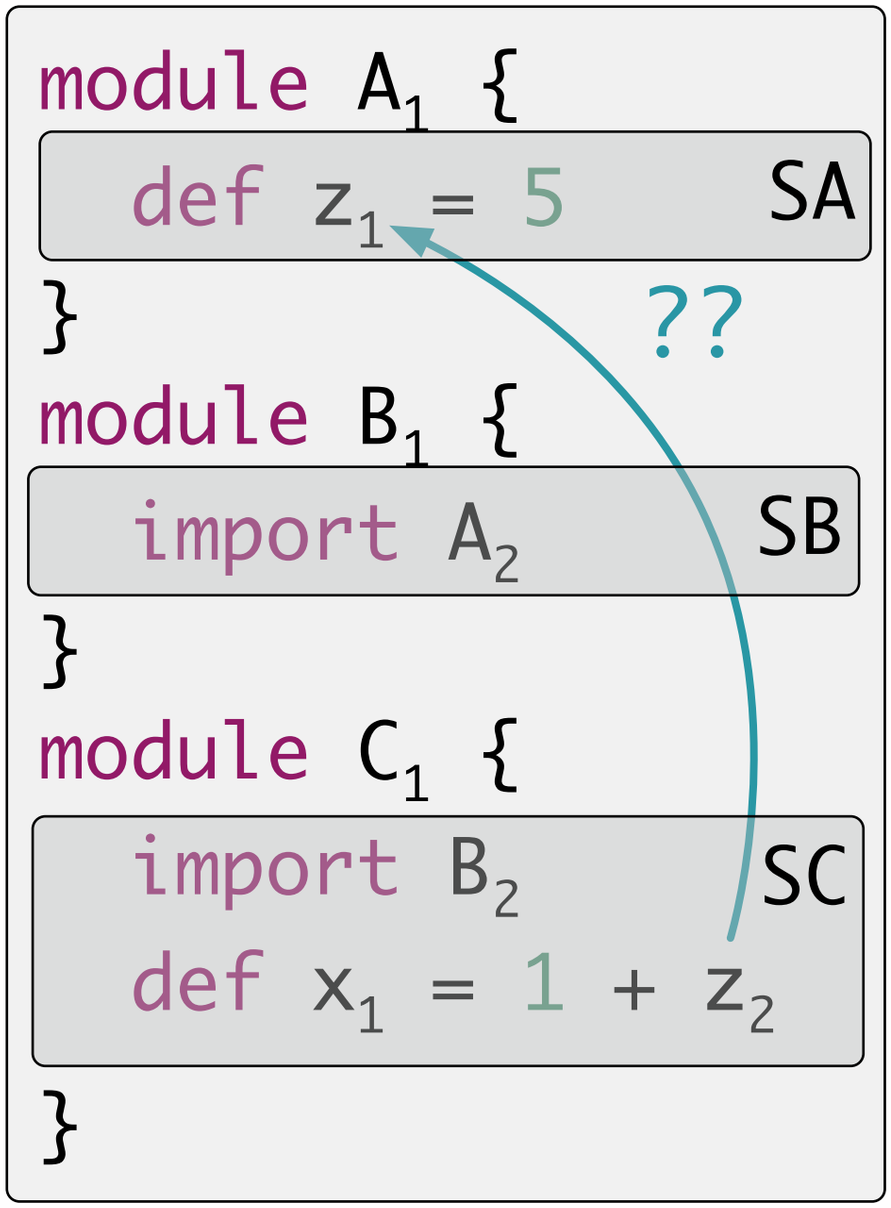
Transitive vs. Non-Transitive

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

Transitive vs. Non-Transitive

