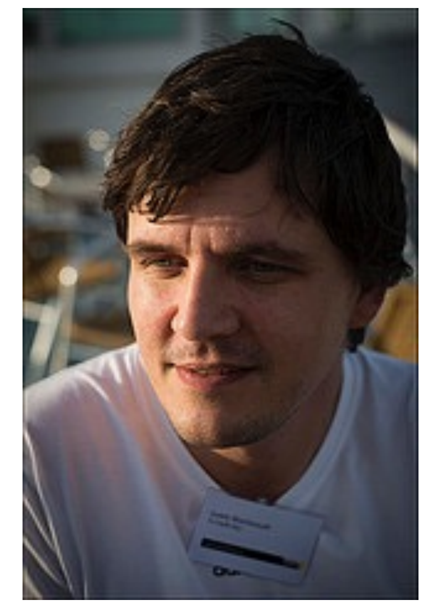
A Theory of Name Resolution

Pierre Neron[1]

Andrew Tolmach[2]

Eelco Visser[1]



Guido Wachsmuth[1]

[1] TU Delft — Delft University of Technology

[2] Portland State University

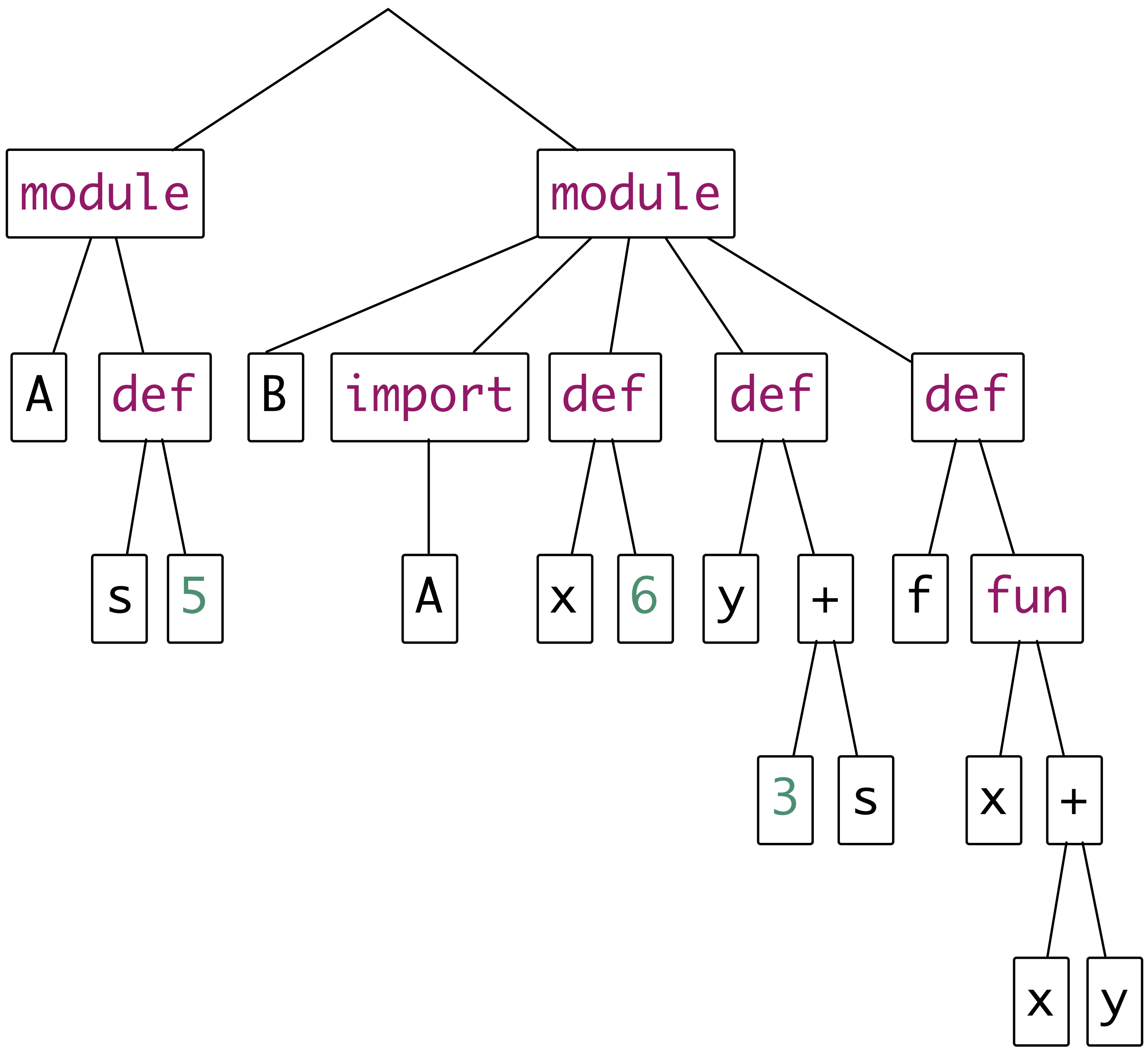
[ESOP15]

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 }
}
```

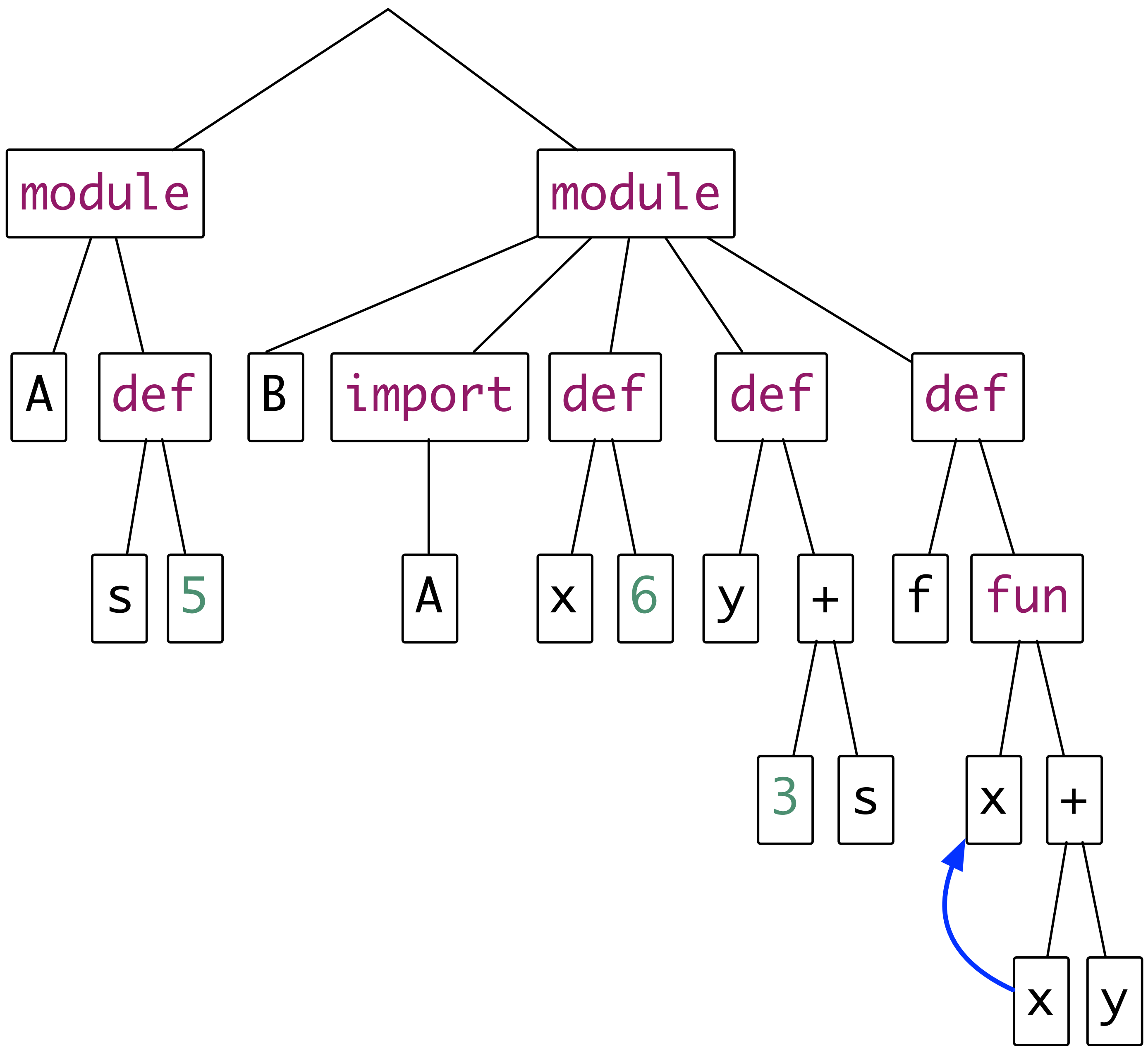
Static Name Resolution

```
module A {
    def s = 5
}

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



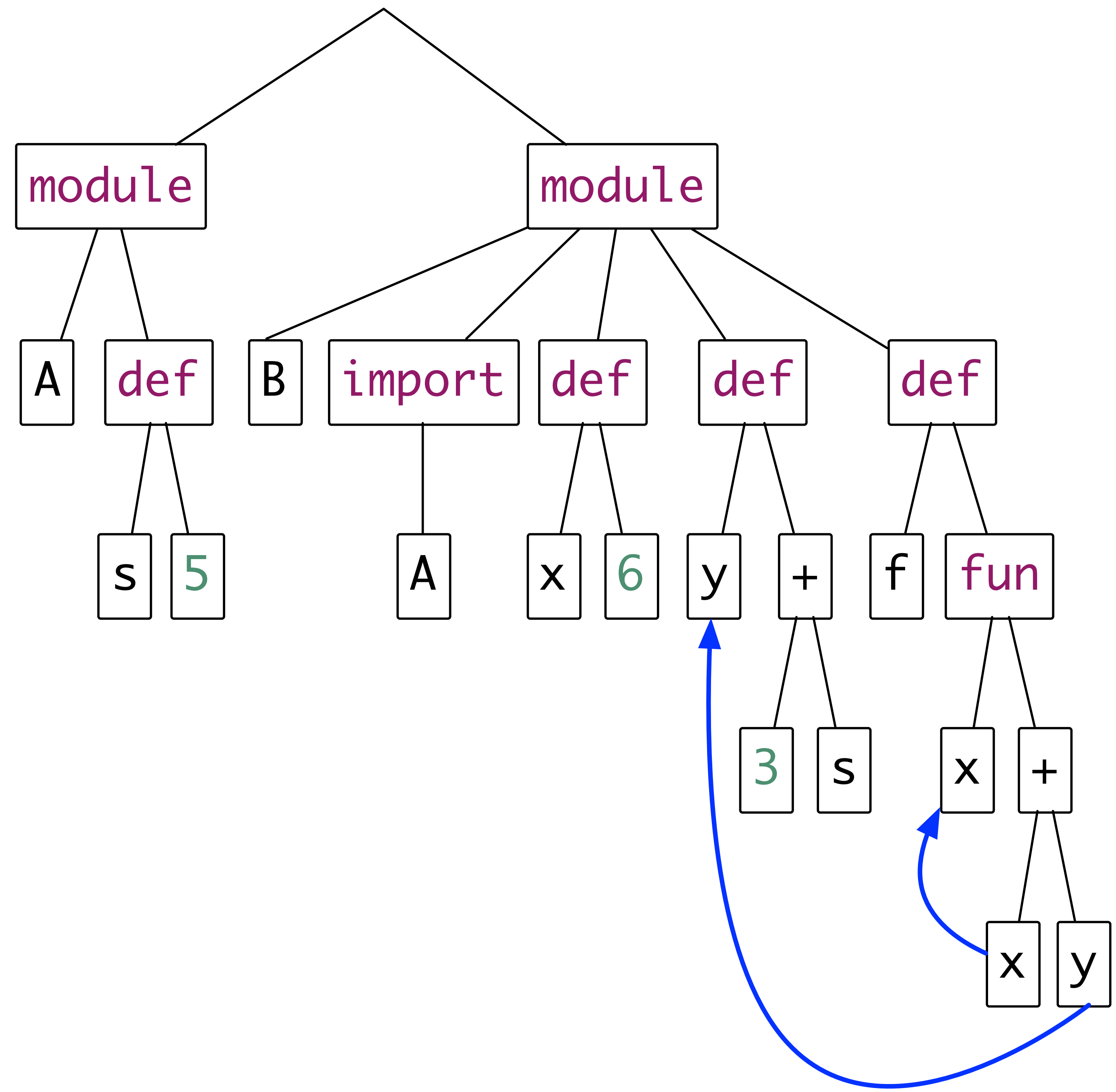
Static Name Resolution

```
module A {
    def s = 5
}

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



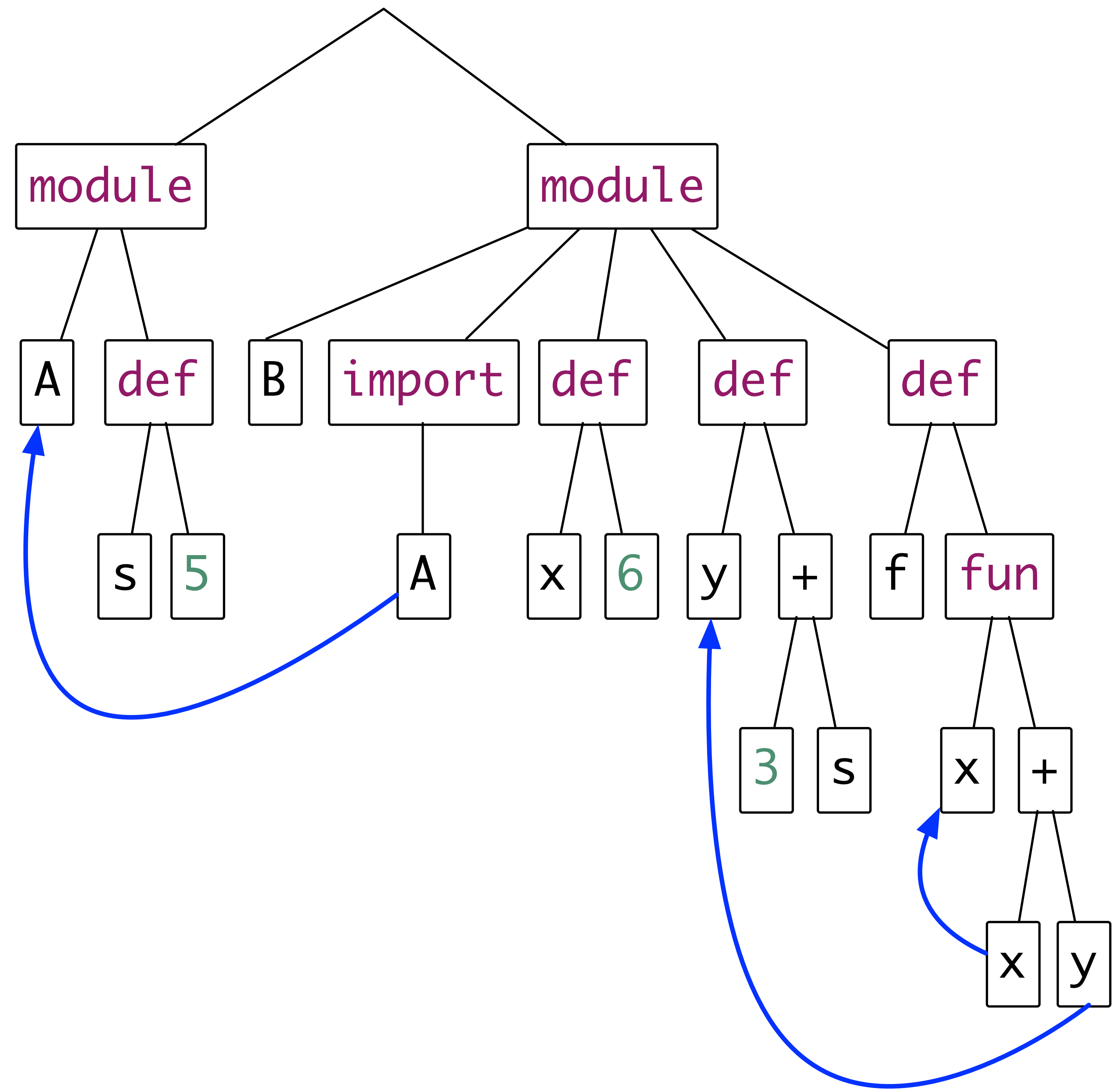
Static Name Resolution

```
module A {
    def s = 5
}

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



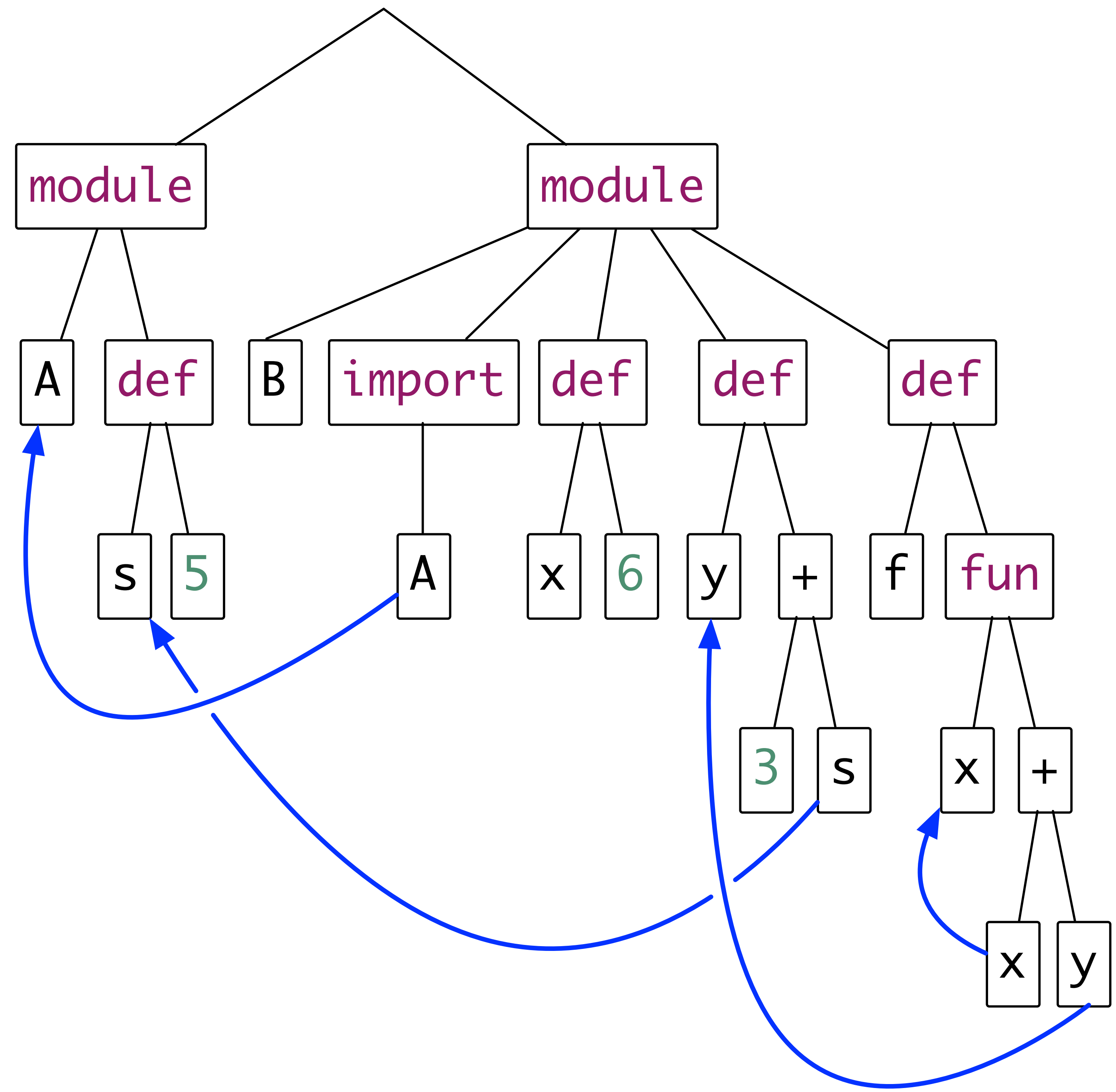
Static Name Resolution

```
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 ??

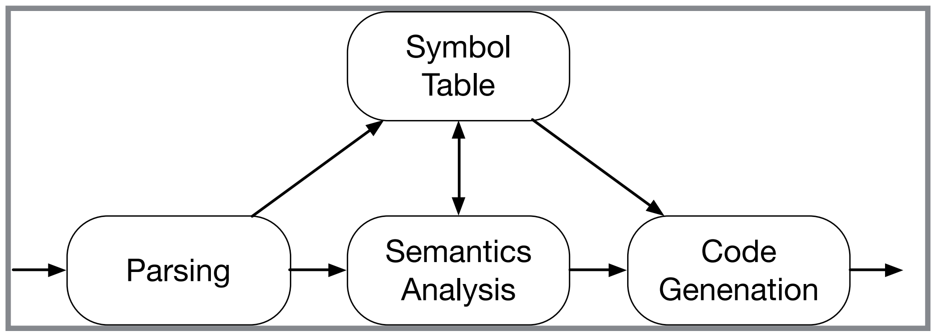
Name Resolution is Pervasive

Name Resolution is Pervasive

Appears in many different artifacts…

Name Resolution is Pervasive

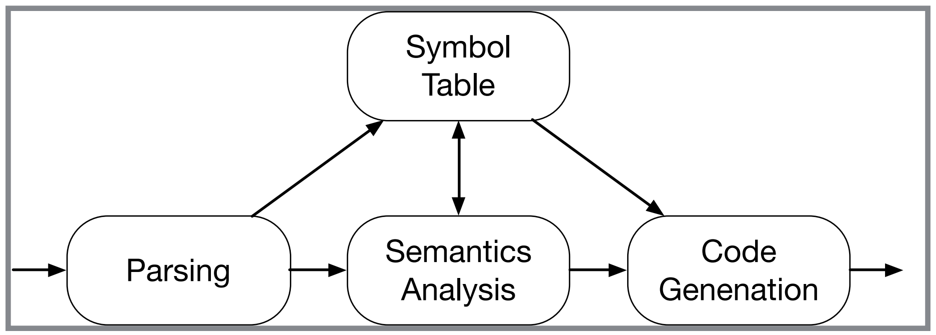
Appears in many different artifacts…



Compiler

Name Resolution is Pervasive

Appears in many different artifacts…



Compiler

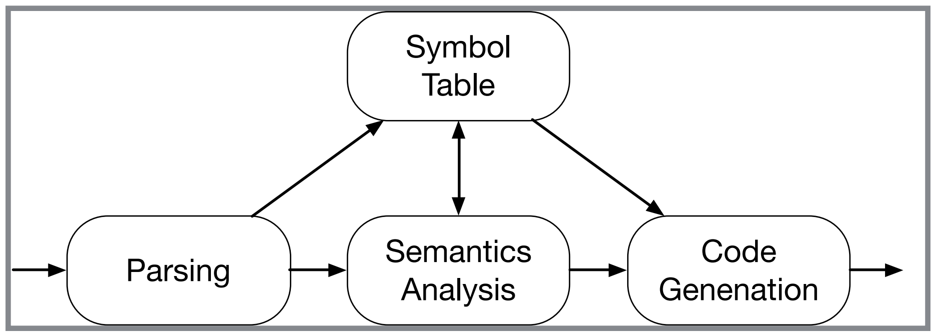
$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

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

Semantics

Name Resolution is Pervasive

Appears in many different artifacts…

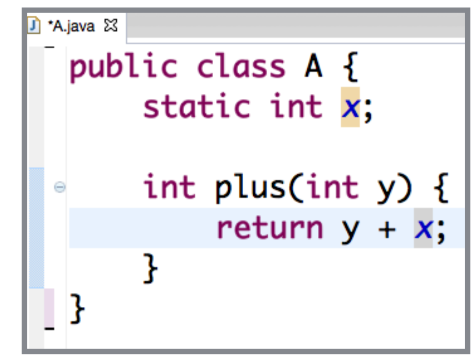


Compiler

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

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

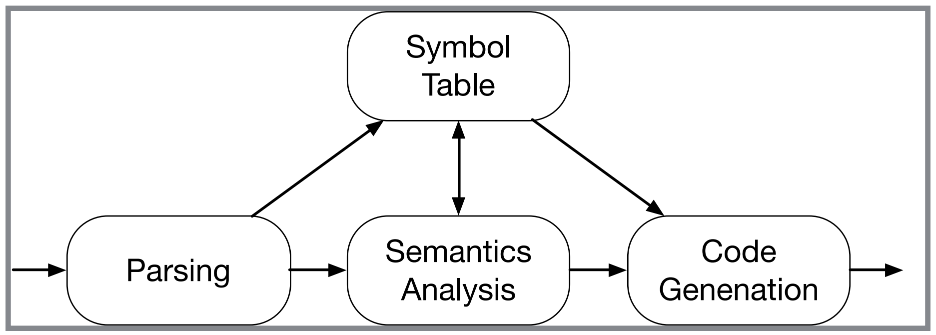
Semantics



IDE

Name Resolution is Pervasive

Appears in many different artifacts…



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

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

*A.java