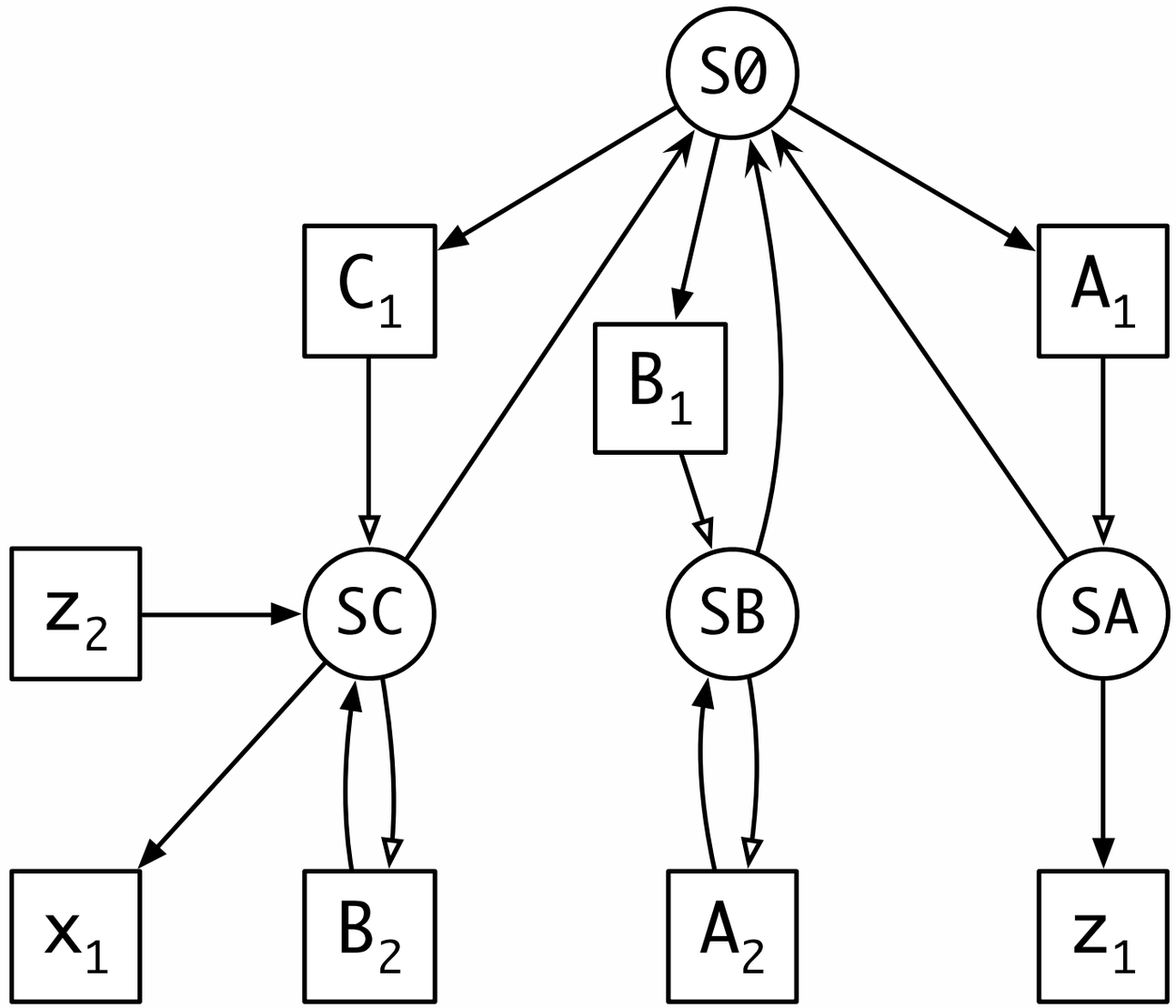


Transitive vs. Non-Transitive



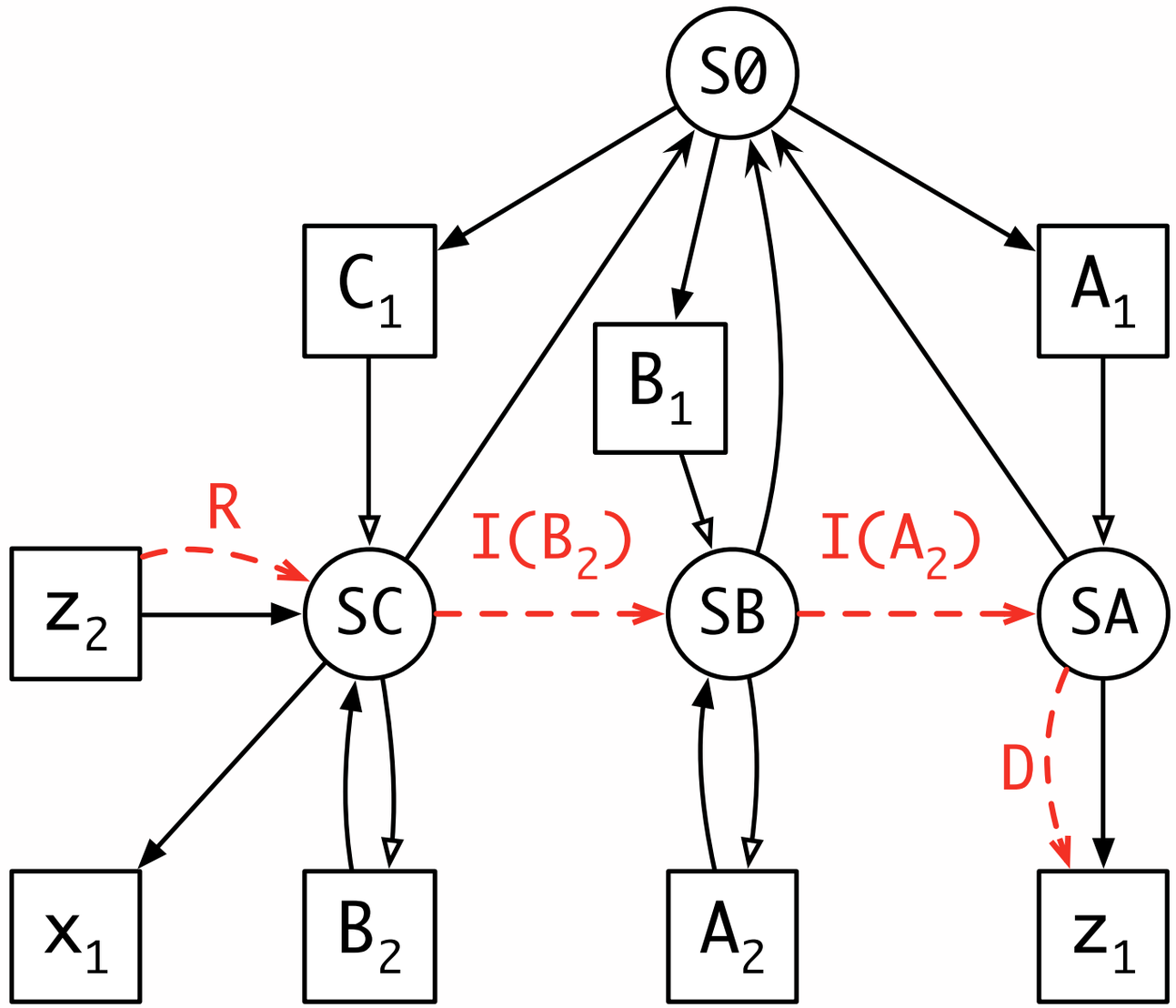
Transitive vs. Non-Transitive