```
public class A {
    static int x;

    int plus(int y) {
        return y + x;
    }
}
```

Compiler

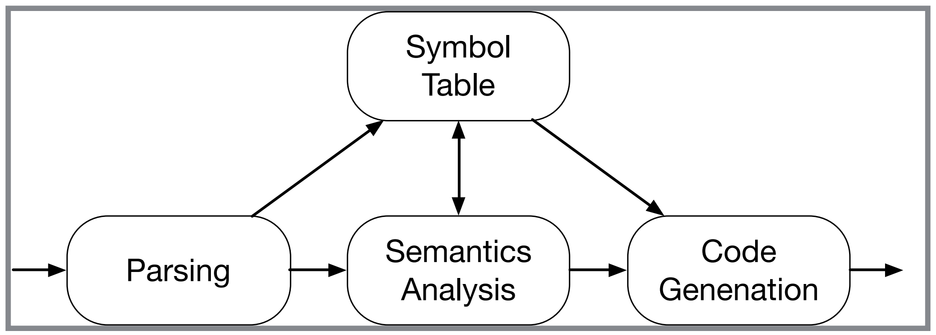
Semantics

IDE

… with rules encoded in many different ad-hoc ways

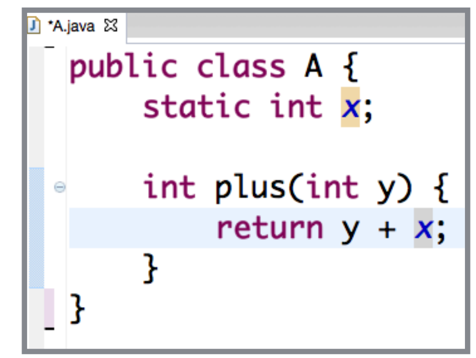
Name Resolution is Pervasive

Appears in many different artifacts…



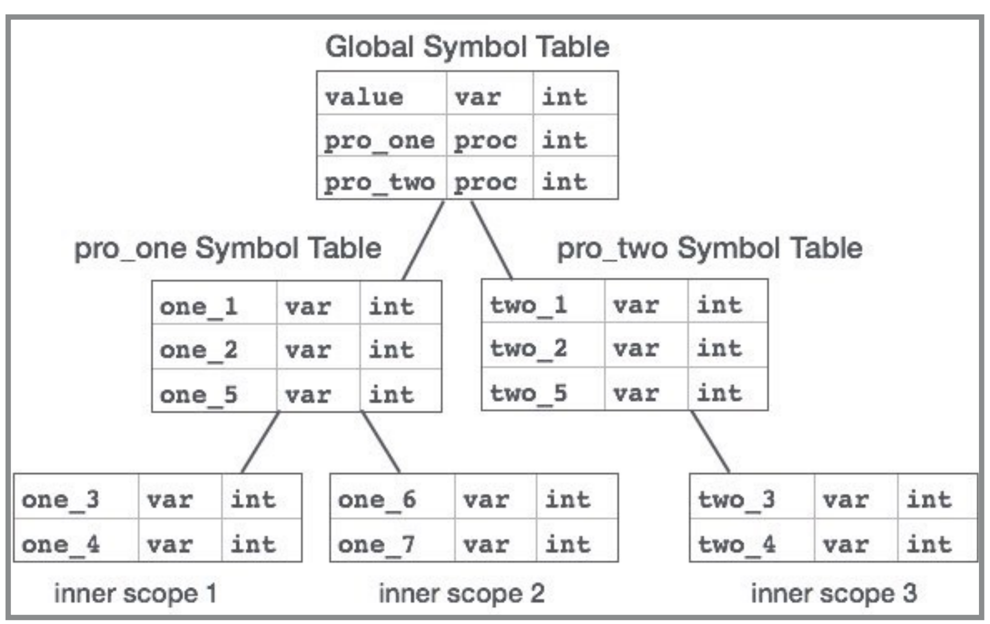
$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

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



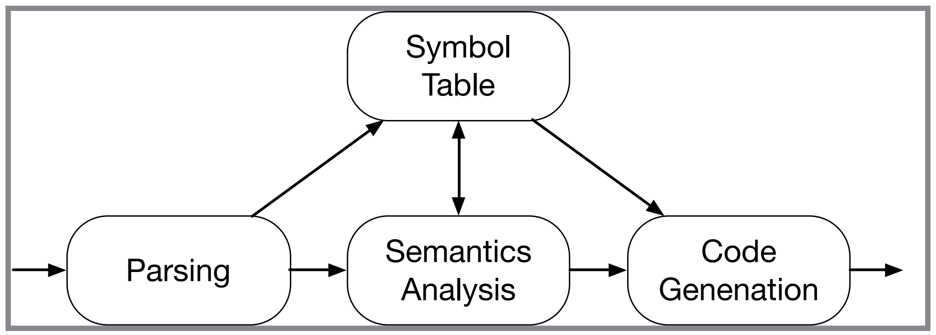
Compiler Semantics IDE

… with rules encoded in many different ad-hoc ways



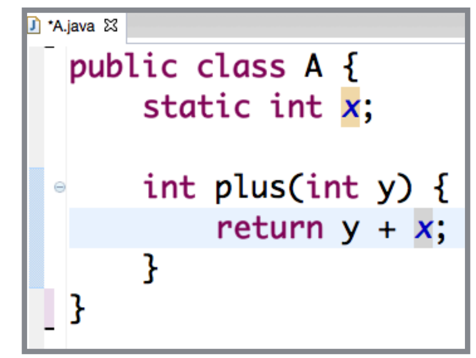
Name Resolution is Pervasive

Appears in many different artifacts…



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

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

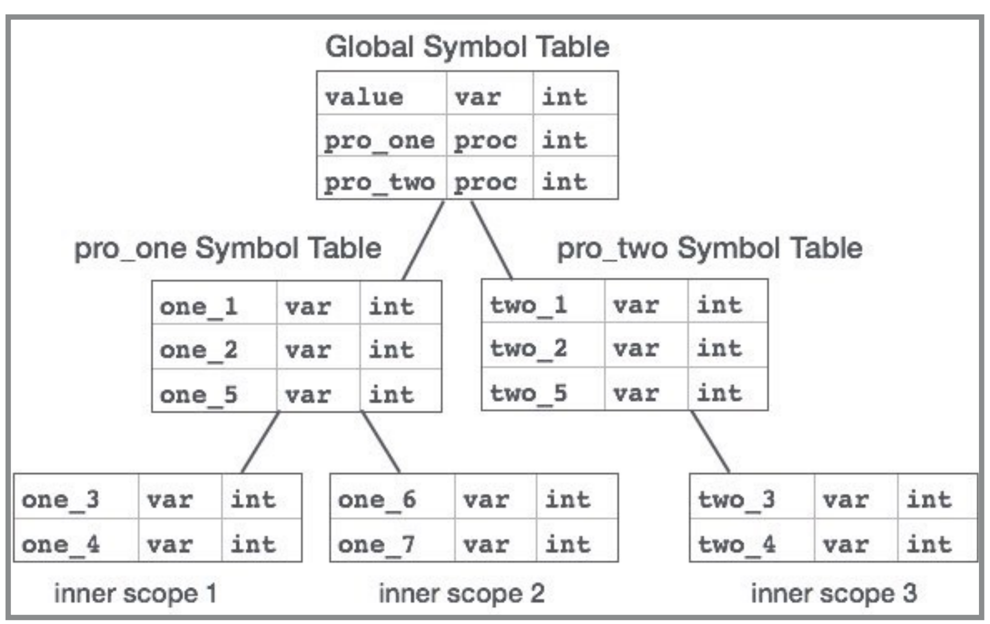


Compiler

Semantics

IDE

… with rules encoded in many different ad-hoc ways

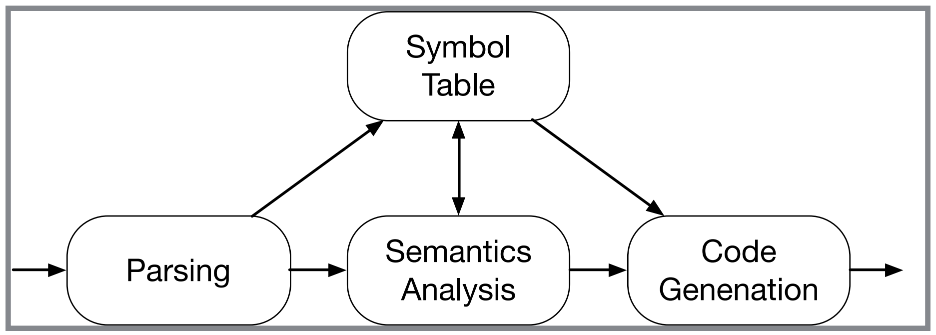


$\boxed{\texttt{x:int, }\Gamma}$

$\boxed{\texttt{[3/x].}\sigma}$

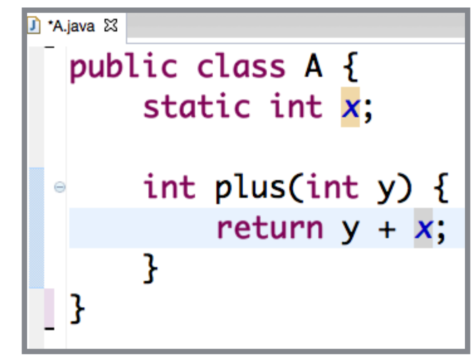
Name Resolution is Pervasive

Appears in many different artifacts…



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

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

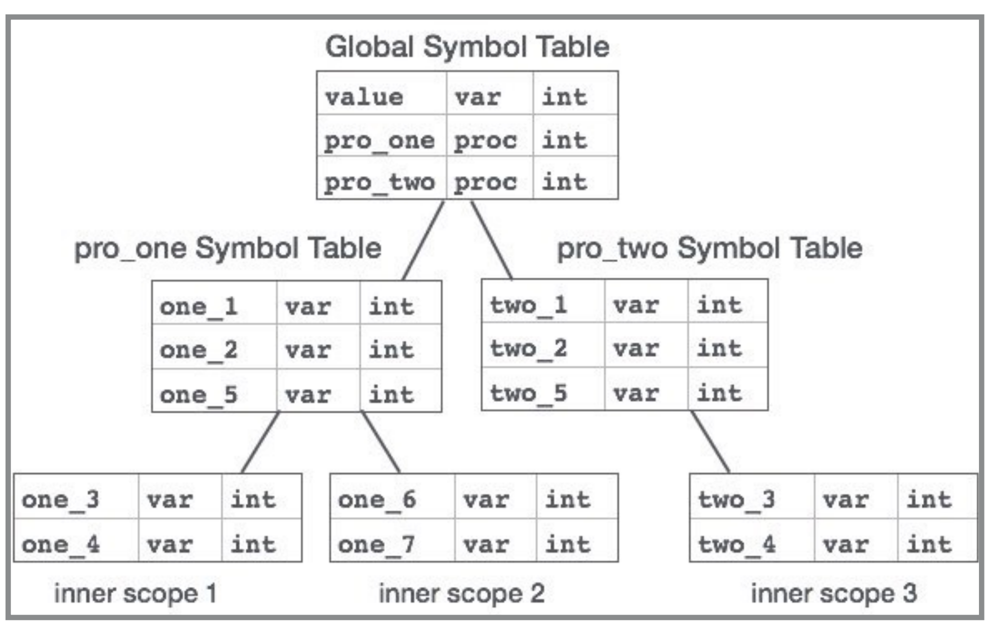


Compiler

Semantics

IDE

… with rules encoded in many different ad-hoc ways



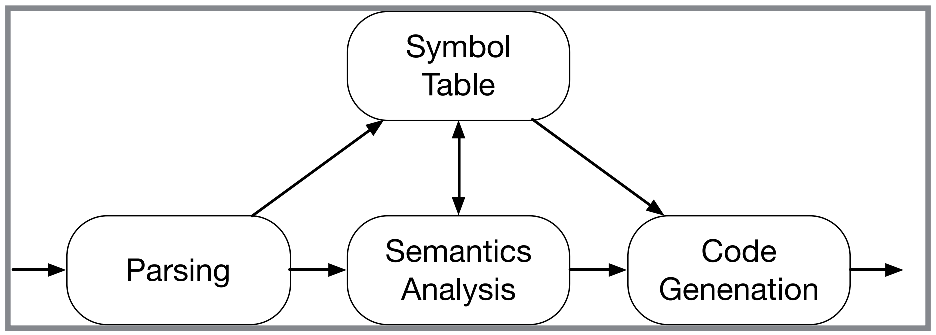
`x:int, Γ`

`lookup(xᵢ)`

`[3/x].σ`

Name Resolution is Pervasive

Appears in many different artifacts…

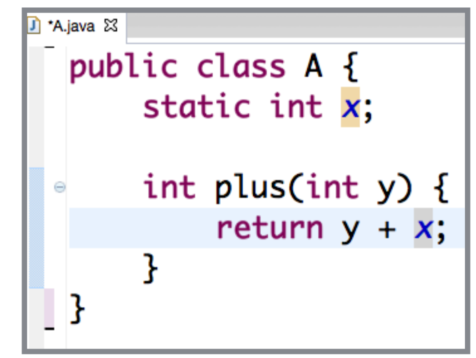


Compiler

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

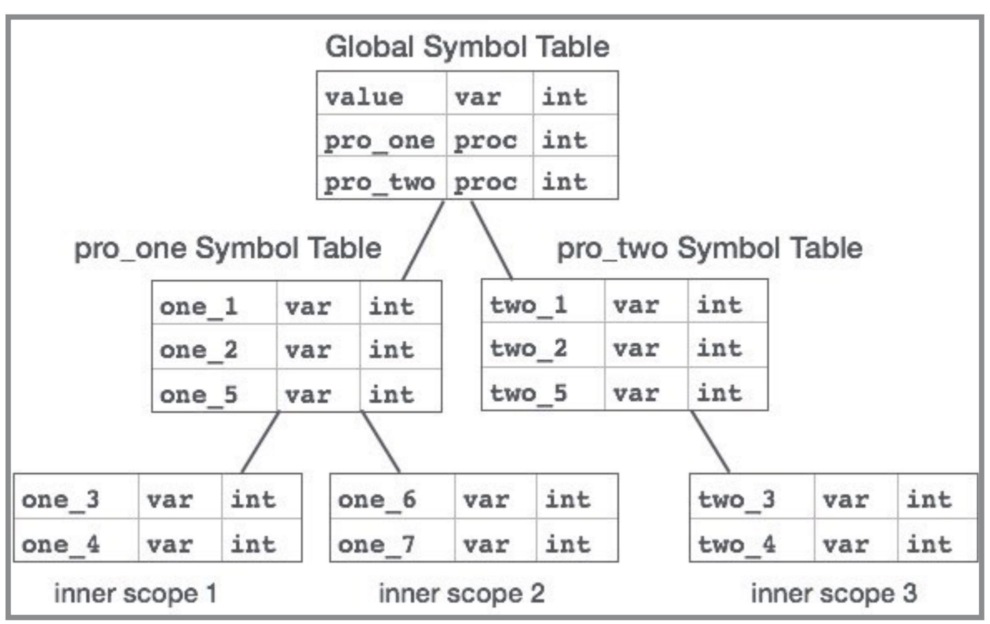
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Semantics



IDE

… with rules encoded in many different ad-hoc ways



`x:int, Γ`

`lookup(x_i)`

`[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	*A standard formalism*	*Supports*

```
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

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

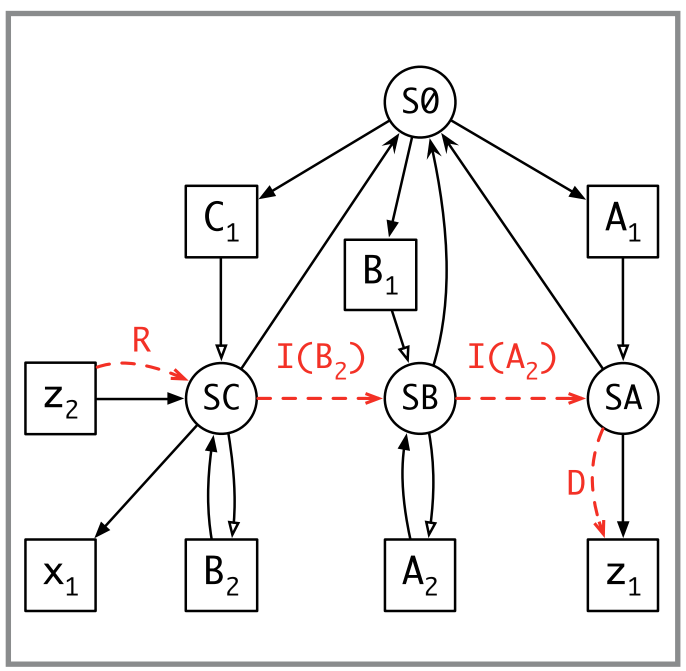
Scope Graphs

A Theory of Name Resolution

For ***statically lexically scoped*** *languages*

A unique representation

A standard formalism

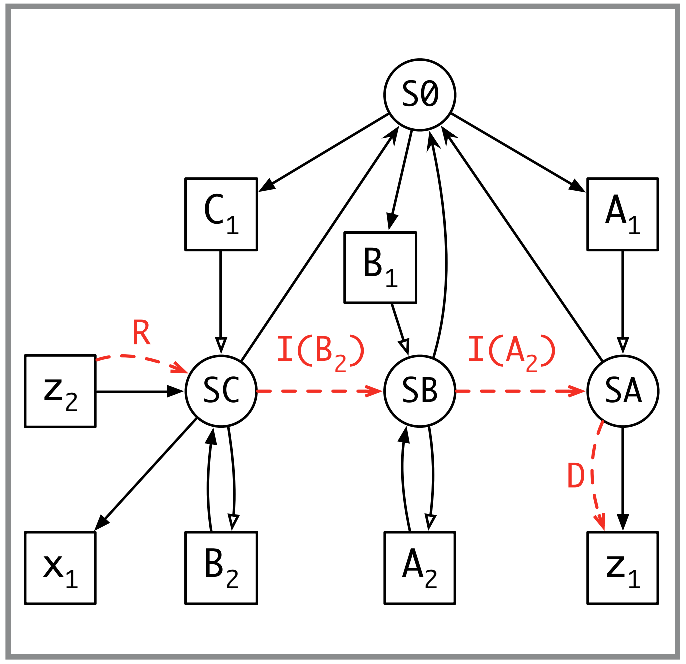


Scope Graphs

A Theory of Name Resolution

For ***statically lexically scoped*** *languages*

A unique representation



A standard formalism

Scope Graphs

Supports

Resolution

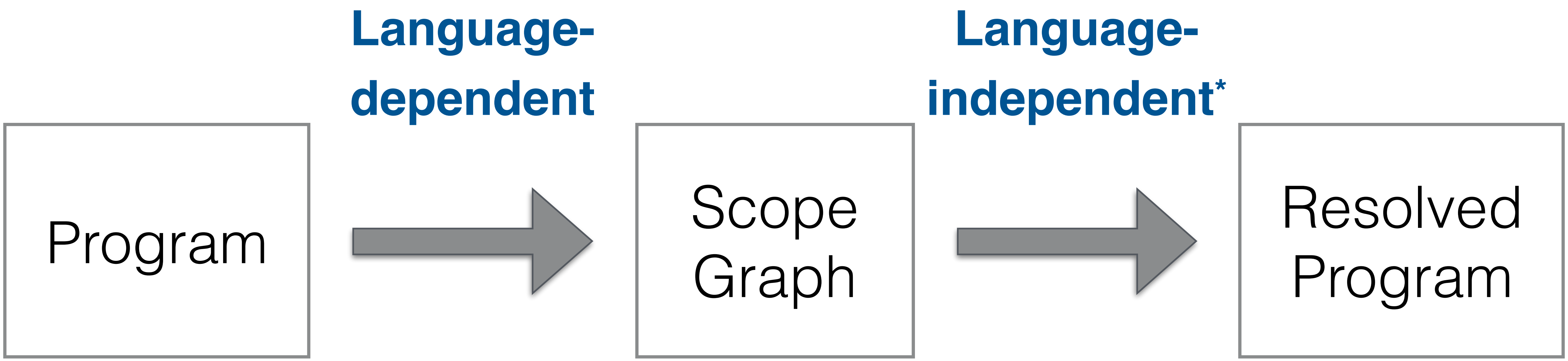
α-equivalence

IDE Navigation

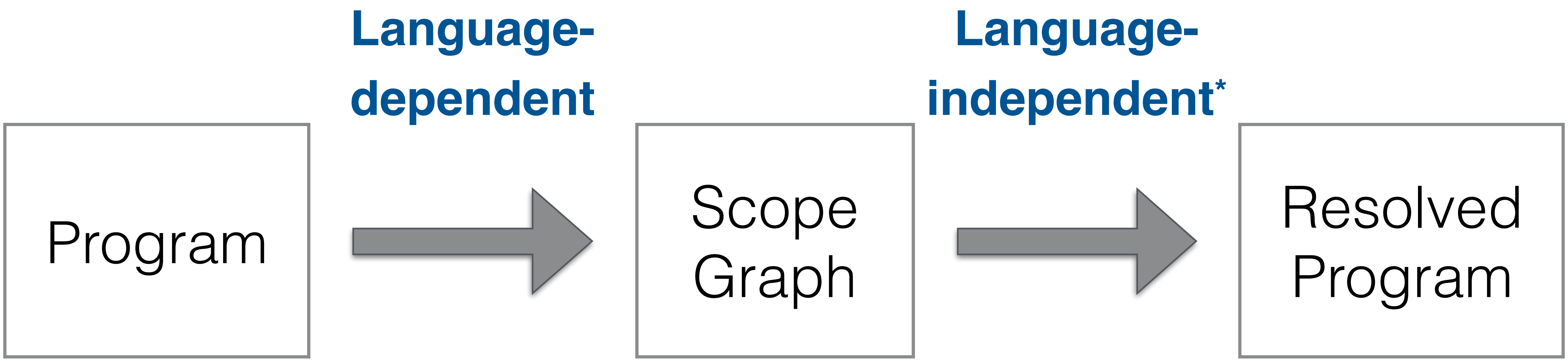
Refactoring tools

Reasoning tools

Resolution Scheme



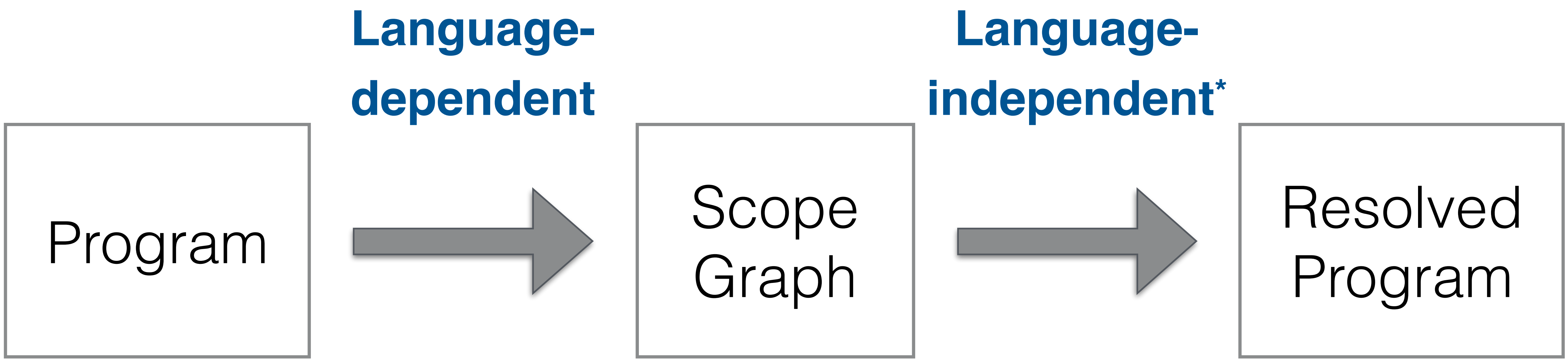
Resolution Scheme



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

$$\texttt{def } y_1 = x_2 + 1$$
$$\texttt{def } x_1 = 5$$

Simple Scopes

```
def y₁ = x₂ + 1
def x₁ = 5
```

Simple Scopes

S0

$\texttt{def}\ y_1 = x_2 + 1$

$\texttt{def}\ x_1 = 5$

Simple Scopes

S0

def $y_1 = x_2 + 1$

def $x_1 = 5$

S0

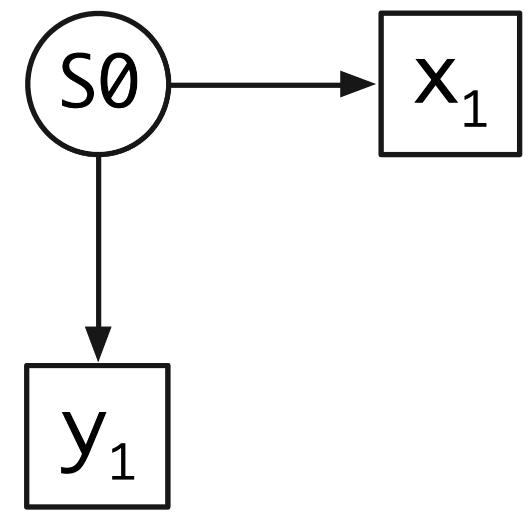
S Scope

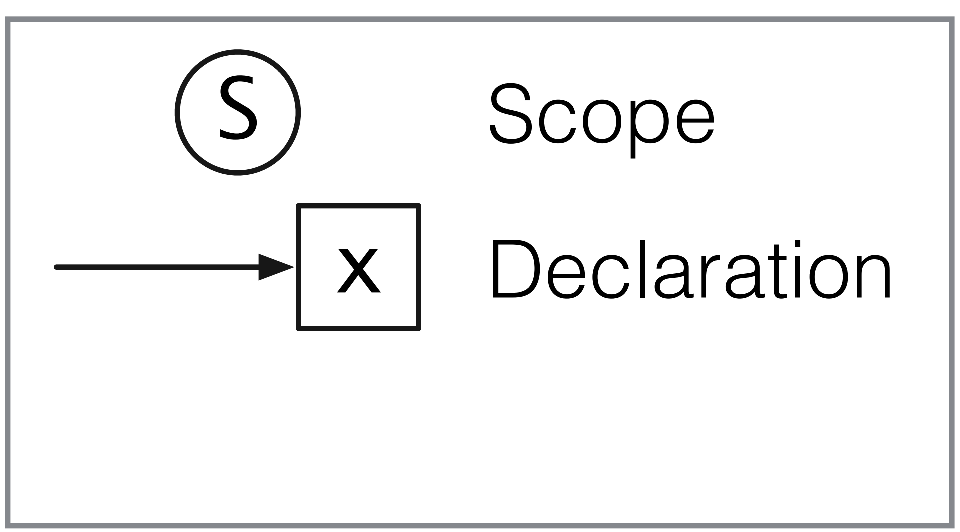
Simple Scopes

S0

$\mathtt{def}\ y_1 = x_2 + 1$

$\mathtt{def}\ x_1 = 5$



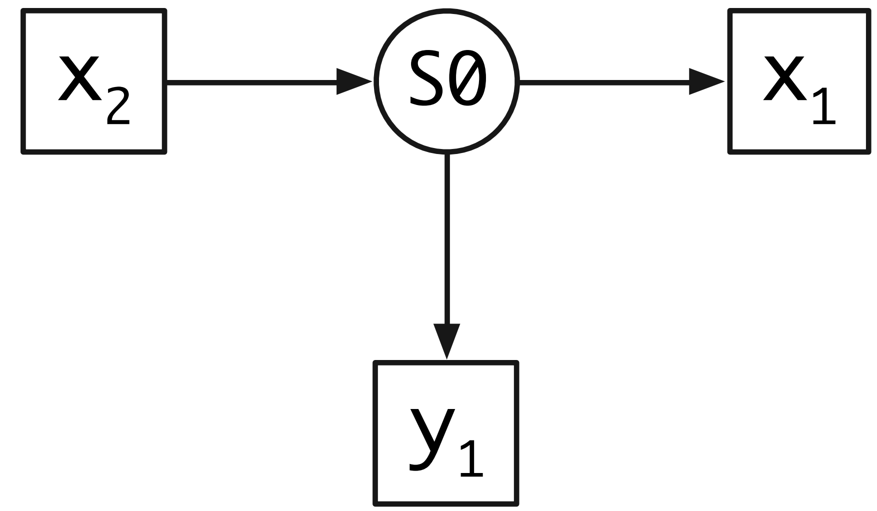


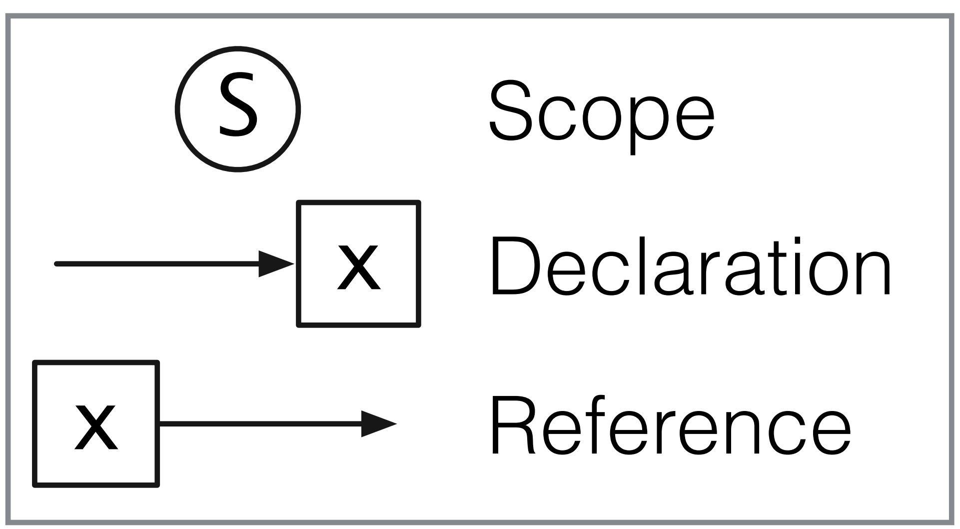
Simple Scopes

S0

$\texttt{def}\ y_1 = x_2 + 1$

$\texttt{def}\ x_1 = 5$

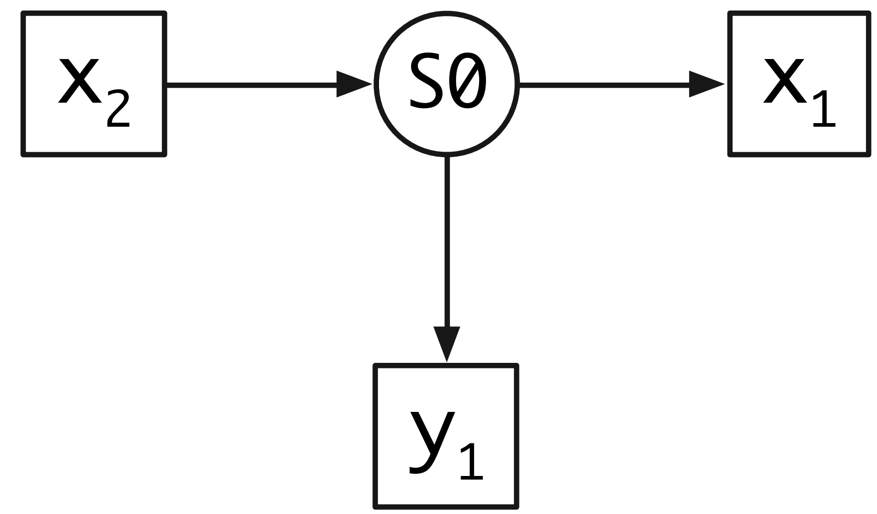


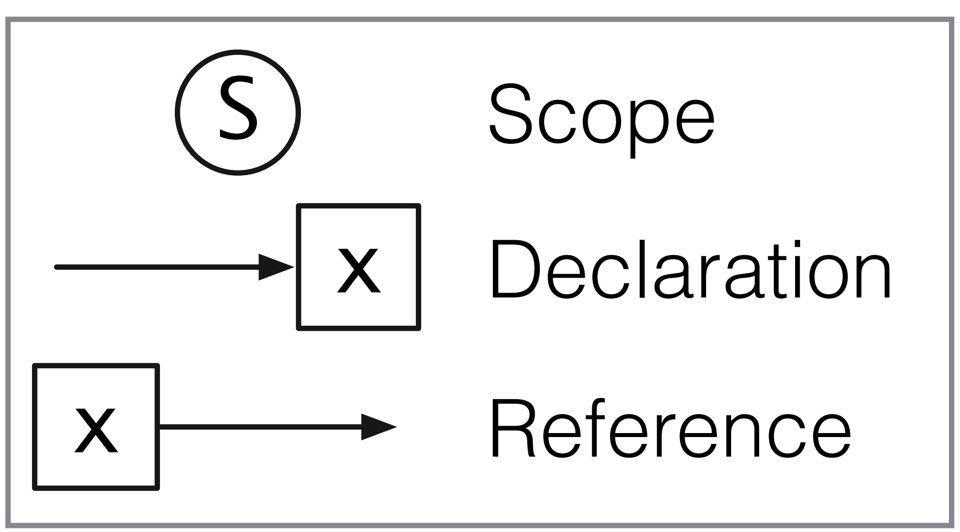


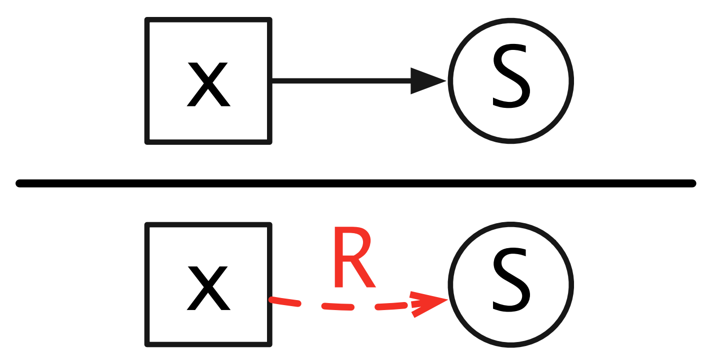
Simple Scopes

S0

def $y_1 = x_2 + 1$
def $x_1 = 5$







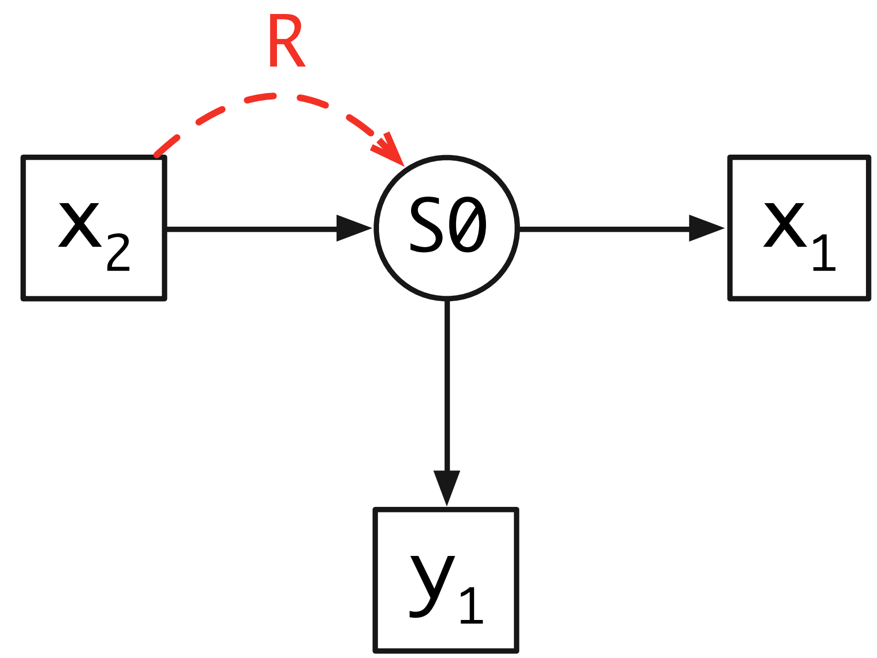
Reference Step

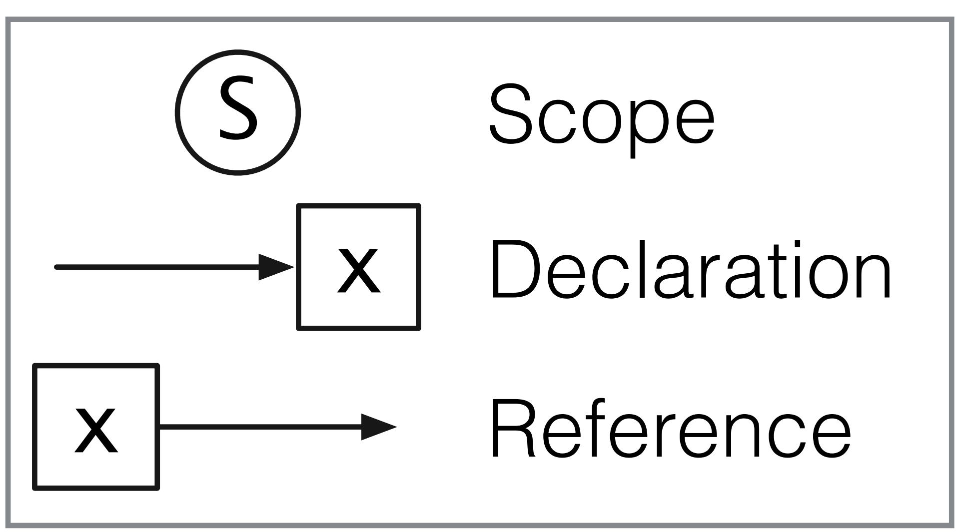
Simple Scopes

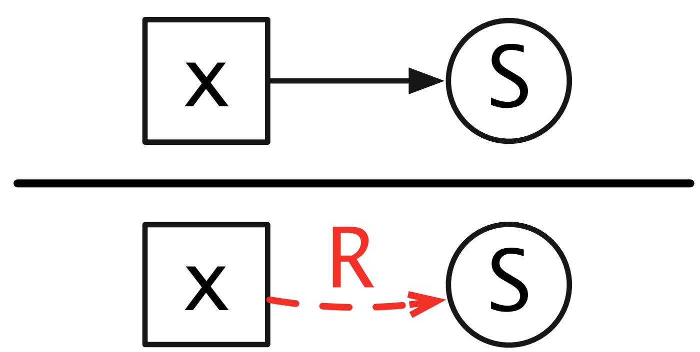
S0

$\mathtt{def}\ y_1 = x_2 + 1$

$\mathtt{def}\ x_1 = 5$





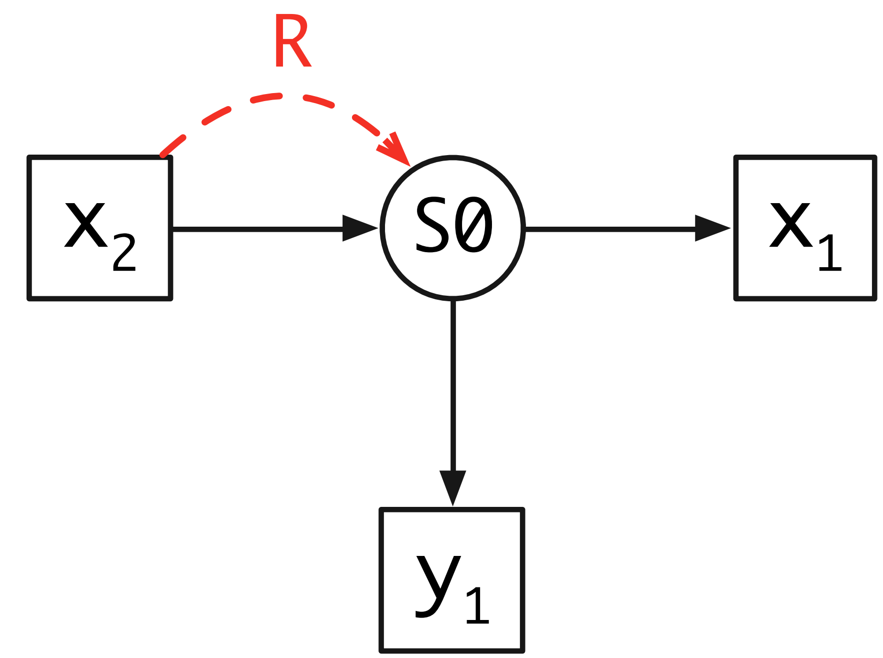


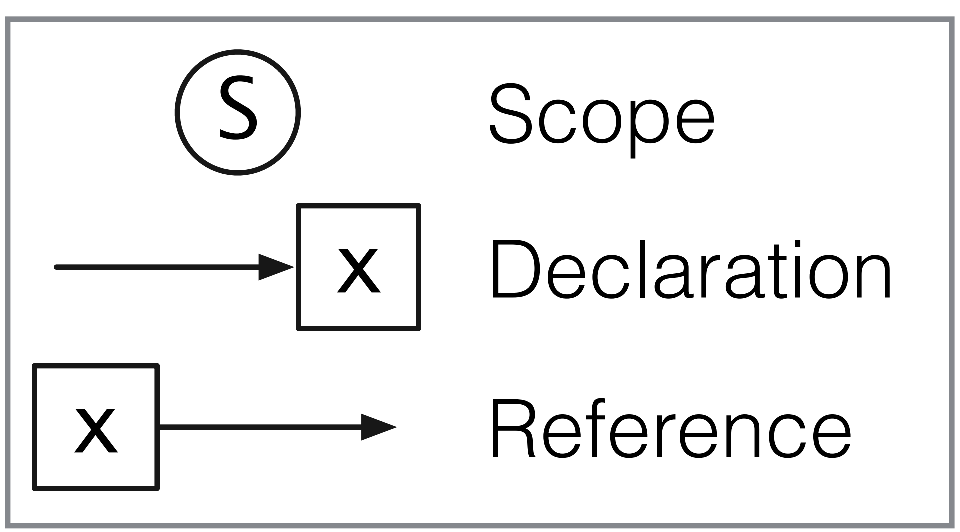
Reference Step

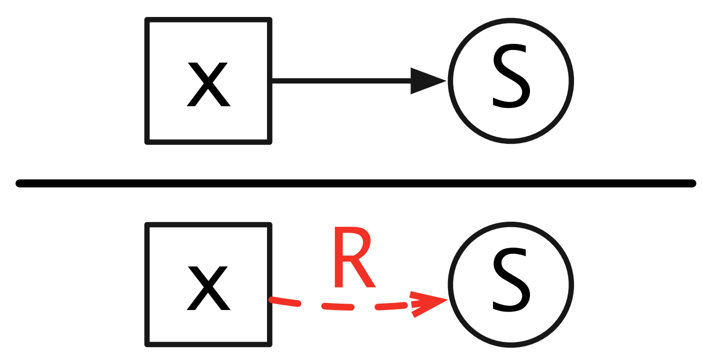
Simple Scopes

S0

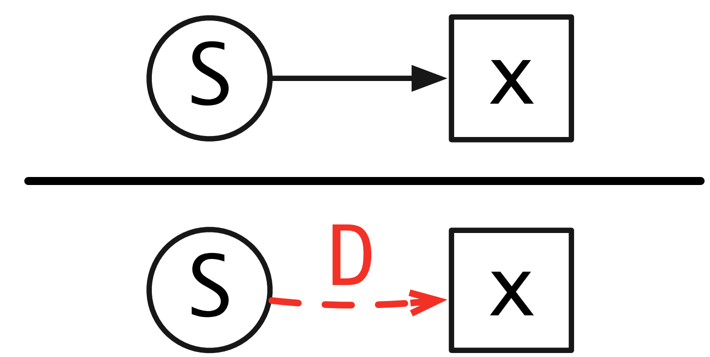
def y_1 = x_2 + 1
def x_1 = 5







Reference Step



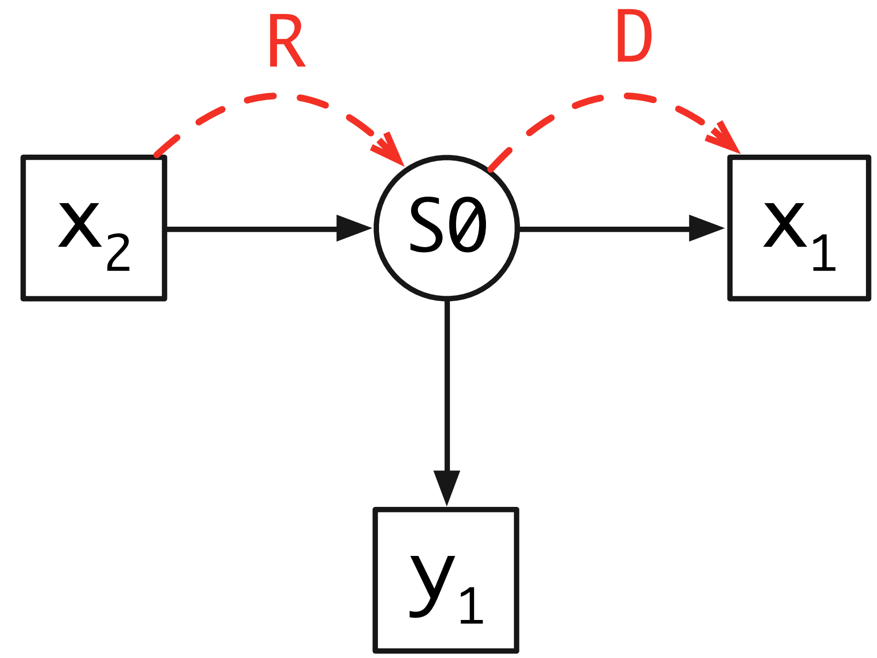
Declaration Step

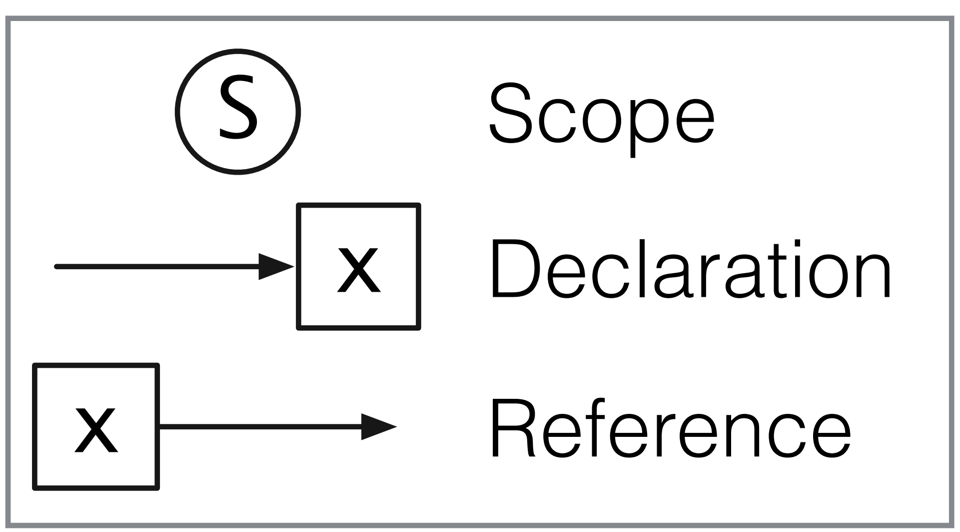
Simple Scopes

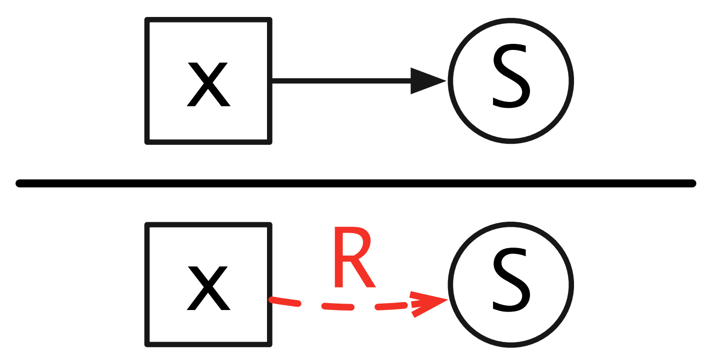
S0

def $y_1 = x_2 + 1$

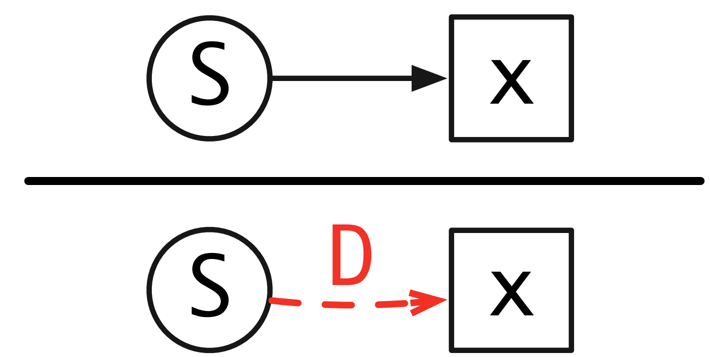
def $x_1 = 5$







Reference Step



Declaration Step

Ambiguous Resolutions

S0

```
def x₁ = 5

def x₂ = 3

def z₁ = x₃ + 1
```

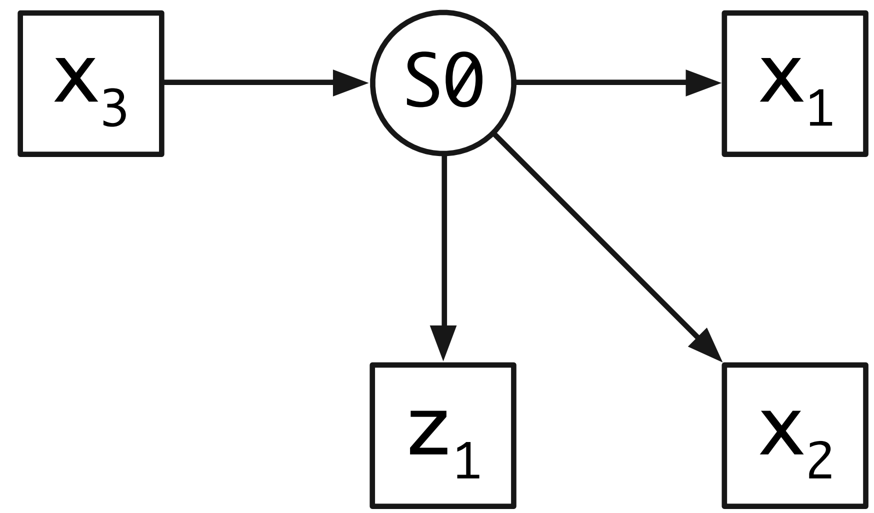
Ambiguous Resolutions

S0

$\mathtt{def}\ \boxed{x_1} = 5$

$\mathtt{def}\ \boxed{x_2} = 3$

$\mathtt{def}\ z_1 = x_3 + 1$



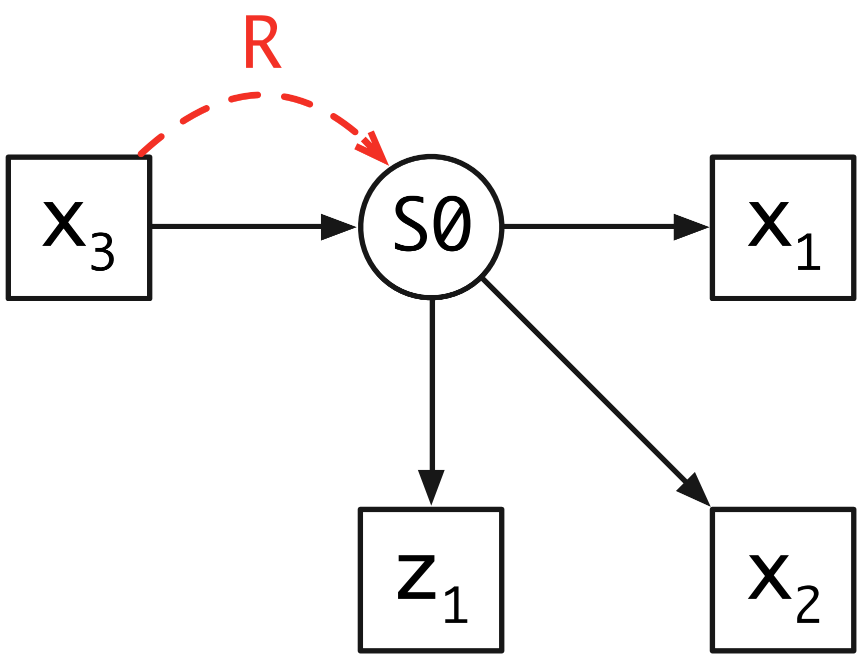
Ambiguous Resolutions

S0

$\texttt{def}\ \boxed{x_1} = 5$

$\texttt{def}\ \boxed{x_2} = 3$

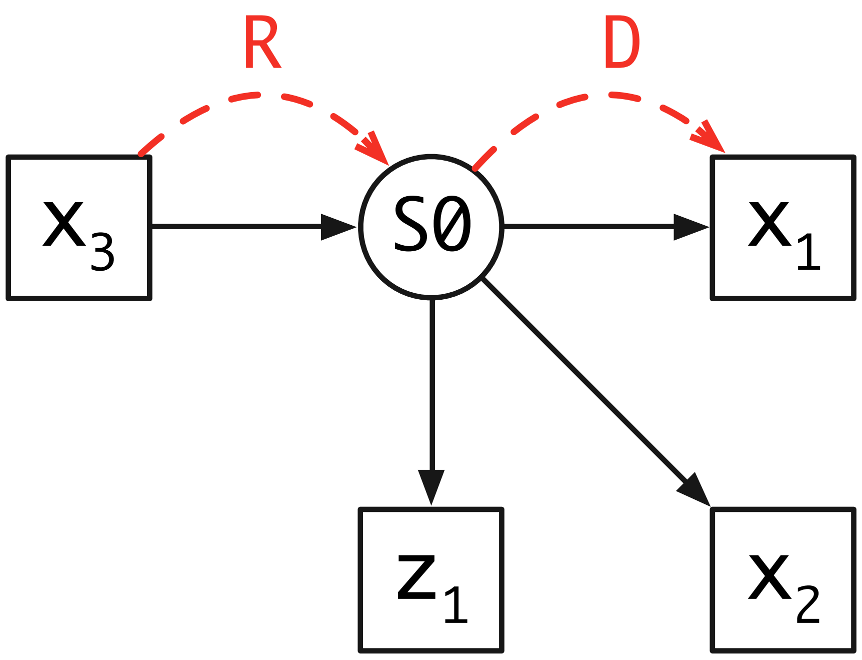
$\texttt{def}\ z_1 = x_3 + 1$



Ambiguous Resolutions

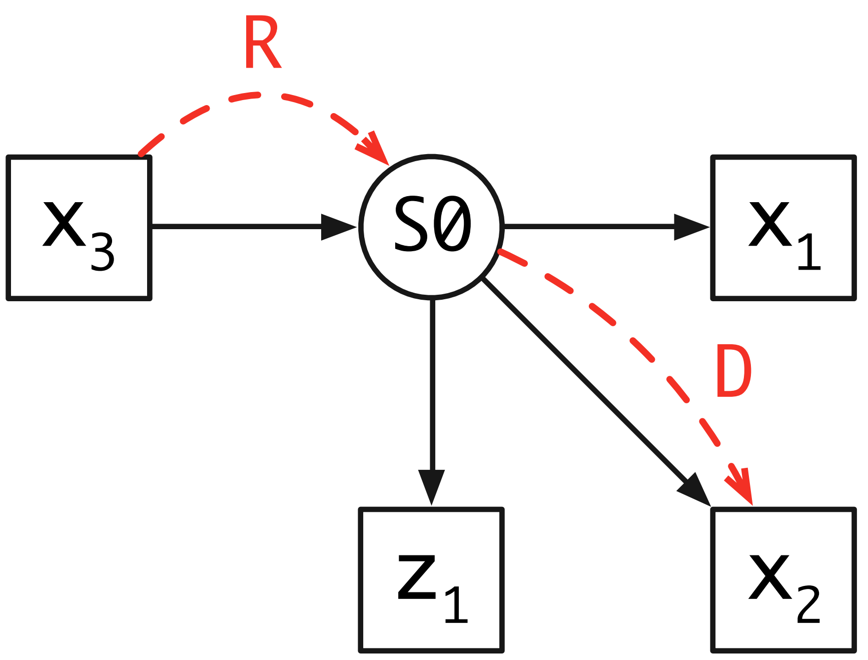
S0

```
def x₁ = 5

def x₂ = 3

def z₁ = x₃ + 1
```



Ambiguous Resolutions

```
def x₁ = 5                    S0

def x₂ = 3

def z₁ = x₃ + 1
```