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



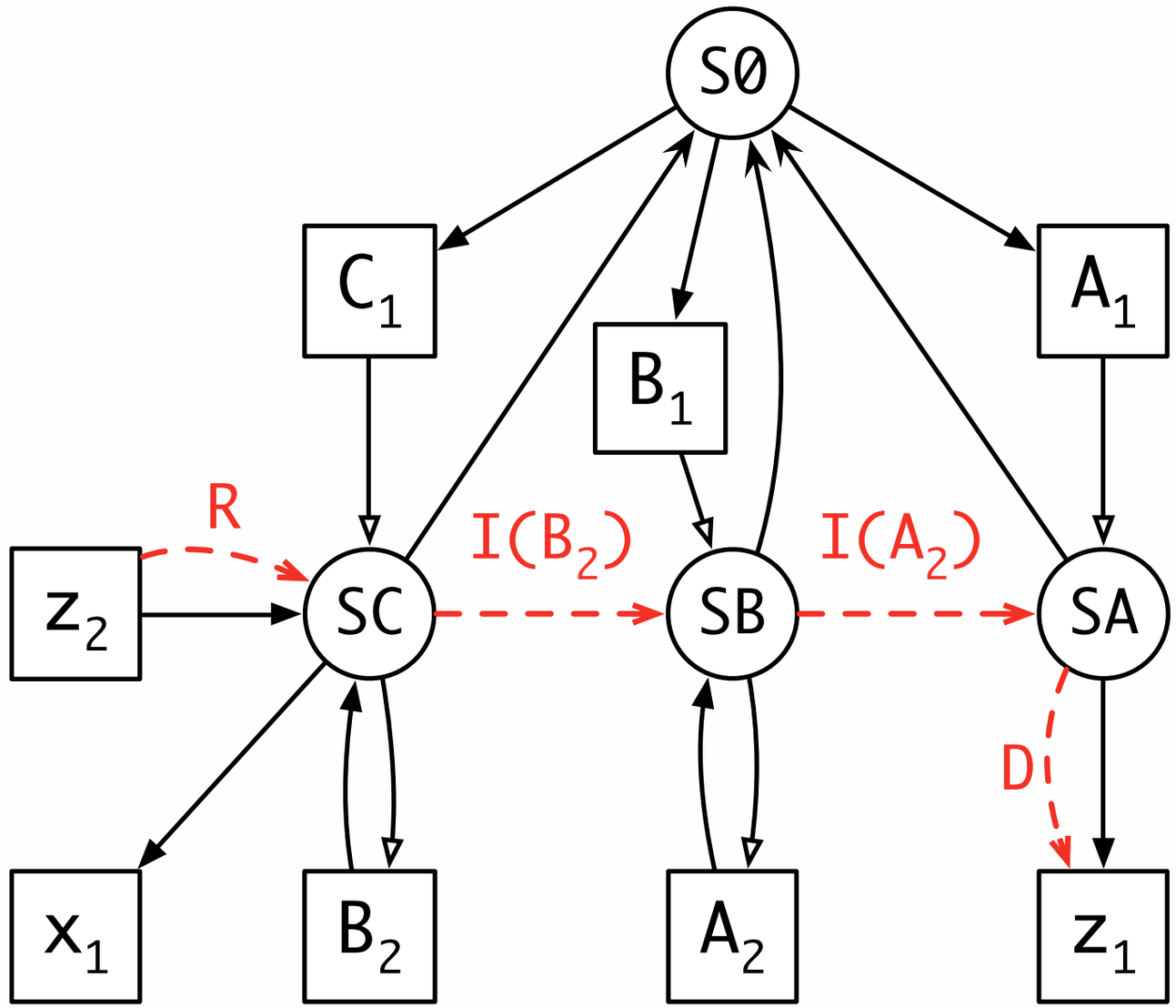
Transitive vs. Non-Transitive

```
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
}
```

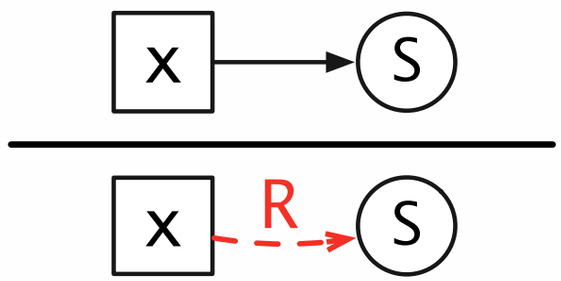


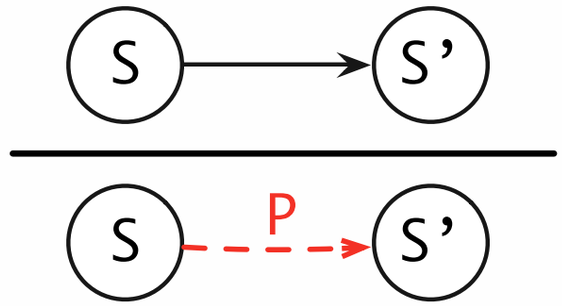
With transitive imports, a well formed path is `R.P*.I(_)*.D`