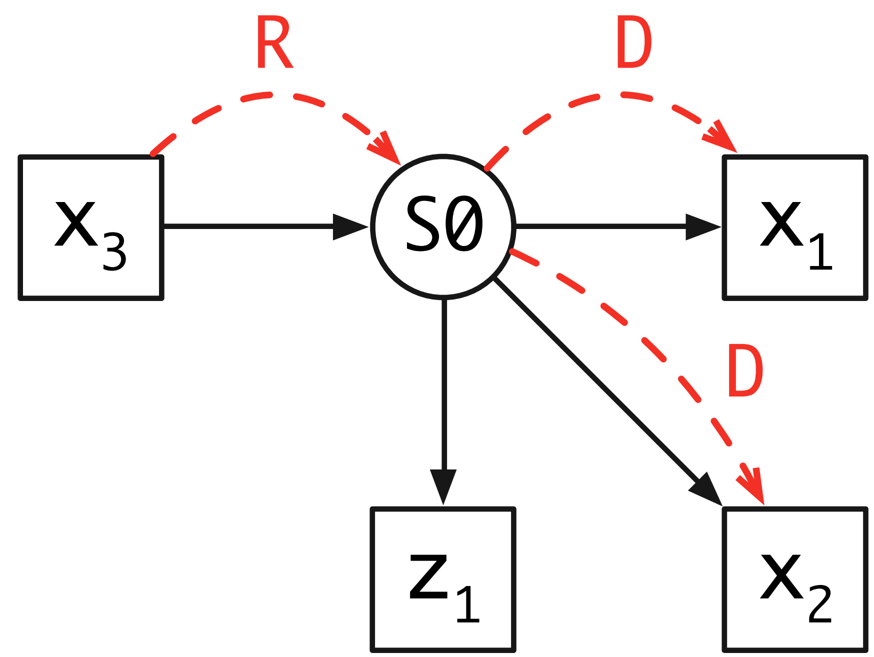
Ambiguous Resolutions

S0

$\texttt{def}\ \boxed{x_1} = 5$

$\texttt{def}\ \boxed{x_2} = 3$

$\texttt{def}\ z_1 = x_3 + 1$



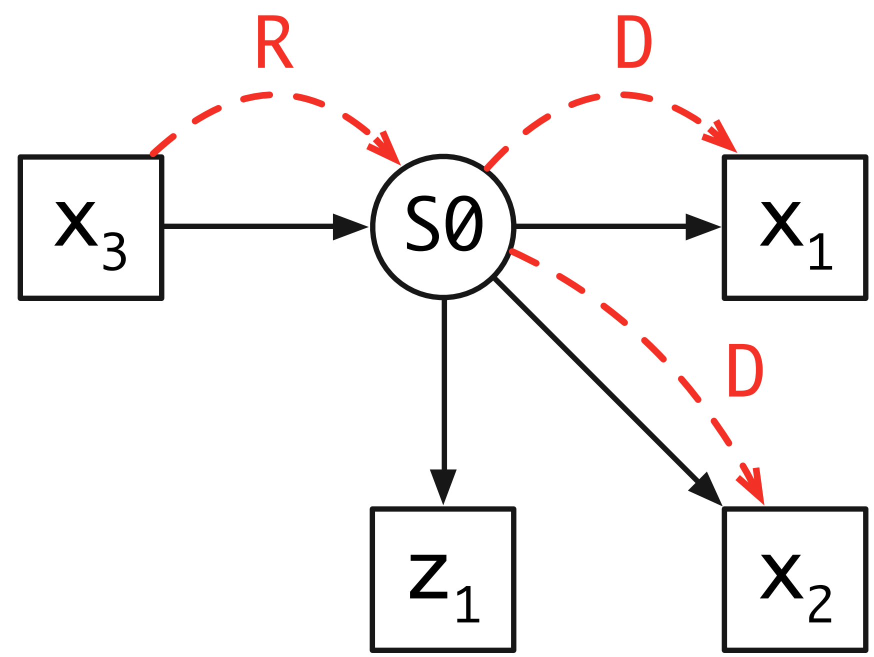
Ambiguous Resolutions

S0

def $\boxed{x_1}$ = 5

def $\boxed{x_2}$ = 3

def z_1 = x_3 + 1



match t with

| A $\boxed{x}$ | B $\boxed{x}$ => …

Lexical Scoping

```
def x1 = z2 5

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

Lexical Scoping

S0

$\texttt{def}\ x_1 = z_2\ 5$

$\texttt{def}\ z_1 =$

$\quad \texttt{fun}\ y_1\ \{$

$\qquad x_2 + y_2$

$\quad \}$

Lexical Scoping

S0

def x_1 = z_2 5

def z_1 =

fun y_1 {   S1

  x_2 + y_2

}

Lexical Scoping

S0

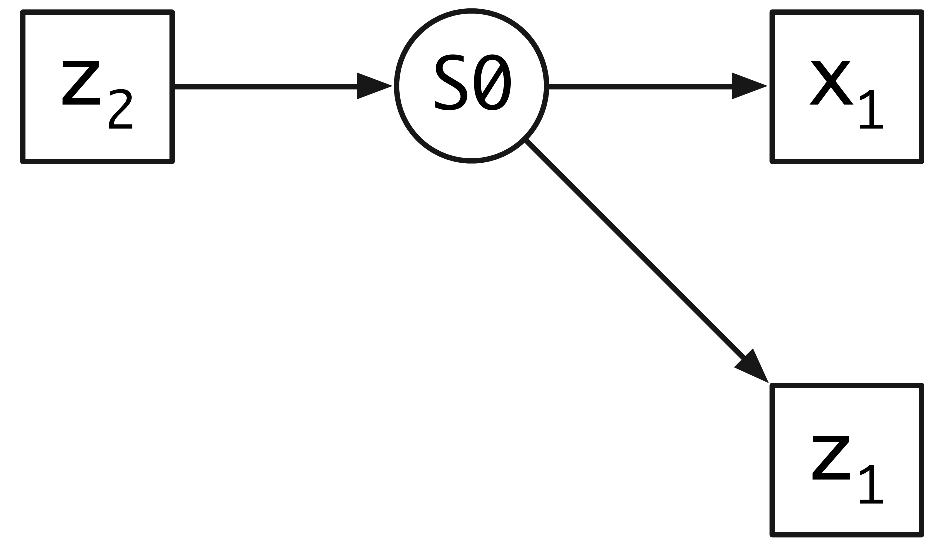
$\mathtt{def}\ x_1 = z_2\ 5$

$\mathtt{def}\ z_1 =$

$\quad \mathtt{fun}\ y_1\ \{$ S1

$\qquad x_2 + y_2$

$\quad \}$



Lexical Scoping

S0

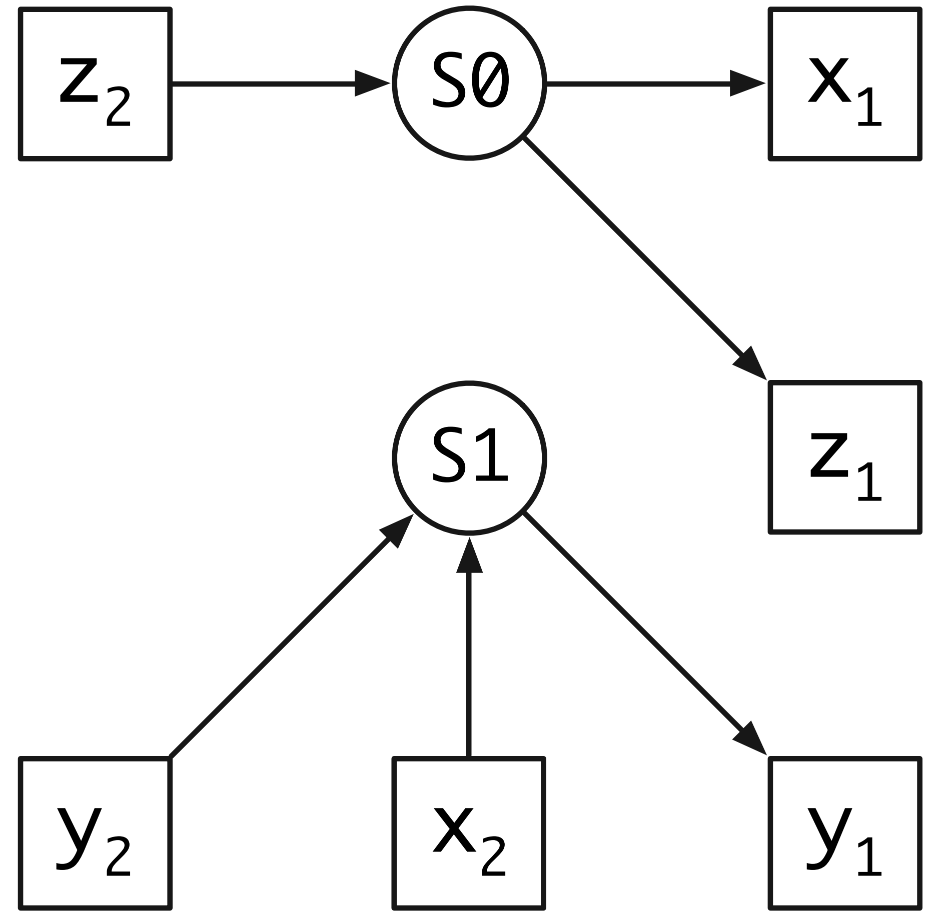
def x_1 = z_2 5

def z_1 =

fun y_1 { S1

x_2 + y_2

}

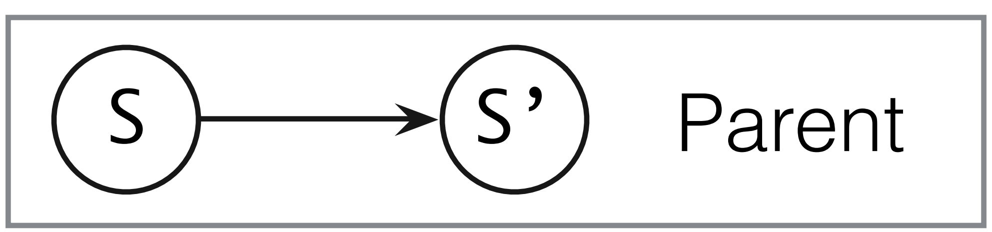


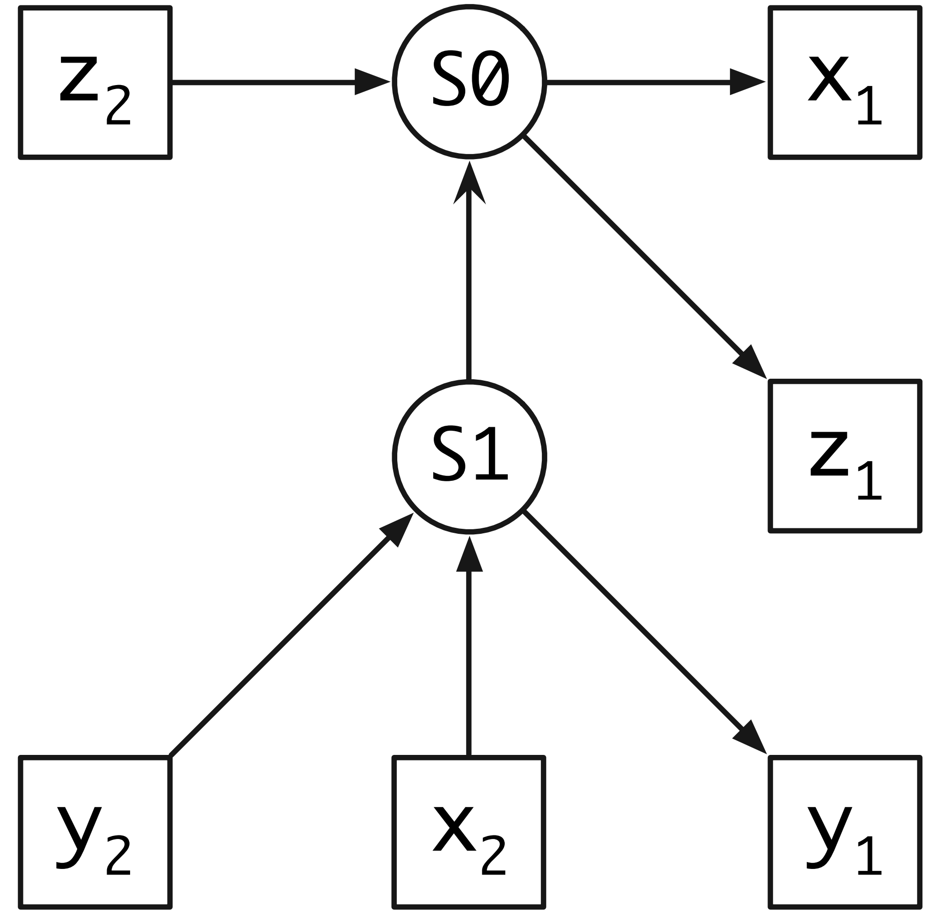
Lexical Scoping

S0

def x_1 = z_2 5

def z_1 =
  fun y_1 {   S1
    x_2 + y_2
  }





Lexical Scoping

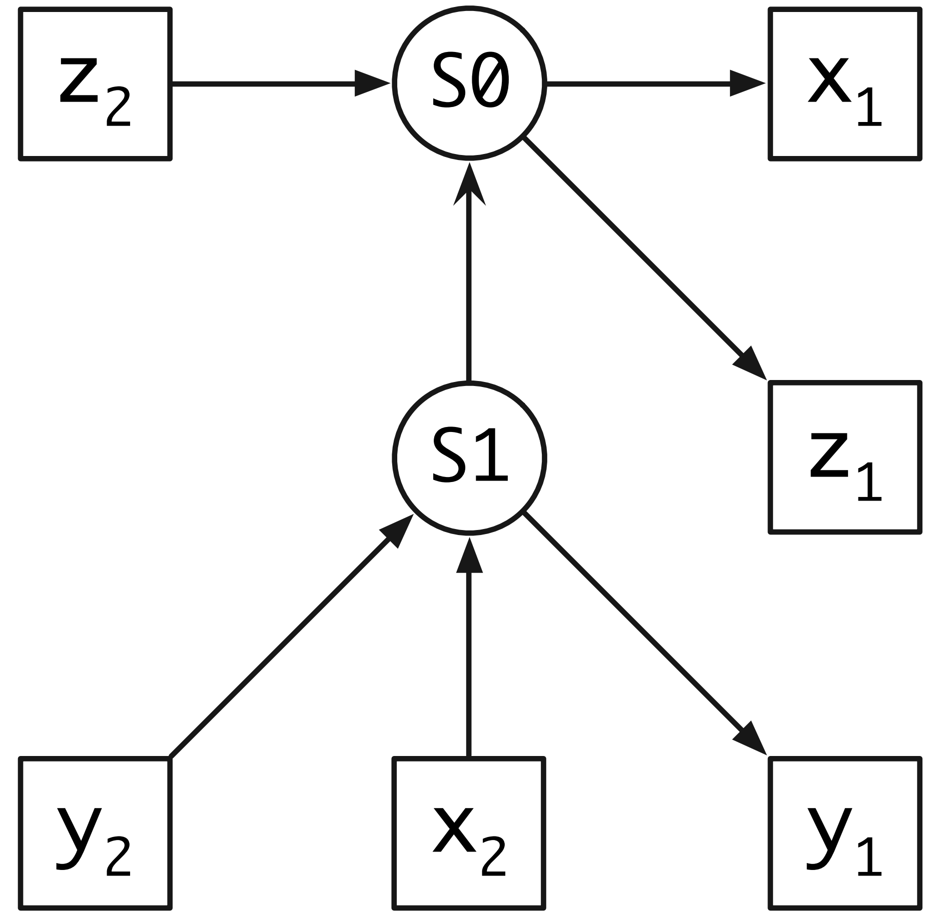
S0

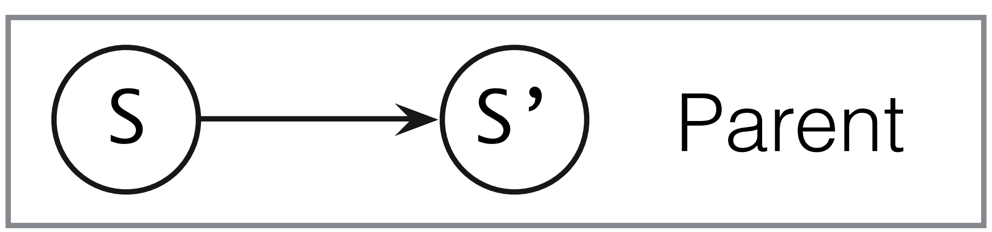
def x_1 = z_2 5

def z_1 =

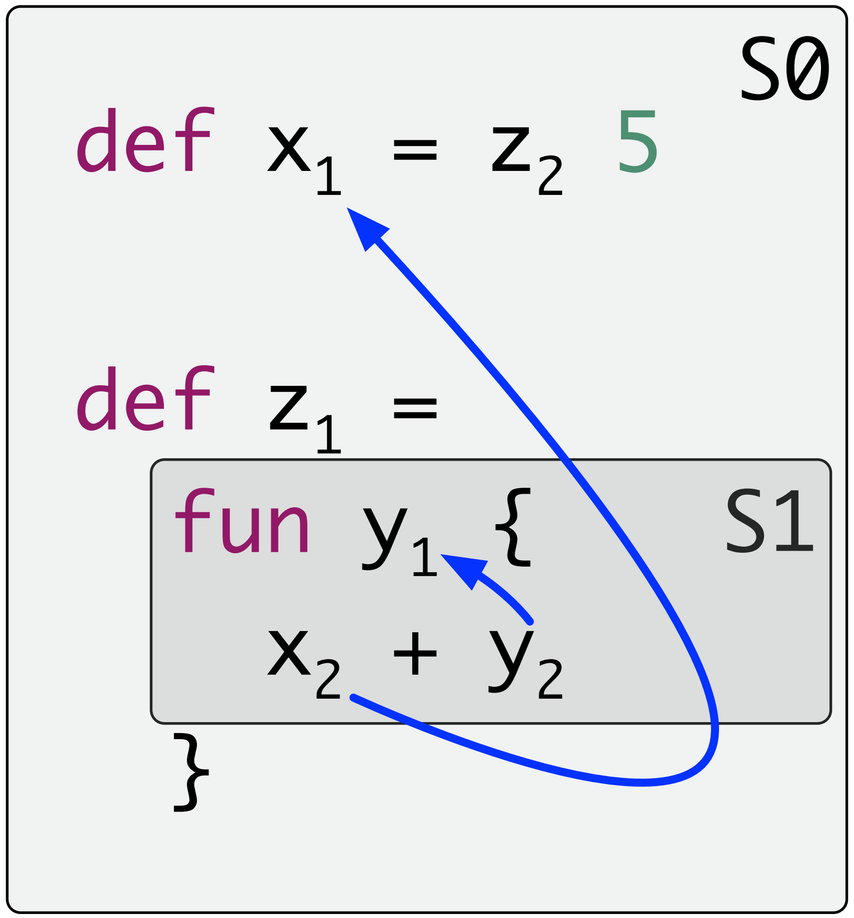
S1

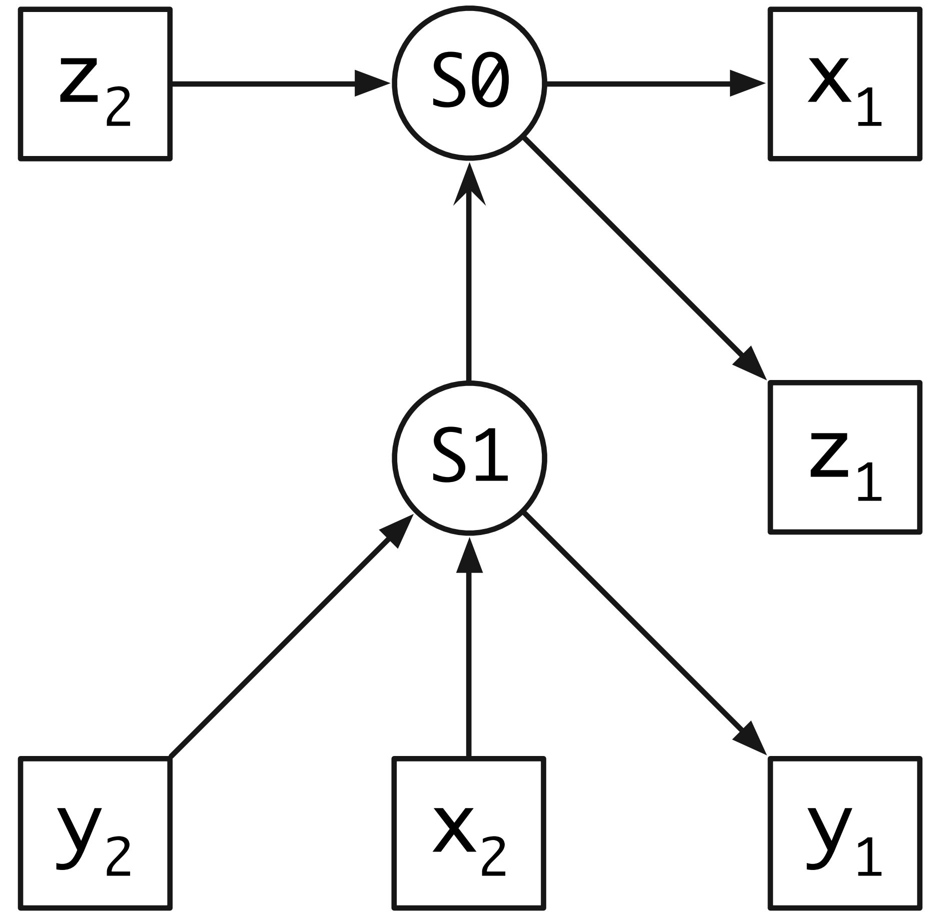
fun y_1 {
  x_2 + y_2
}

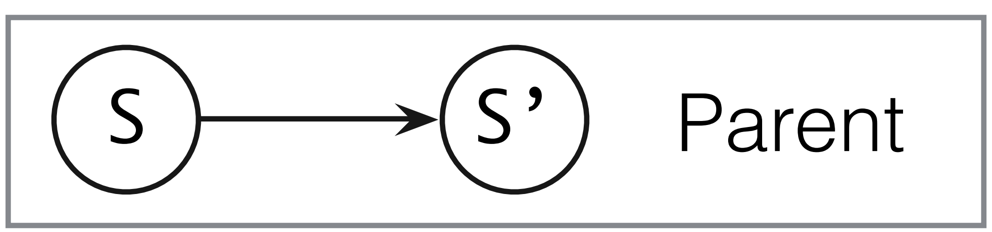




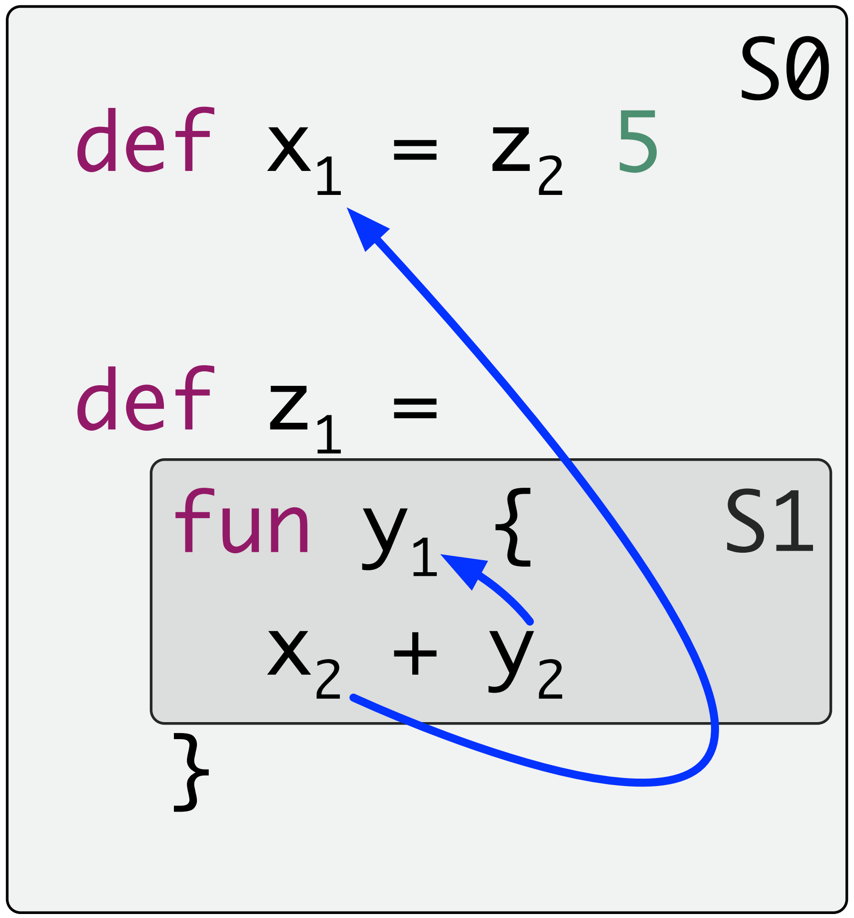
Lexical Scoping

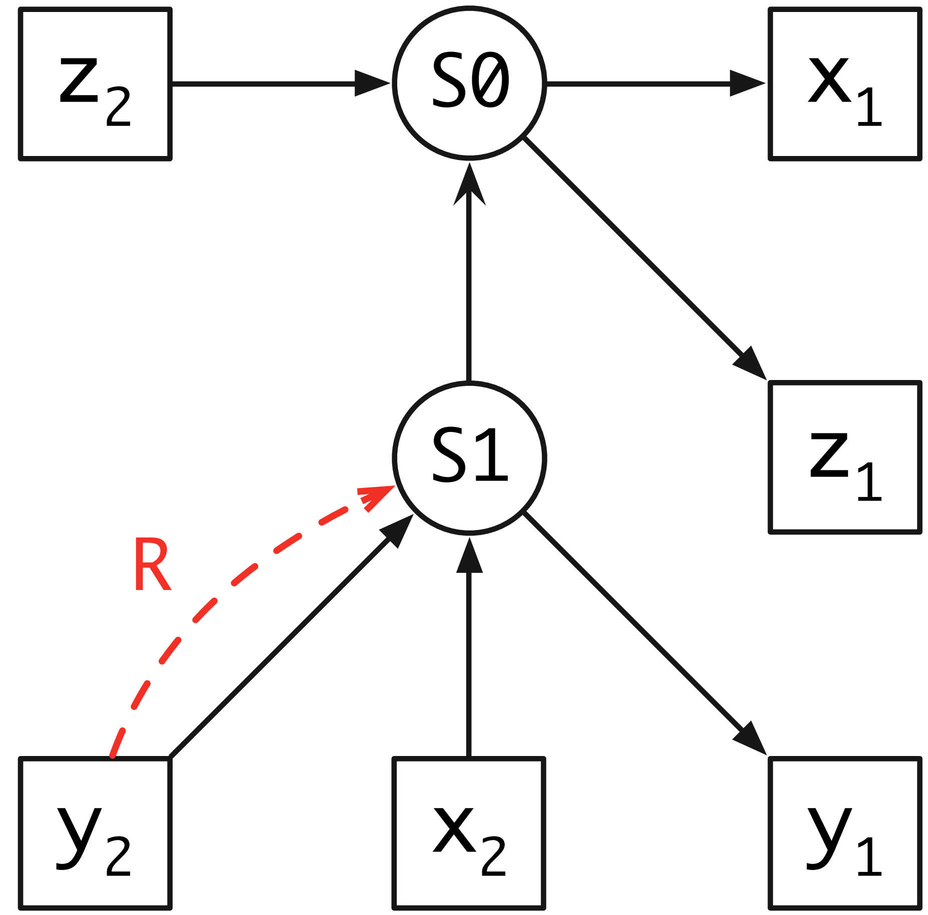


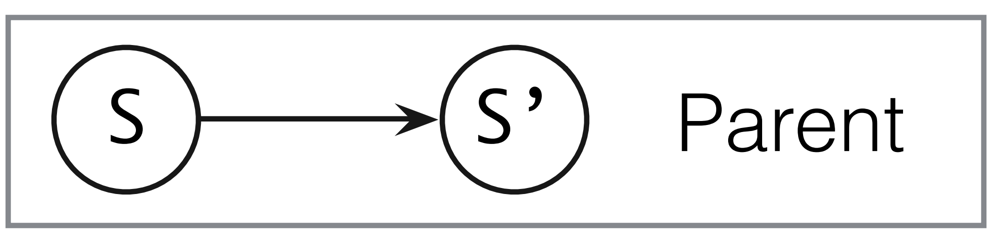




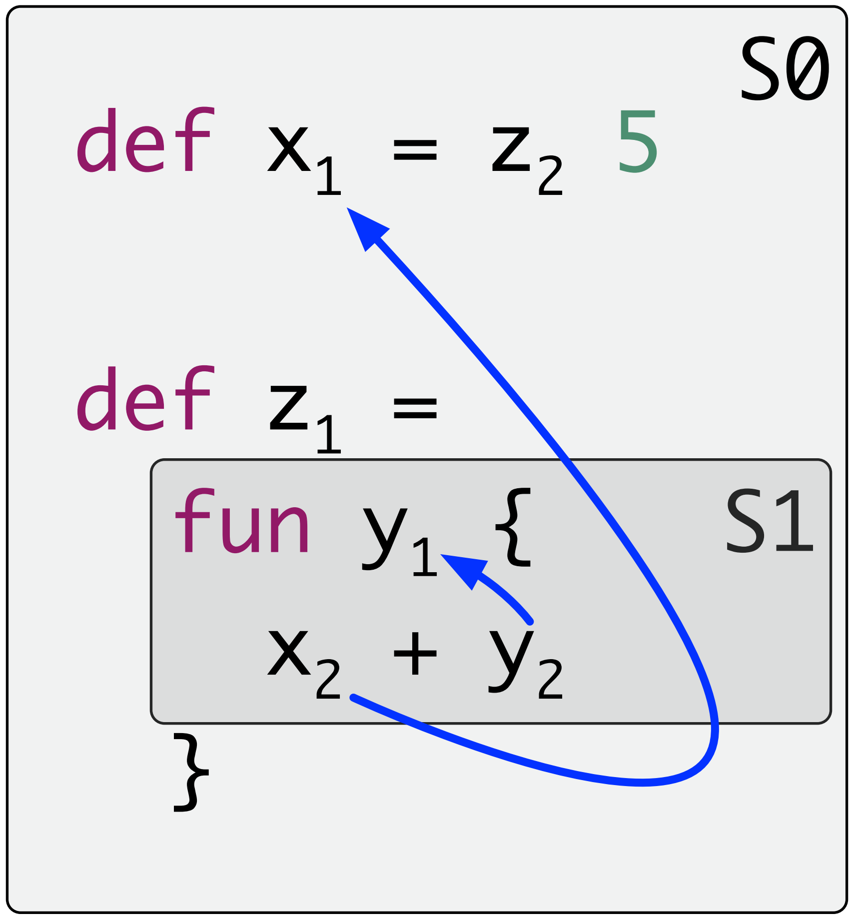
Lexical Scoping

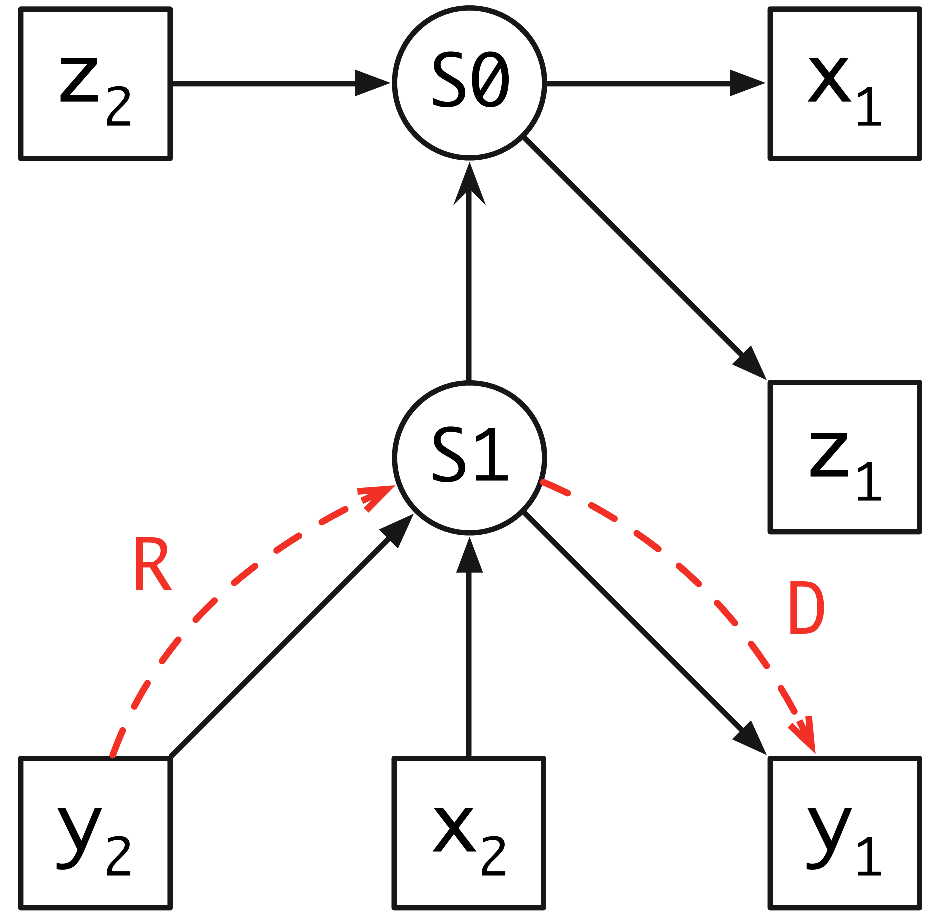


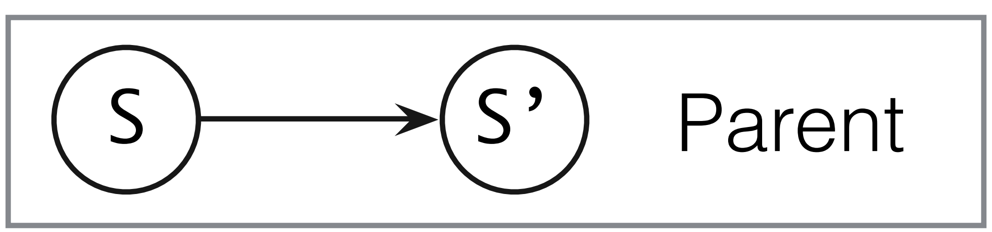




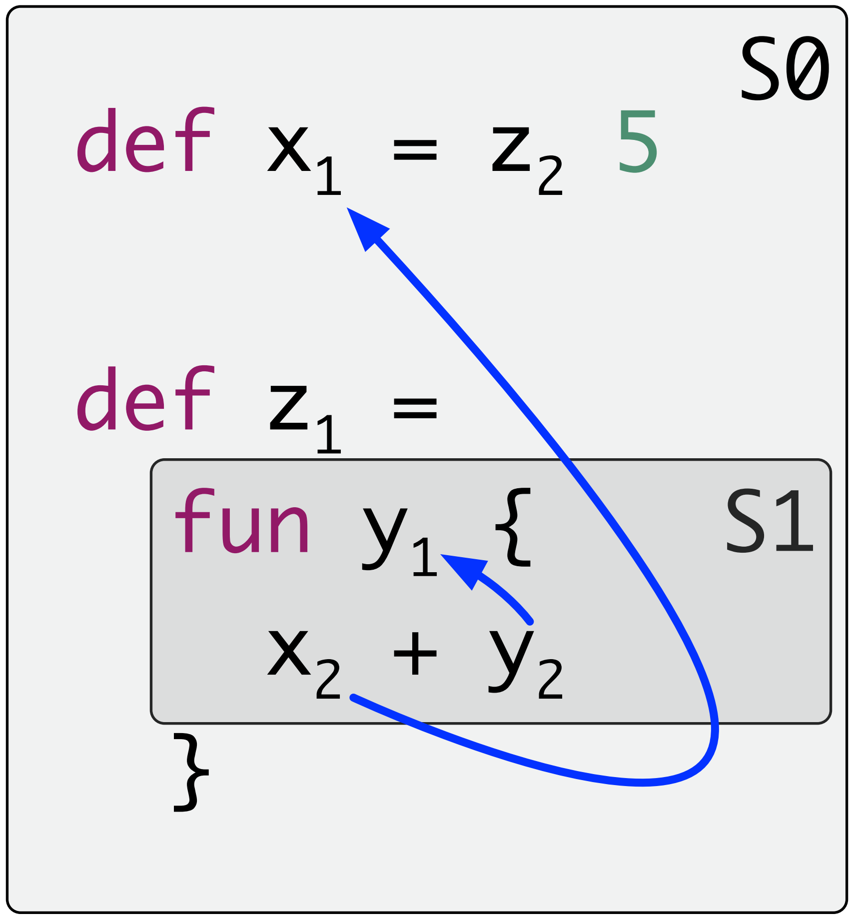
Lexical Scoping

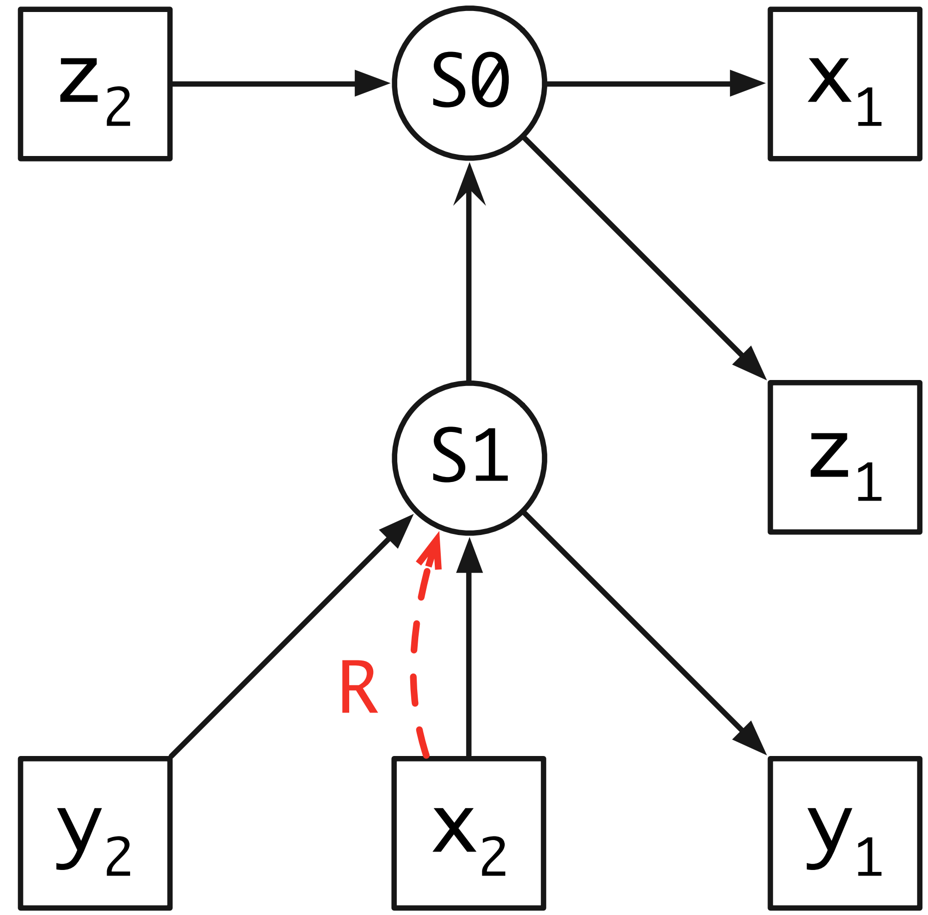


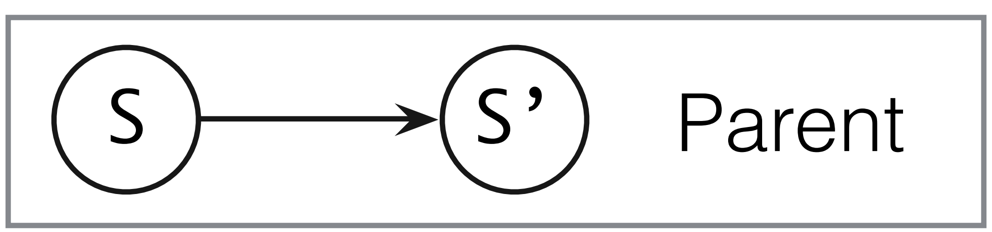




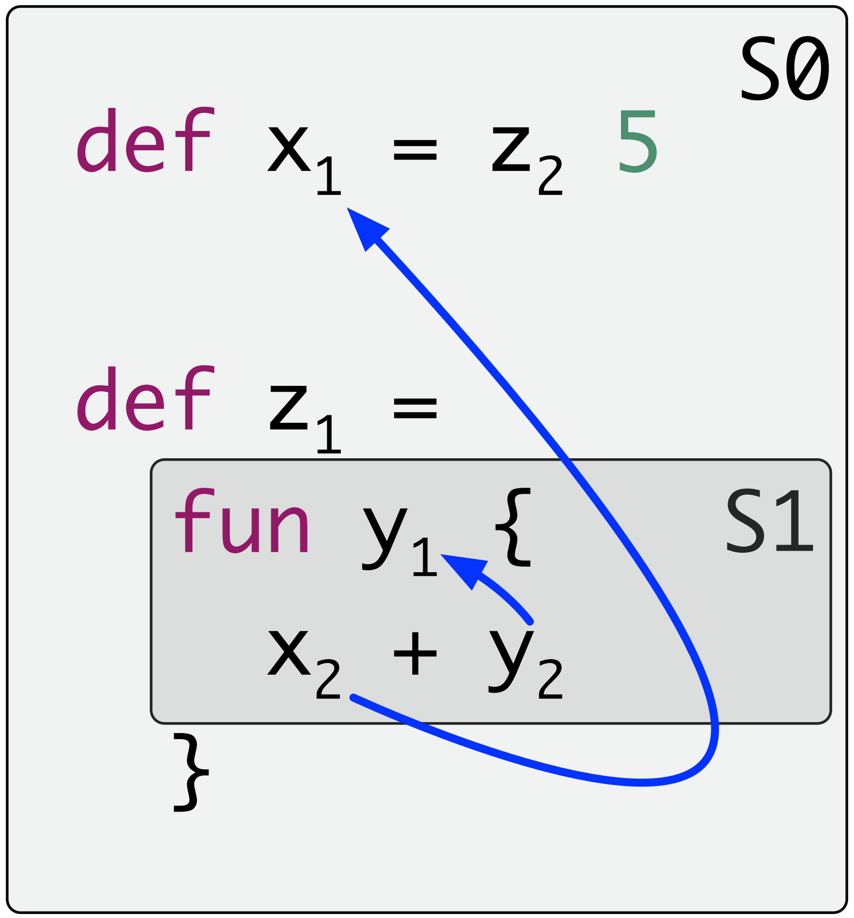
Lexical Scoping

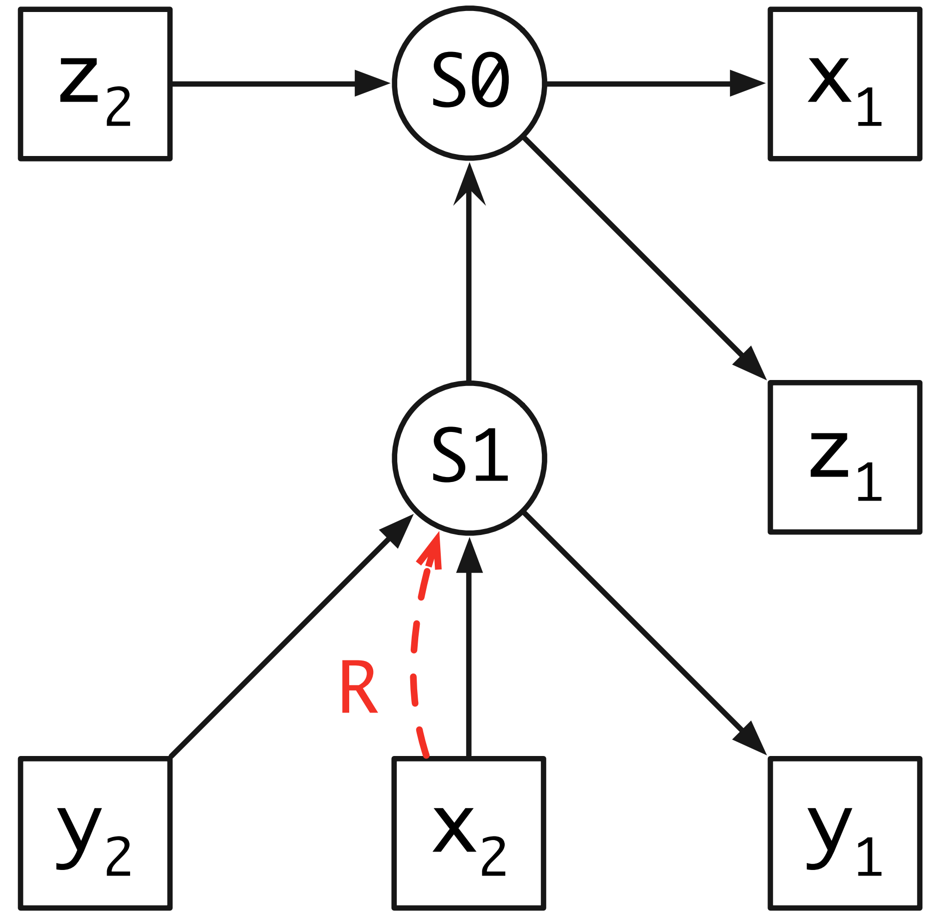


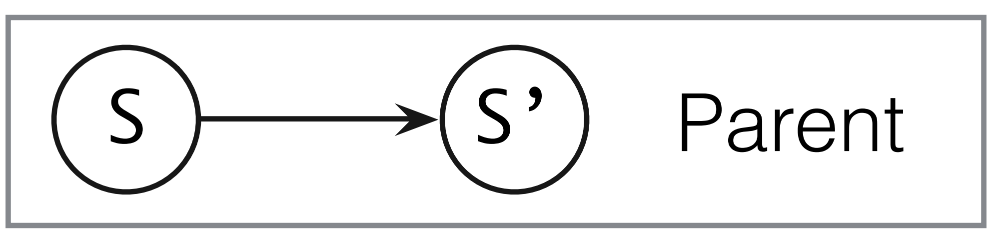