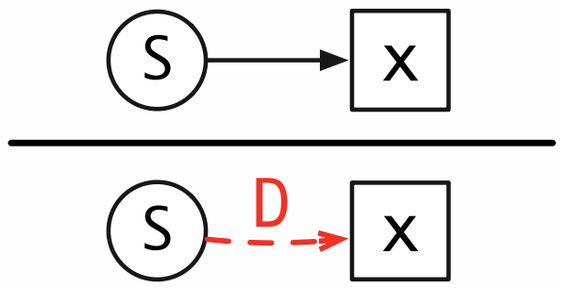
With non-transitive imports, a well formed path is `R.P*.I(_)?.D`

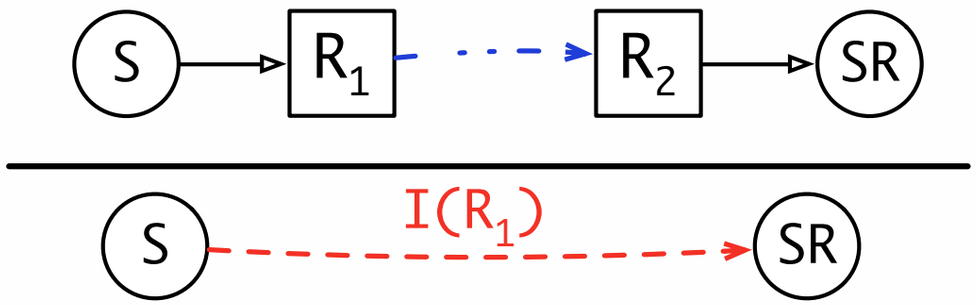
A Calculus for Name Resolution

Reachability









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

Visibility

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

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

What we have

- Formalization of the resolution calculus

Edges in scope graph

$$\frac{\mathcal{P}(S_1) = S_2}{\mathbb{I} \vdash \mathbf{P} : S_1 \longrightarrow S_2} \quad (P)$$

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

Transitive closure

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

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

Reachable declarations

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

Visible declarations

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

Reference resolution

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

Well-formed paths

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_, \_)^*$$

Specificity ordering on paths

$$\frac{}{\mathbf{D}(\_) < \mathbf{I}(\_, \_)} \quad (DI) \qquad \frac{}{\mathbf{I}(\_, \_) < \mathbf{P}} \quad (IP) \qquad \frac{}{\mathbf{D}(\_) < \mathbf{P}} \quad (DP)$$

$$\frac{s_1 < s_2}{s_1 \cdot p_1 < s_2 \cdot p_2} \; (Lex1) \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2} \; (Lex2)$$

What we have

- Formalization of the resolution calculus
- Validation by an extensive set of examples

What we have

- Formalization of the resolution calculus
- Validation by an extensive set of examples
    - definition before use

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

    - definition before use
    - different let binding flavors

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

    - definition before use
    - different let binding flavors
    - recursive modules

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

    - definition before use
    - different let binding flavors
    - recursive modules
    - imports and includes

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

    - definition before use
    - different let binding flavors
    - recursive modules
    - imports and includes
    - qualified names

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

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

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

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

What we have

- Formalization of the resolution calculus

- Validation by an extensive set of examples

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

What we have

- Formalization of the resolution calculus
- Validation by an extensive set of examples
- Sound and complete resolution algorithm
  (for specific choice of WF and ordering)

$$
\begin{array}{lll}
Res[\mathbb{I}](x_i^{\mathsf{R}}) & := \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in Env_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](S)\} \\
Env_V[\mathbb{I}, \mathbb{S}](S) & := Env_L[\mathbb{I}, \mathbb{S}](S) \triangleleft Env_P[\mathbb{I}, \mathbb{S}](S) \\
Env_L[\mathbb{I}, \mathbb{S}](S) & := Env_D[\mathbb{I}, \mathbb{S}](S) \triangleleft Env_I[\mathbb{I}, \mathbb{S}](S) \\
Env_D[\mathbb{I}, \mathbb{S}](S) & := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \mathcal{D}(S) \end{cases} \\
Env_I[\mathbb{I}, \mathbb{S}](S) & := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \bigcup \{ Env_L[\mathbb{I}, \{S\} \cup \mathbb{S}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}}{:}S_y \in Res[\mathbb{I}](y_i^{\mathsf{R}}) \} \end{cases} \\
Env_P[\mathbb{I}, \mathbb{S}](S) & := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ Env_V[\mathbb{I}, \{S\} \cup \mathbb{S}](\mathcal{P}(S)) \end{cases}
\end{array}
$$

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

What we have

- Formalization of the resolution calculus
- Validation by an extensive set of examples
- Sound and complete resolution algorithm
- Language-independent definition of $\boldsymbol{\alpha}$-equivalence

$$\frac{\vdash p : x_i^{\mathsf{R}} \longmapsto x_{i'}^{\mathsf{D}}}{i \overset{\mathtt{P}}{\sim} i'} \qquad \frac{i' \overset{\mathtt{P}}{\sim} i}{i \overset{\mathtt{P}}{\sim} i'} \qquad \frac{i \overset{\mathtt{P}}{\sim} i' \quad i' \overset{\mathtt{P}}{\sim} i''}{i \overset{\mathtt{P}}{\sim} i''} \qquad \frac{}{i \overset{\mathtt{P}}{\sim} i}$$

$$\mathtt{P1} \overset{\alpha}{\approx} \mathtt{P2} \triangleq \mathtt{P1} \simeq \mathtt{P2} \wedge \forall\ i\ i',\ i \overset{\mathtt{P1}}{\sim} i' \Leftrightarrow i \overset{\mathtt{P2}}{\sim} i'$$

What we have

- Formalization of the resolution calculus
- Validation by an extensive set of examples
- Sound and complete resolution algorithm
- Language-independent definition of alpha-equivalence

α-equivalence

Program similarity

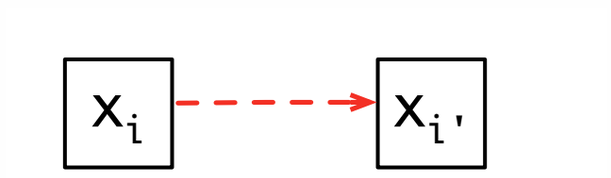
`P` $\simeq$ `P'` if have same AST ignoring identifier names

α-equivalence

Program similarity

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

Position equivalence



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

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

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

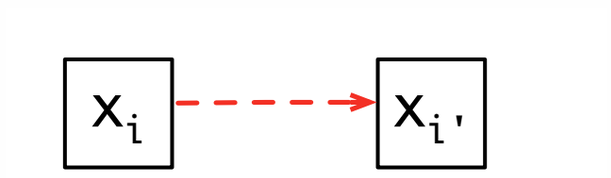
$$\frac{}{i \overset{\texttt{P}}{\sim} i}$$

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

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

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

What we’re working on

What we're working on

- Binding specification languages (NaBL)

What we're working on

- Binding specification languages (NaBL)
- Interaction of name resolution with type resolution

What we're working on

- Binding specification languages (NaBL)
- Interaction of name resolution with type resolution
- Resolution-sensitive program transformations

What we're working on

- Binding specification languages (NaBL)
- Interaction of name resolution with type resolution
- Resolution-sensitive program transformations
- Dynamic analogs to static scope graphs

What we're working on

- Binding specification languages (NaBL)
- Interaction of name resolution with type resolution
- Resolution-sensitive program transformations
- Dynamic analogs to static scope graphs
- Supporting mechanized language meta-theory

Types and Binding

In many cases, can treat binding before typing, e.g.

- associate types with declarations
- to get type of an identifier use, first resolve it and then consult associated type

Types and Binding

In many cases, can treat binding before typing, e.g.

- associate types with declarations
- to get type of an identifier use, first resolve it and then consult associated type

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

```
record A₁ { x₁ : Int }
record B₁ { a₁ : A₂ ; x₂ : Bool}
…
y₁ = z₁.x₃

y₂ = z₂.a₂.x₄
```

Types and Binding

In many cases, can treat binding before typing, e.g.

- associate types with declarations
- to get type of an identifier use, first resolve it and then consult associated type

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

```
record A₁ { x₁ : Int }
record B₁ { a₁ : A₂ ; x₂ : Bool}
…
y₁ = z₁.x₃

y₂ = z₂.a₂.x₄
```

Our approach: interleave **partial** name resolution with type resolution