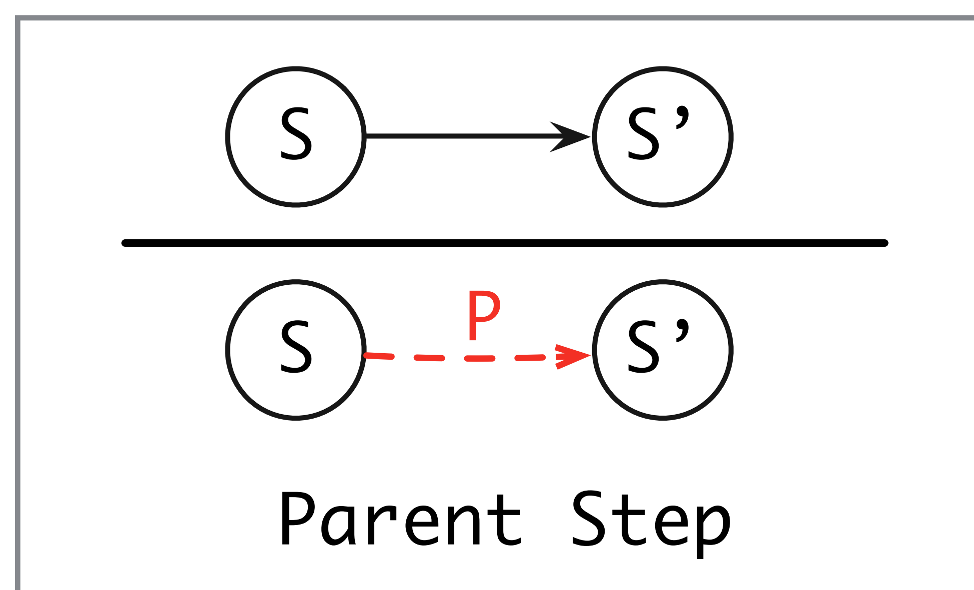


Lexical Scoping

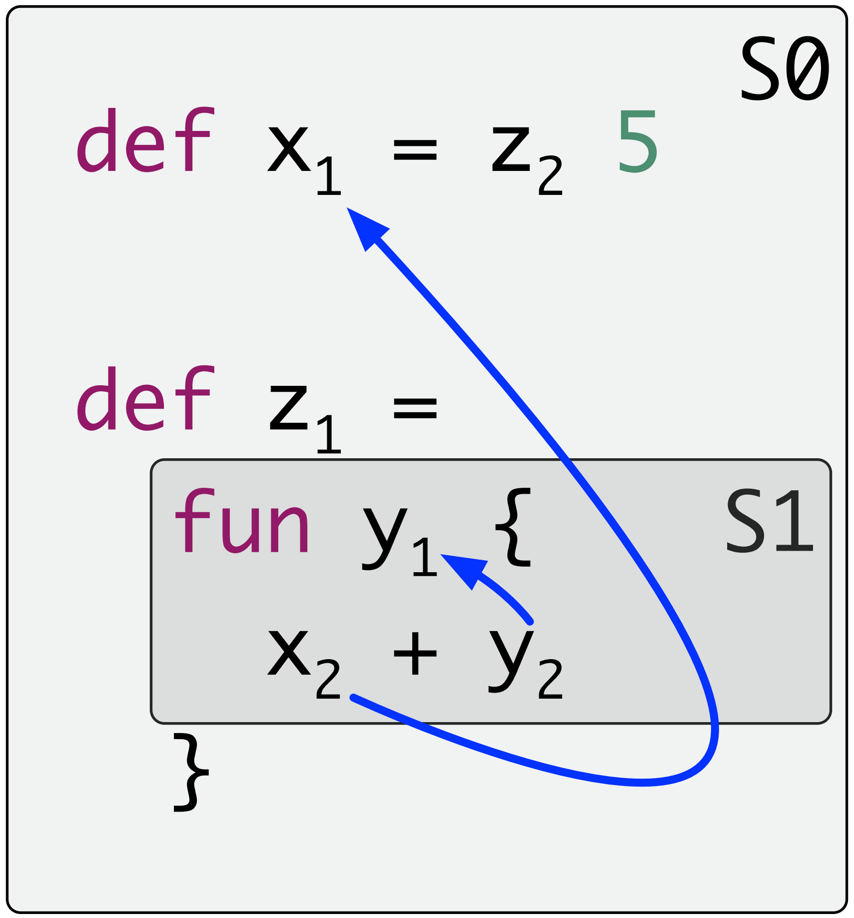


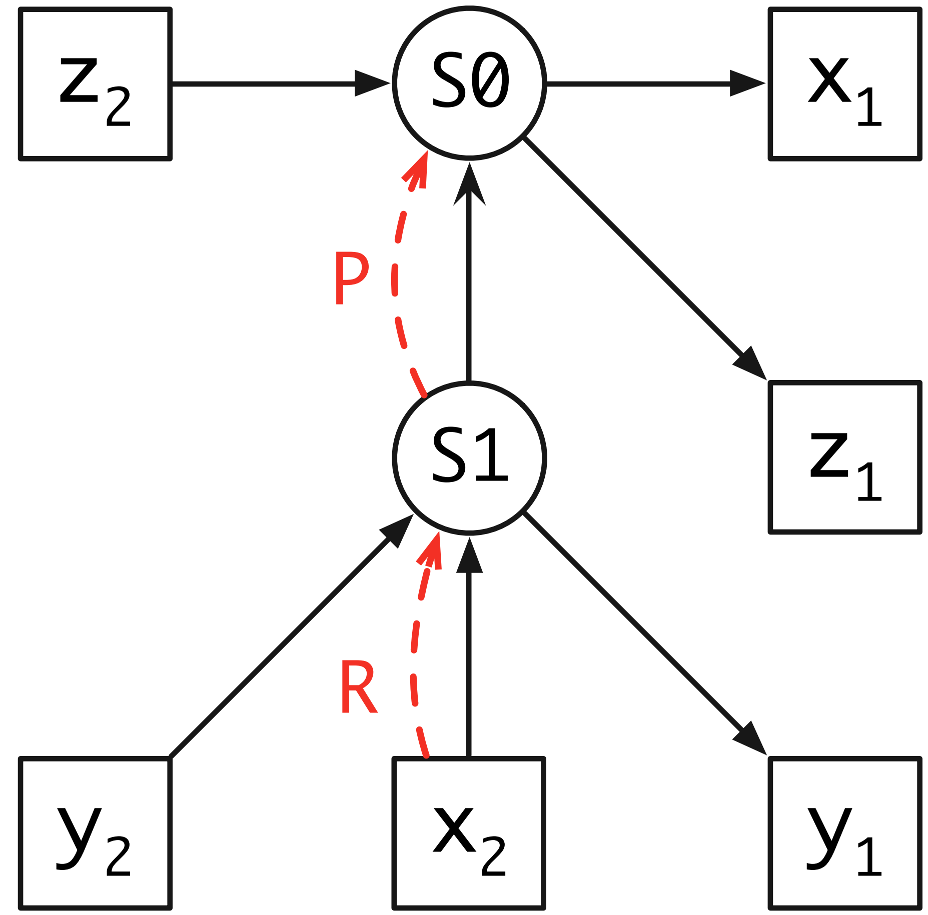


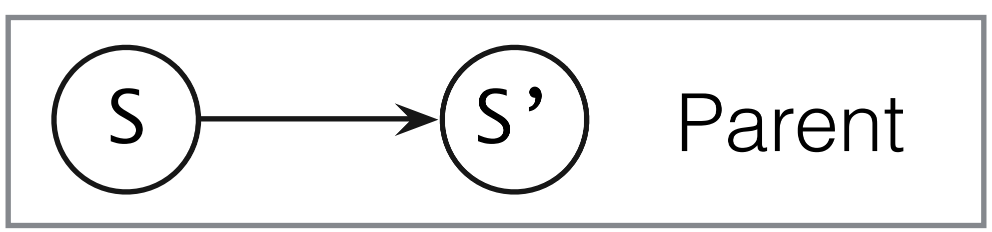


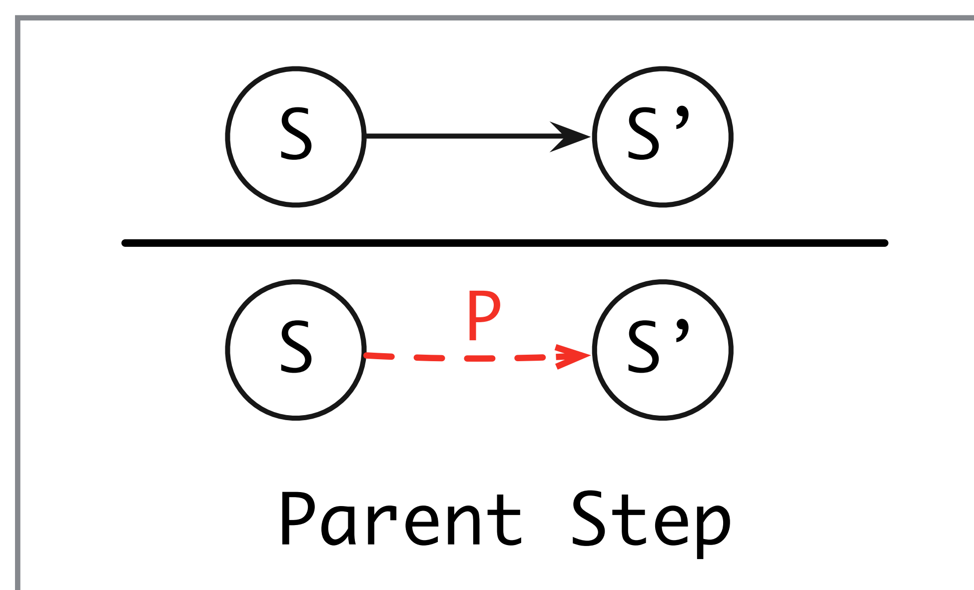


Lexical Scoping

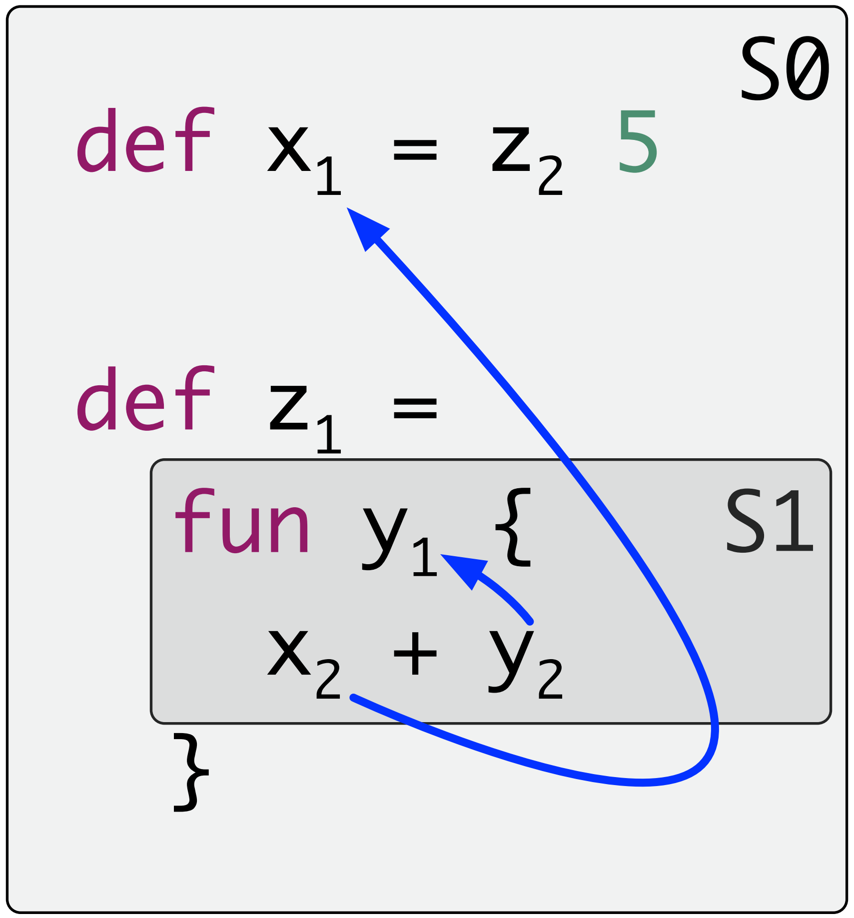


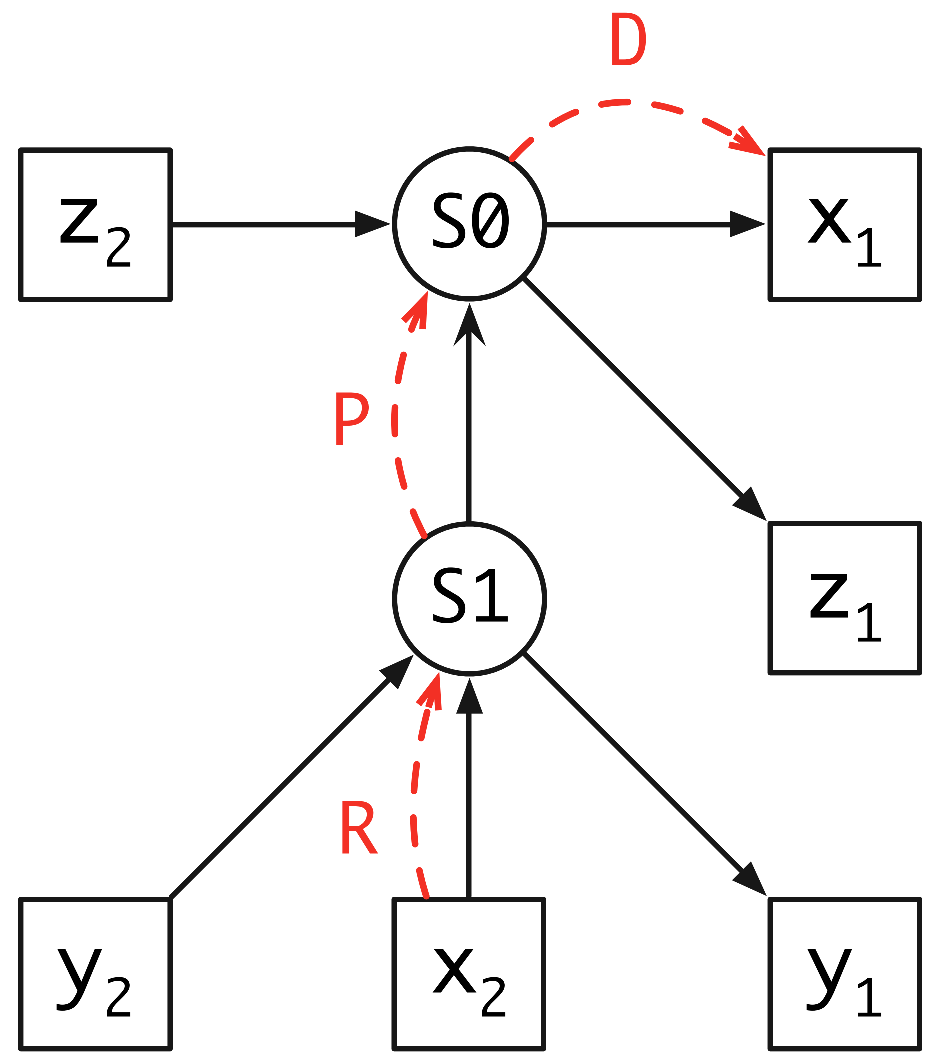


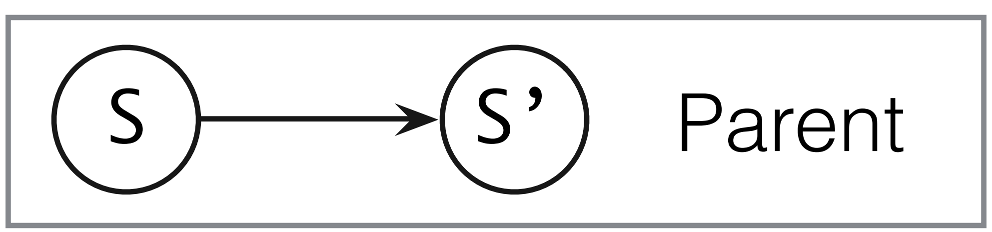


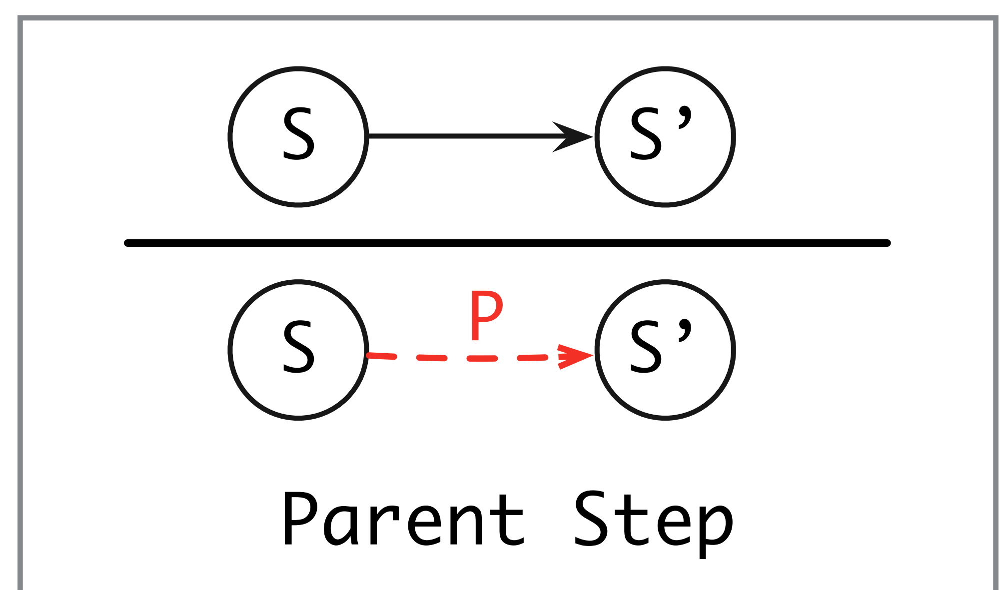


Lexical Scoping

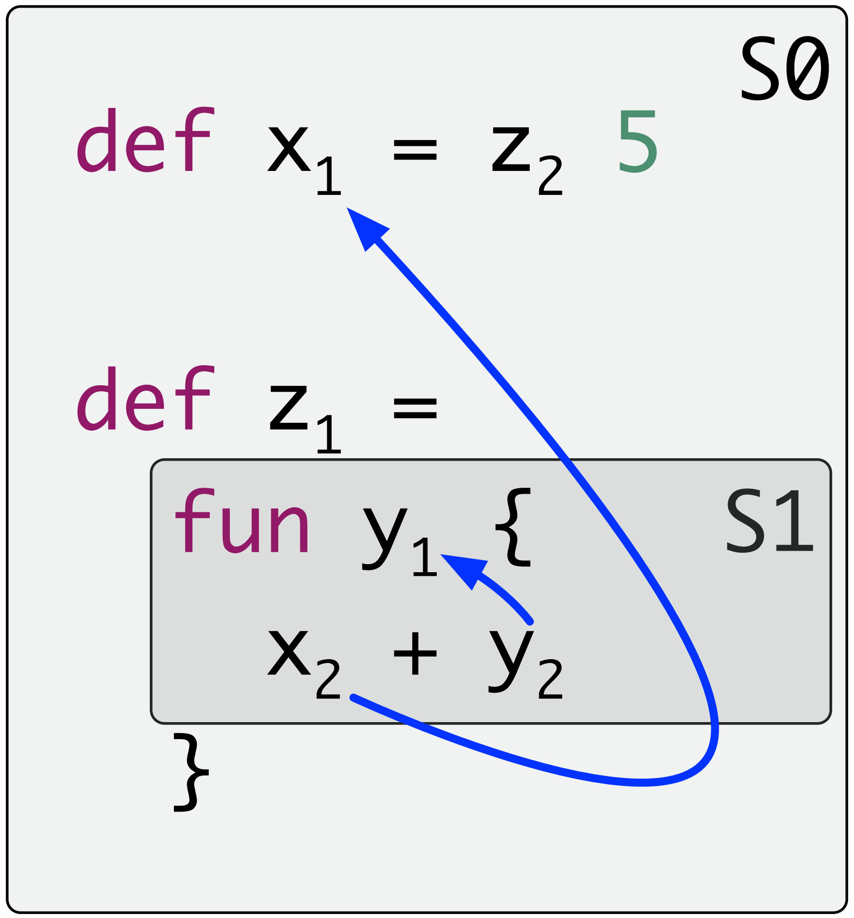


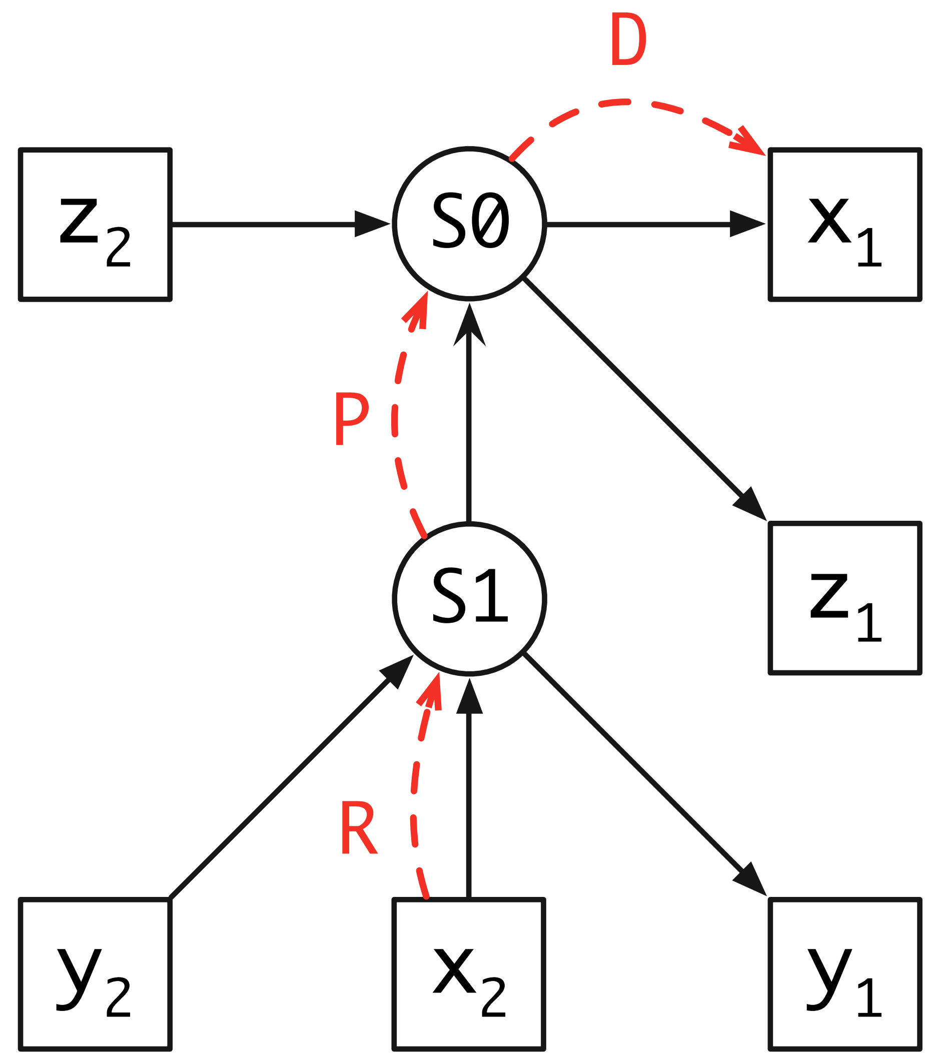


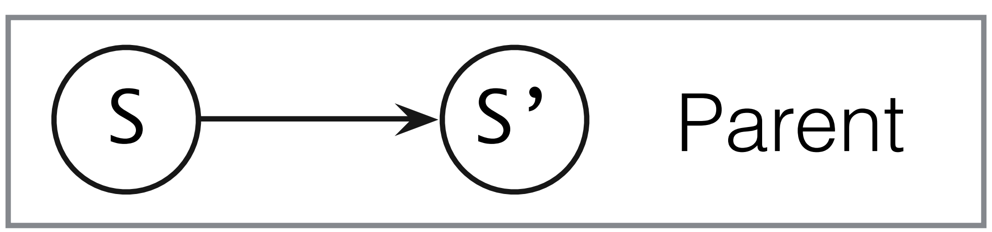


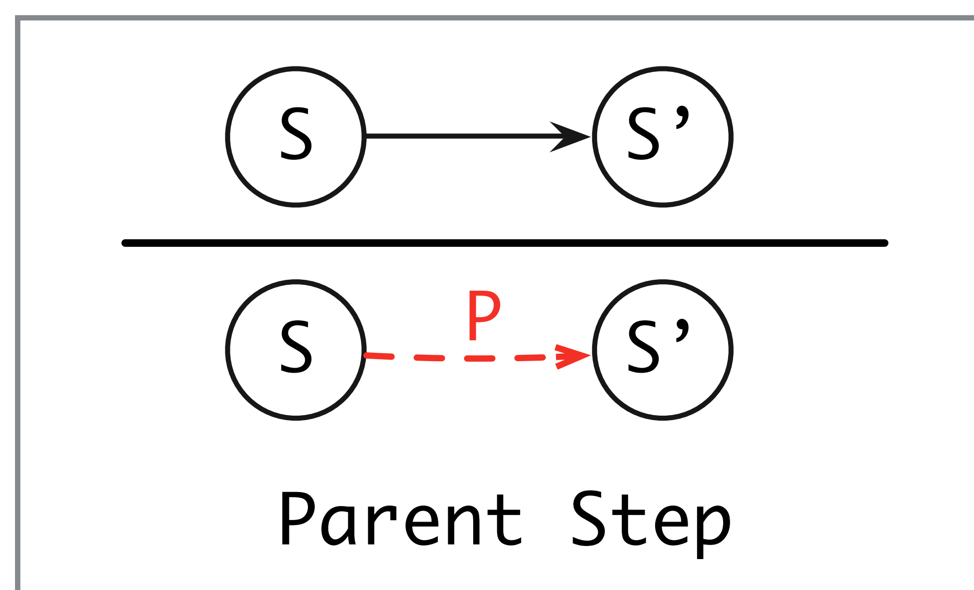


Lexical Scoping









Well formed path: R.P*.D

Shadowing

$\texttt{def } x_3 = z_2\ 5\ 7$

$\texttt{def } z_1 =$

$\quad \texttt{fun } x_1\ \{$

$\quad\quad \texttt{fun } y_1\ \{$

$\quad\quad\quad x_2 + y_2$

$\quad\quad \}$

$\quad \}$

Shadowing

S0

$\texttt{def}\ x_3 = z_2\ 5\ 7$

$\texttt{def}\ z_1 =$

S1

$\quad \texttt{fun}\ x_1\ \{$

S2

$\qquad \texttt{fun}\ y_1\ \{$

$\qquad\quad x_2 + y_2$

$\qquad \}$

$\quad \}$

Shadowing

def x_3 = z_2 5 7 S0

def z_1 =

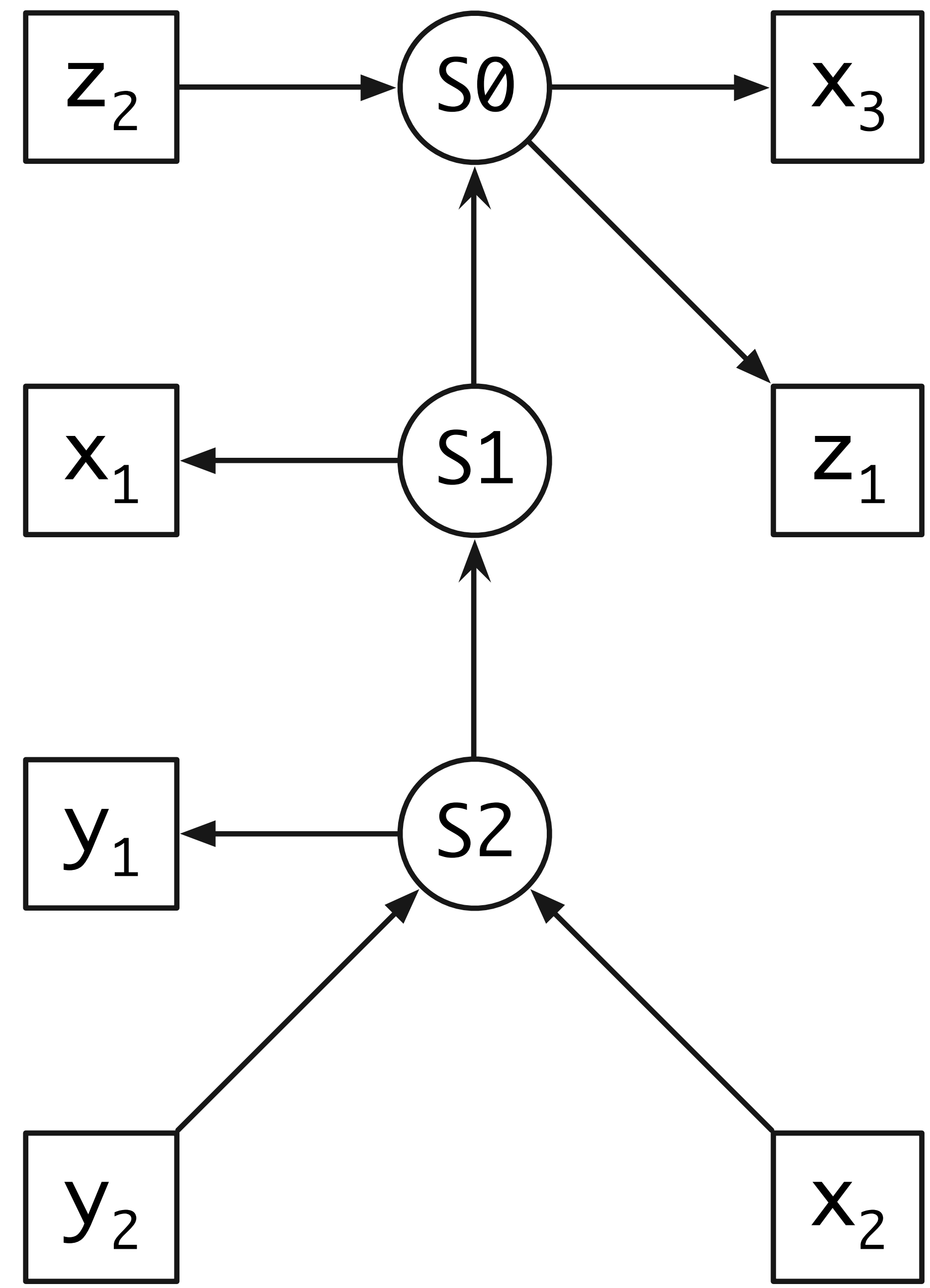
fun x_1 { S1

fun y_1 { S2

x_2 + y_2

}

}



Shadowing

S0

$\texttt{def}\ x_3 = z_2\ 5\ 7$

$\texttt{def}\ z_1 =$

S1

$\quad \texttt{fun}\ x_1\ \{$

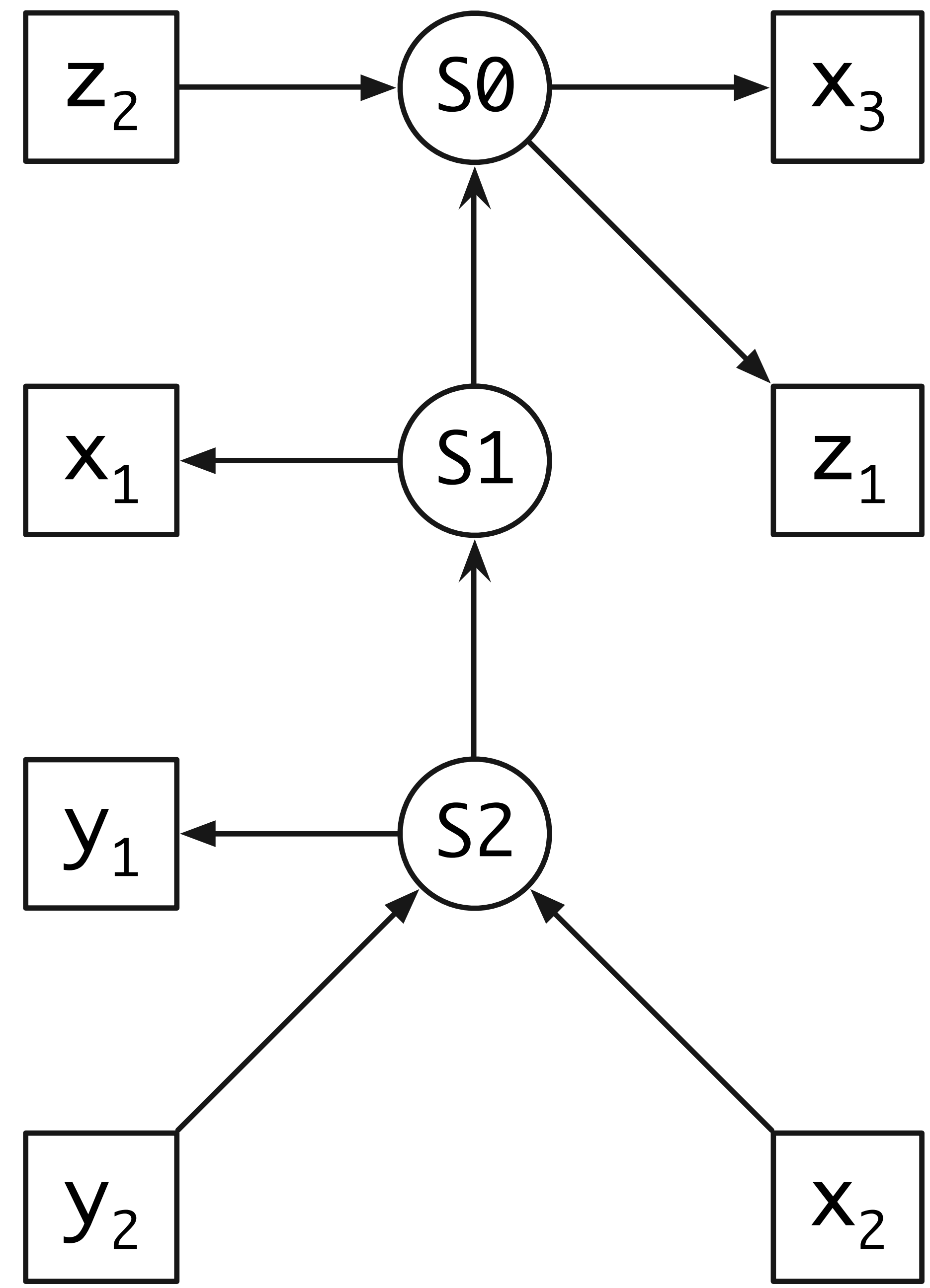
S2

$\quad\quad \texttt{fun}\ y_1\ \{$

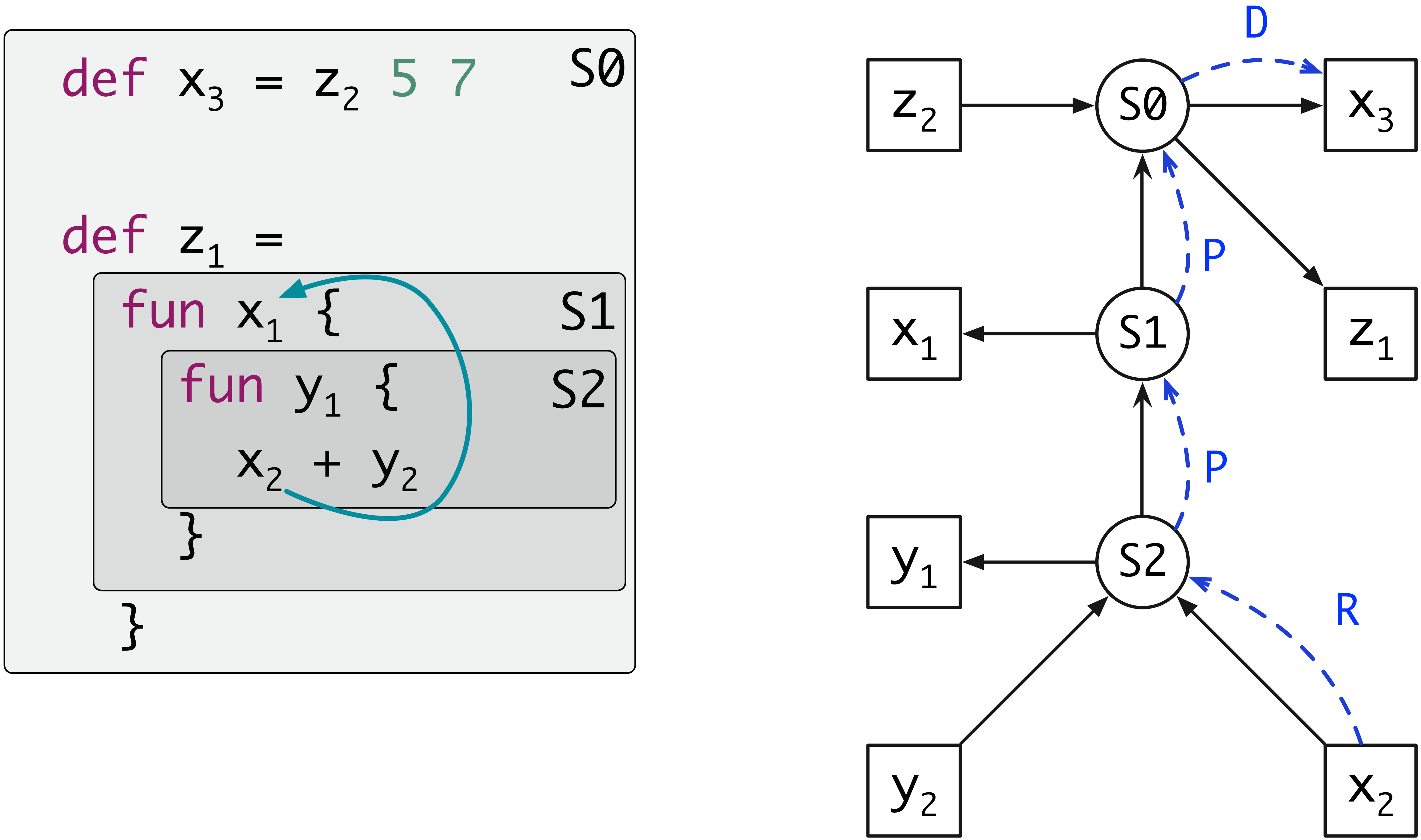
$\quad\quad\quad x_2 + y_2$

$\quad\quad \}$

$\quad \}$

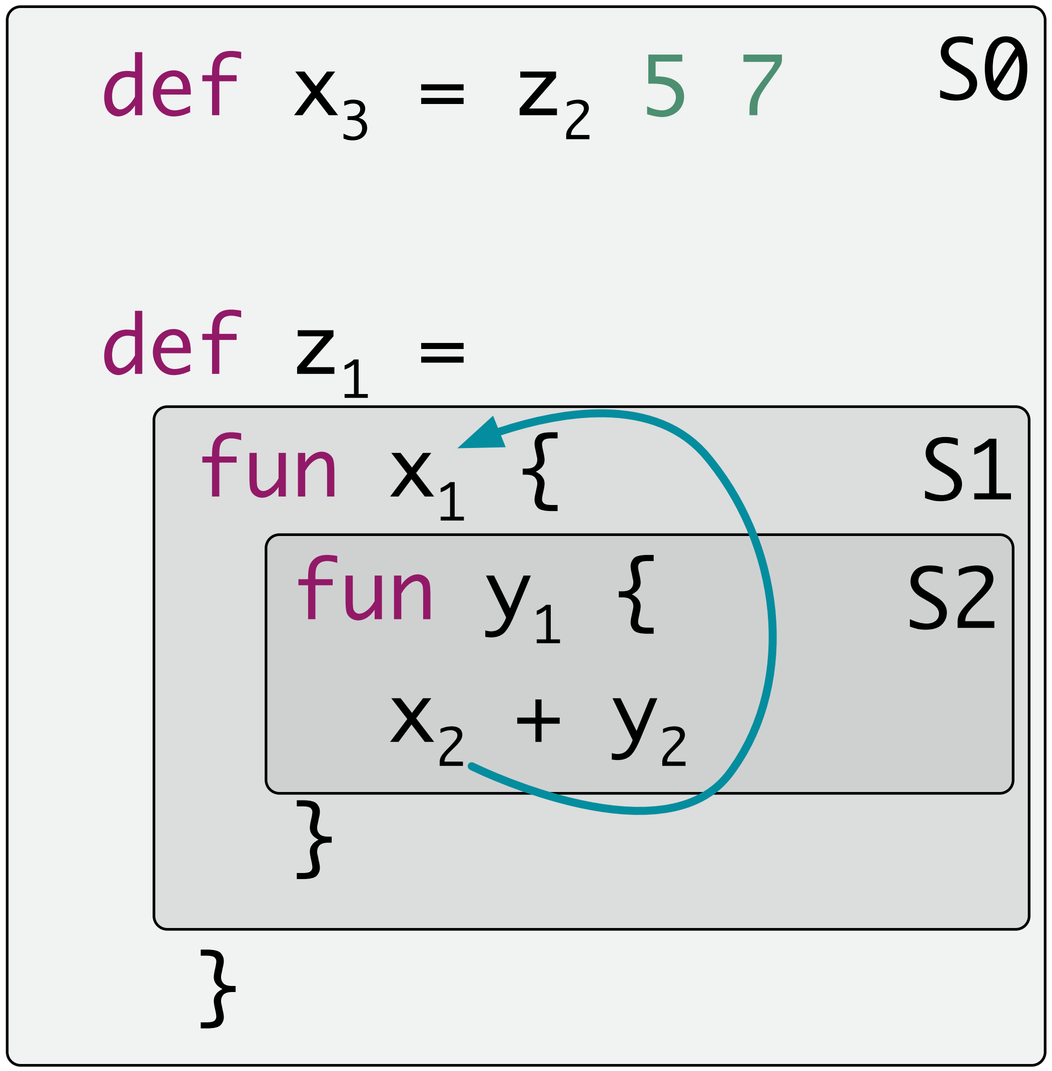


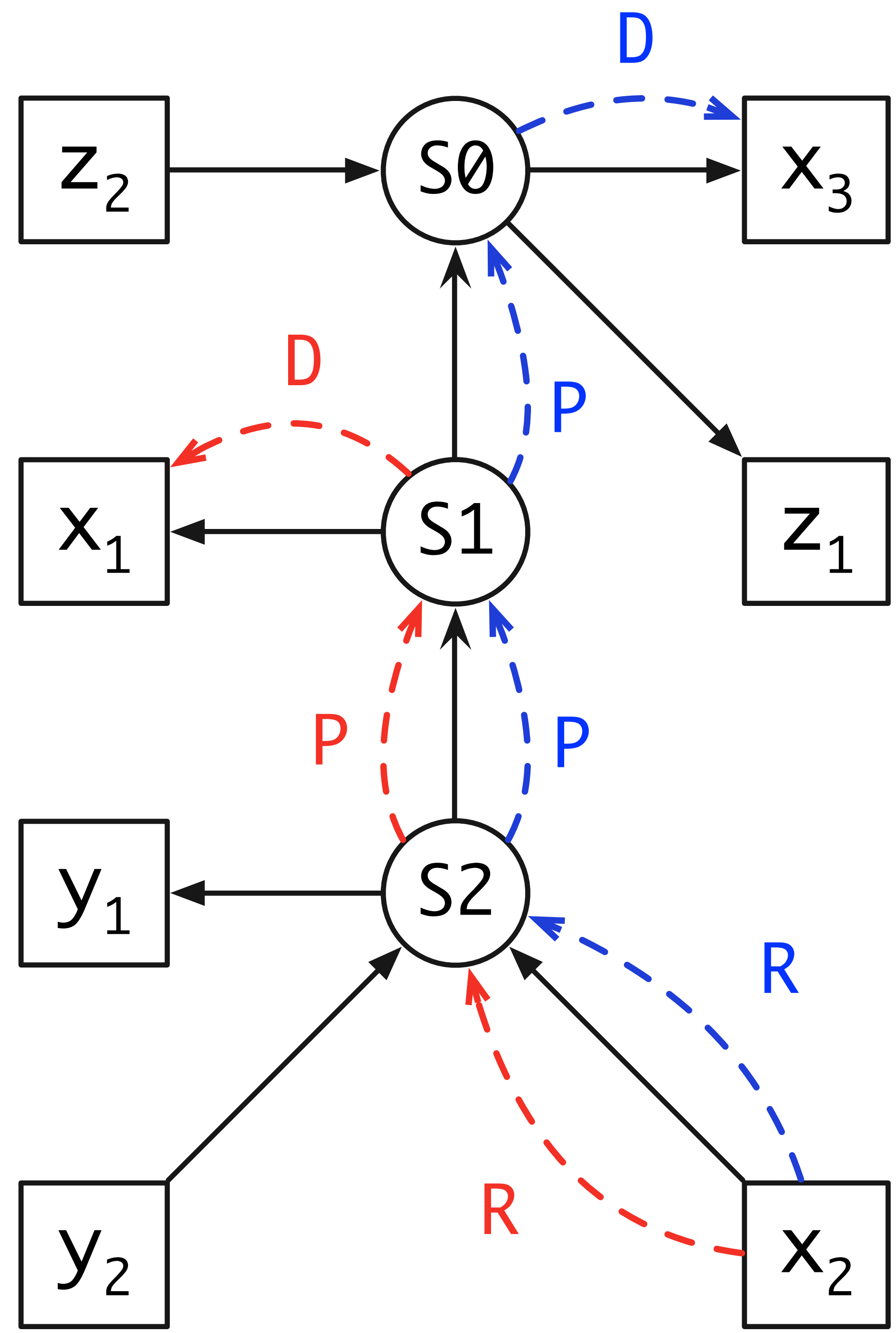
Shadowing



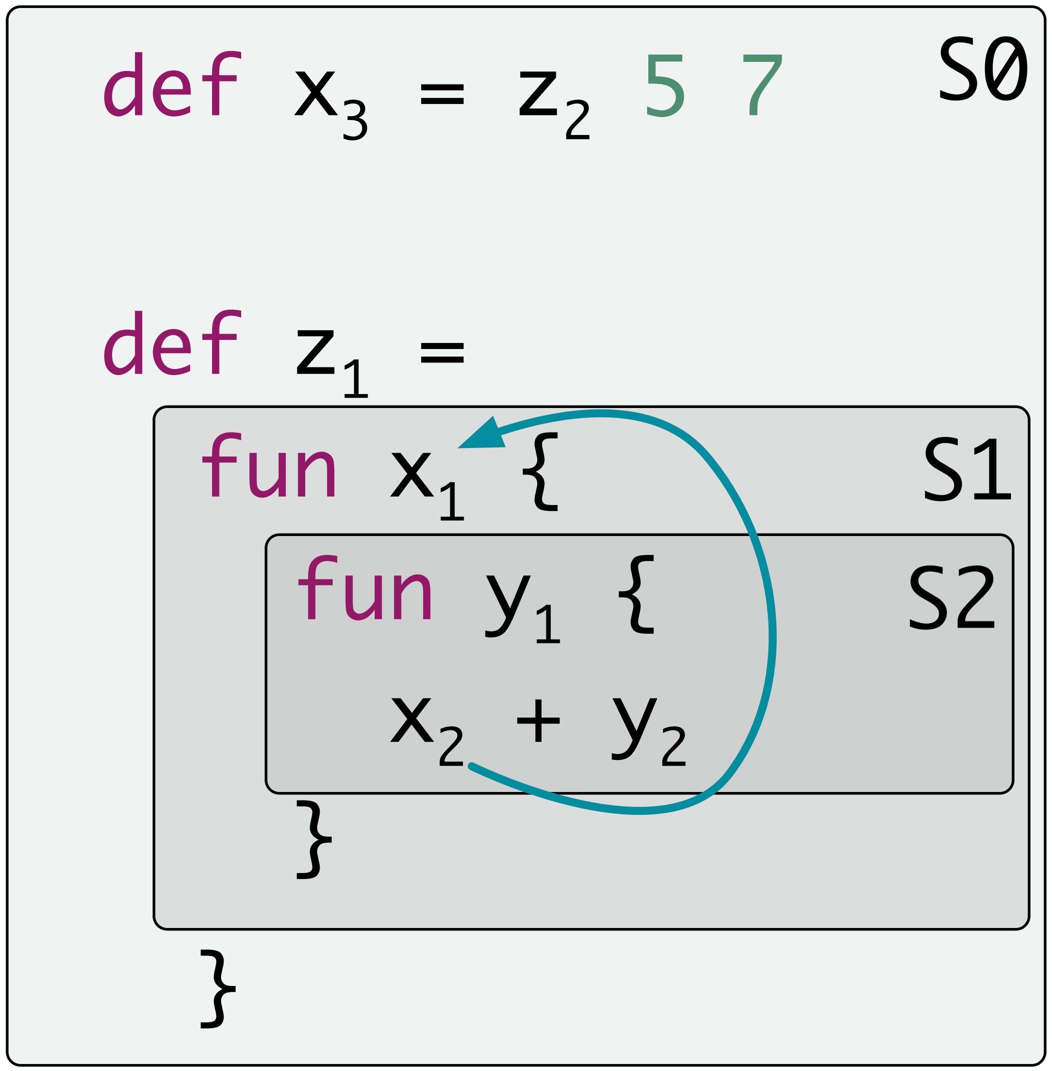
def x3 = z2 5 7
S0
def z1 =
fun x1 {
S1
fun y1 {
S2
x2 + y2
}
}
D
P
P
R
z2
S0
x3
x1
S1
z1
y1
S2
y2
x2

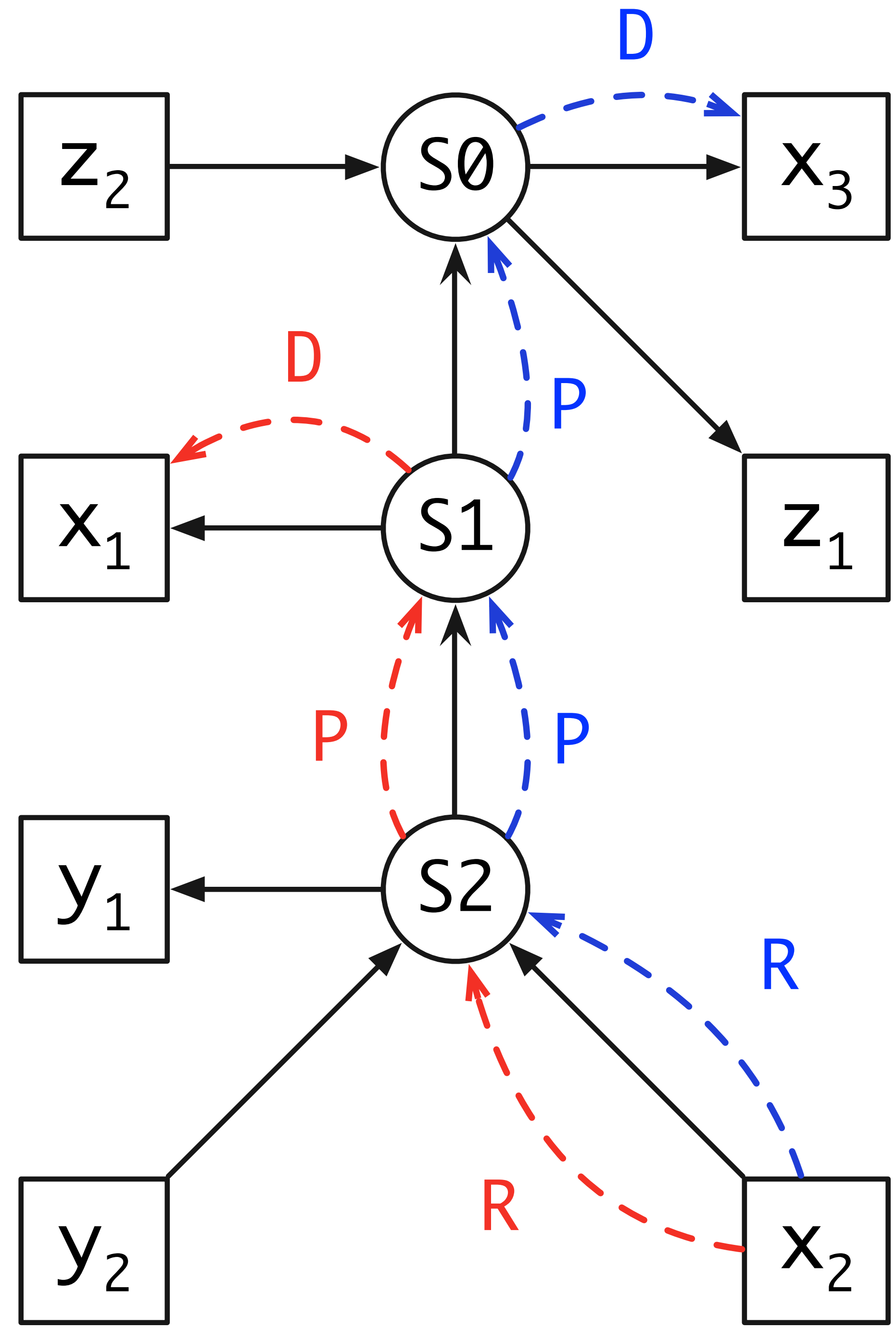
Shadowing


def x_3 = z_2 5 7
S0
def z_1 =
fun x_1 {
S1
fun y_1 {
S2
x_2 + y_2
}
}


z_2
S0
x_3
D
D
P
x_1
S1
z_1
P
P
y_1
S2
R
R
y_2
x_2

Shadowing

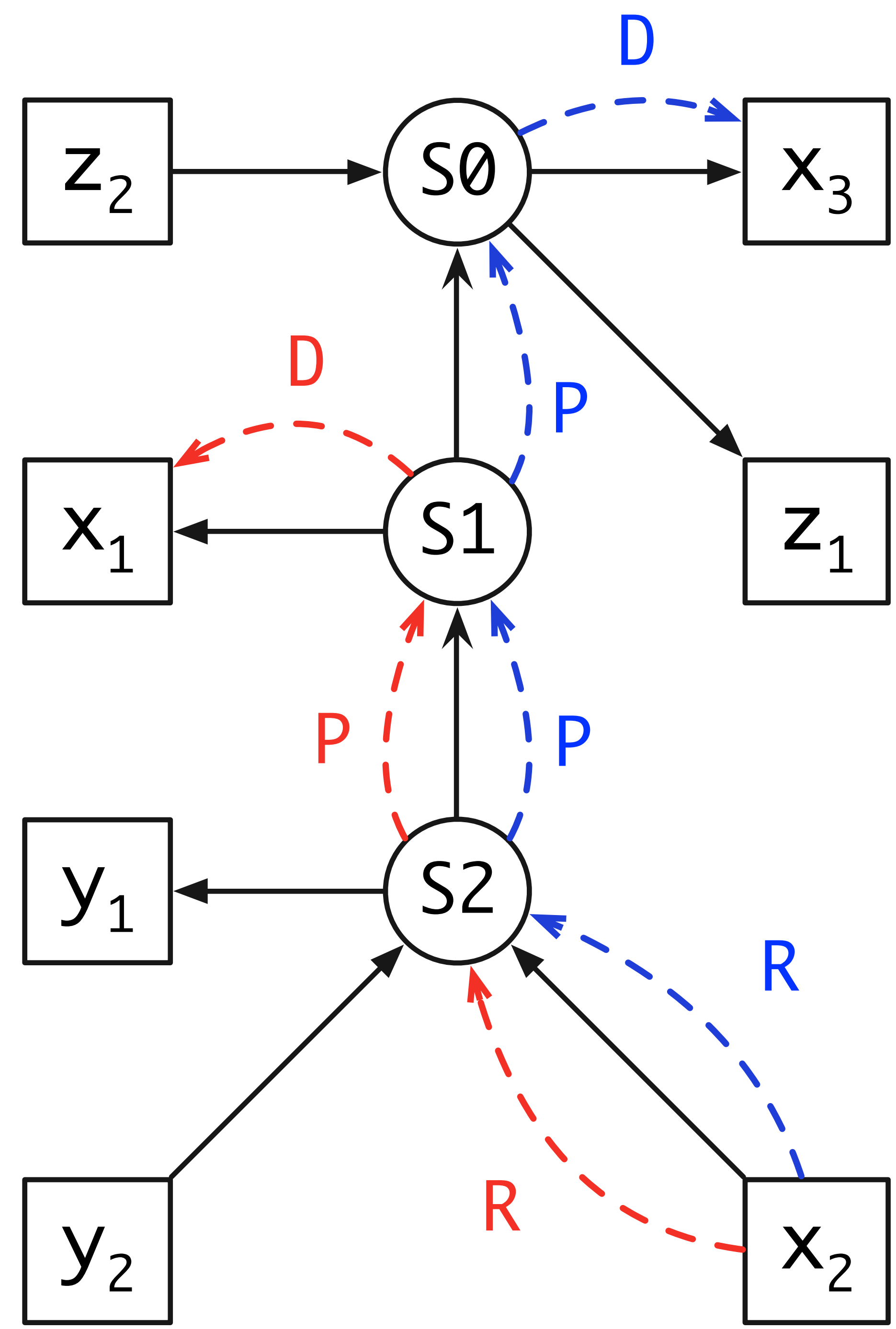




D < P.p

Shadowing

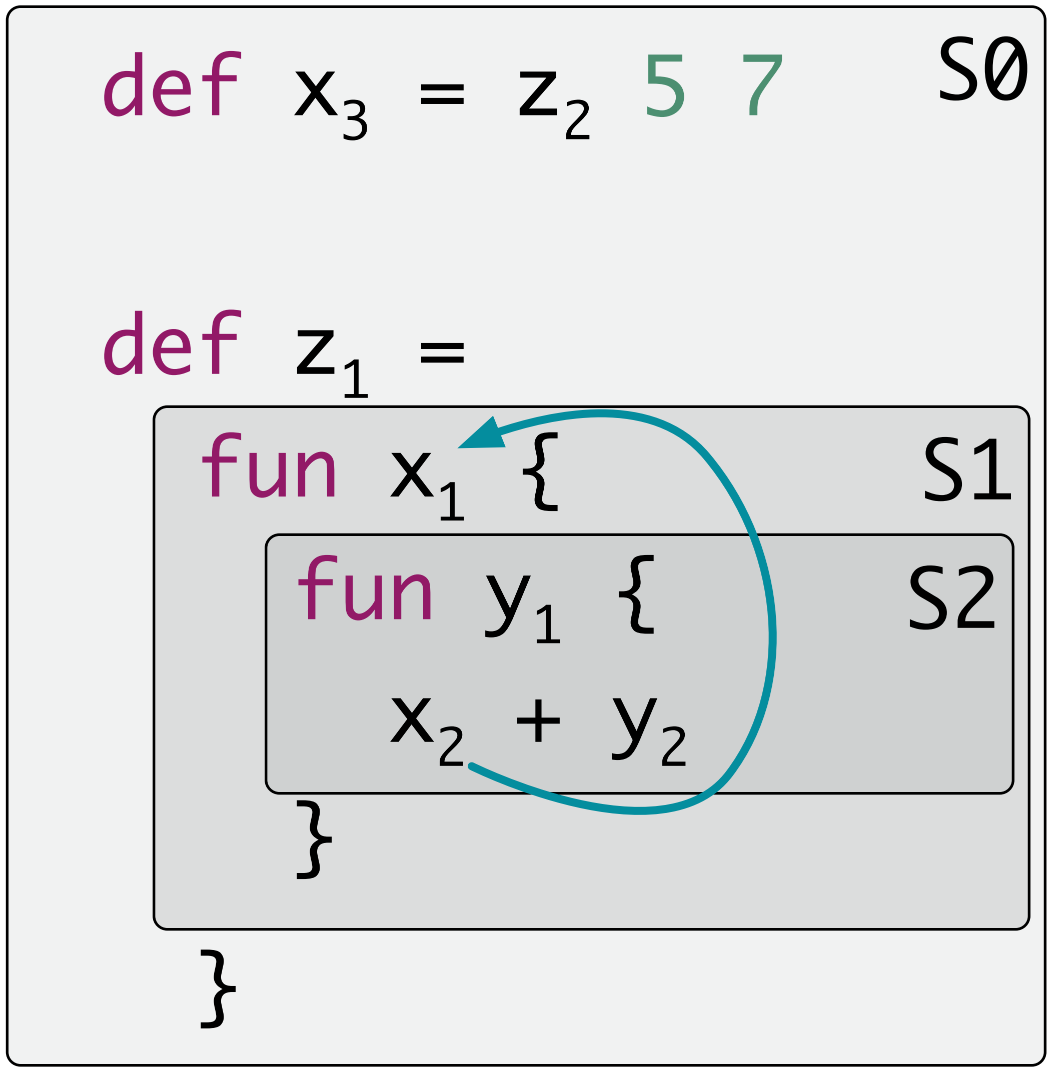
```
def x_3 = z_2 5 7                S0

def z_1 =
  fun x_1 {                      S1
    fun y_1 {                    S2
      x_2 + y_2
    }
  }
```



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

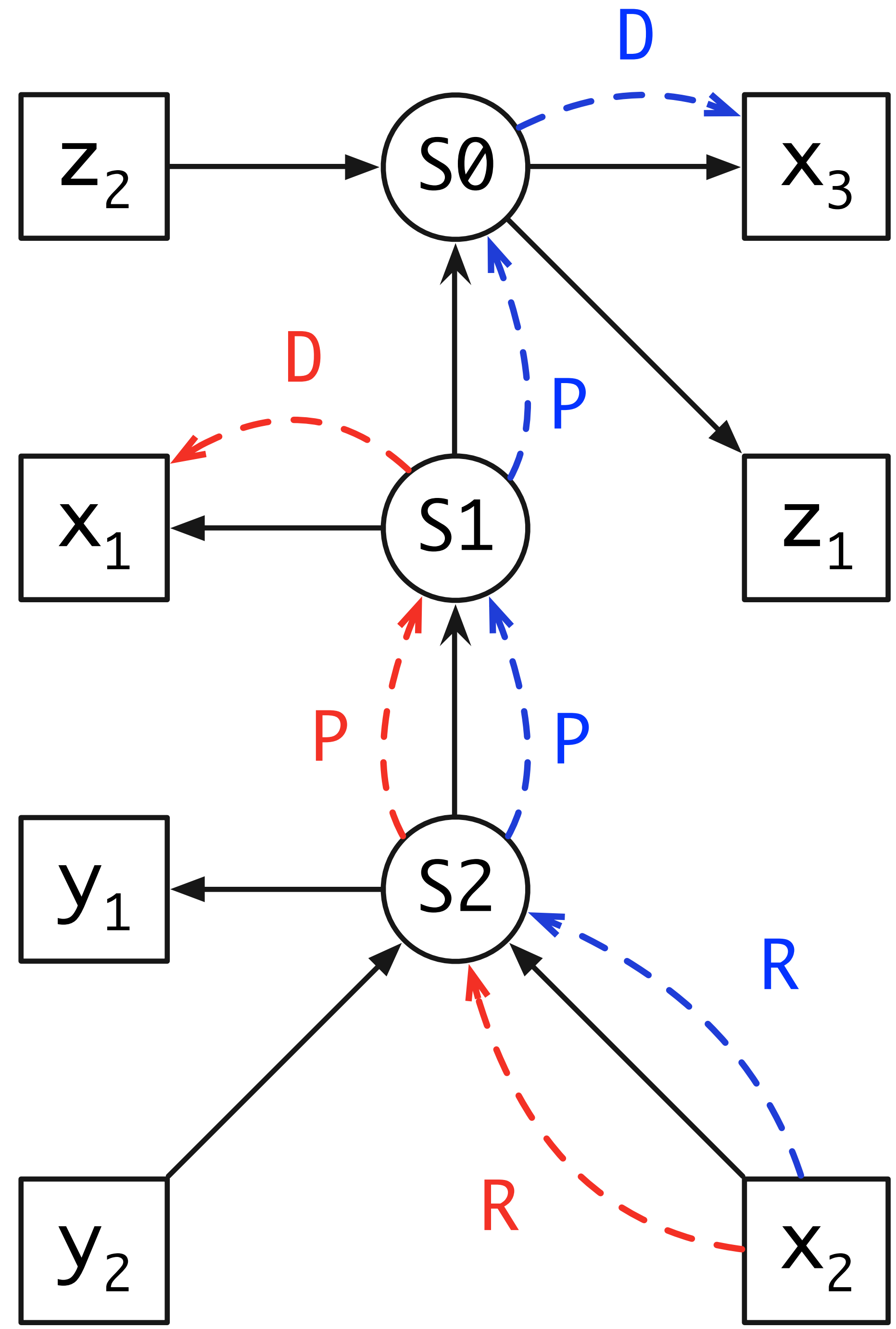
$$\frac{p < p'}{s.p < s.p'}$$

Shadowing



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

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



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

Imports

S0

module A_1 {

  def z_1 = 5   SA

}

module B_1 {

  import A_2 SB

  def x_1 = 1 + z_2

}

Imports

S0

module A_1 {

  def z_1 = 5   SA

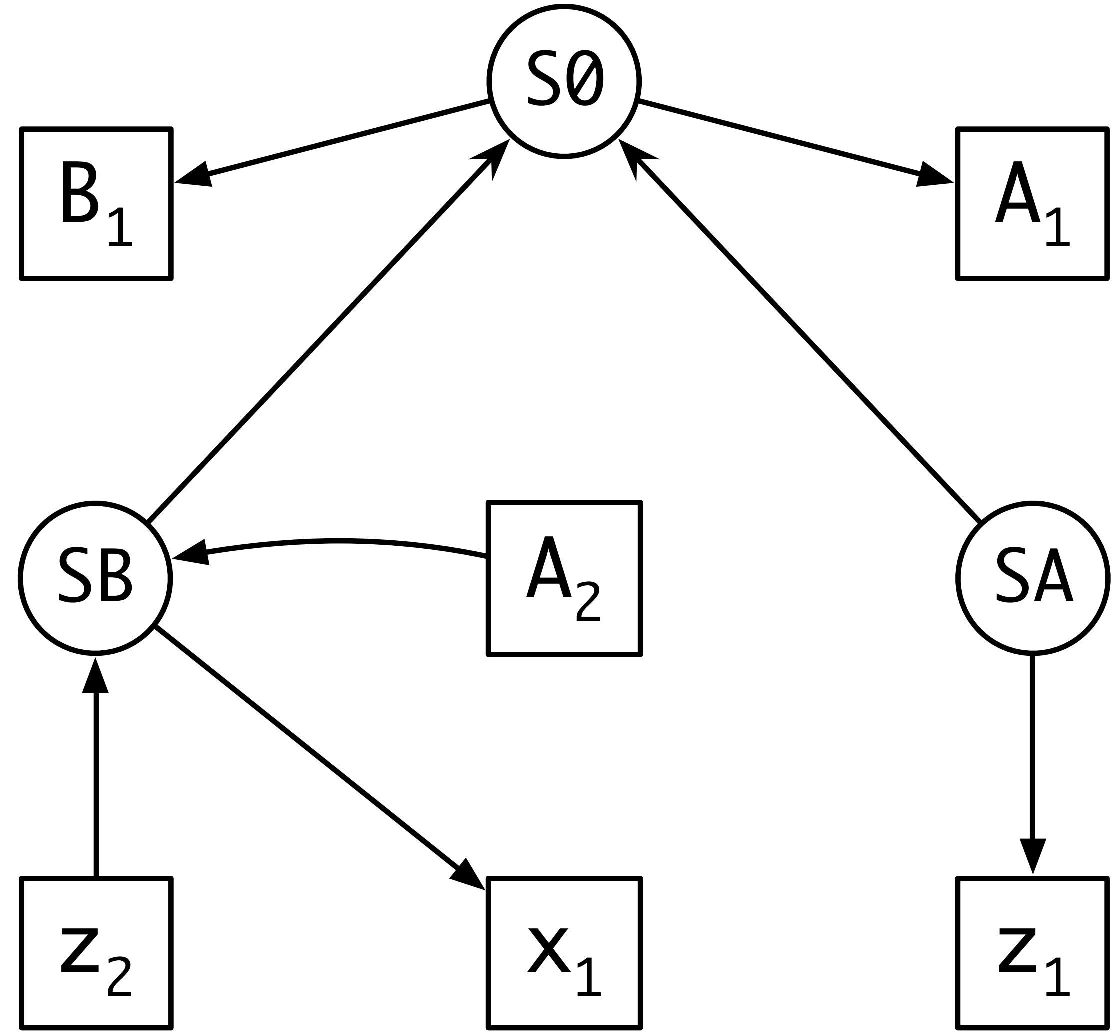
}

module B_1 {

  import A_2 SB

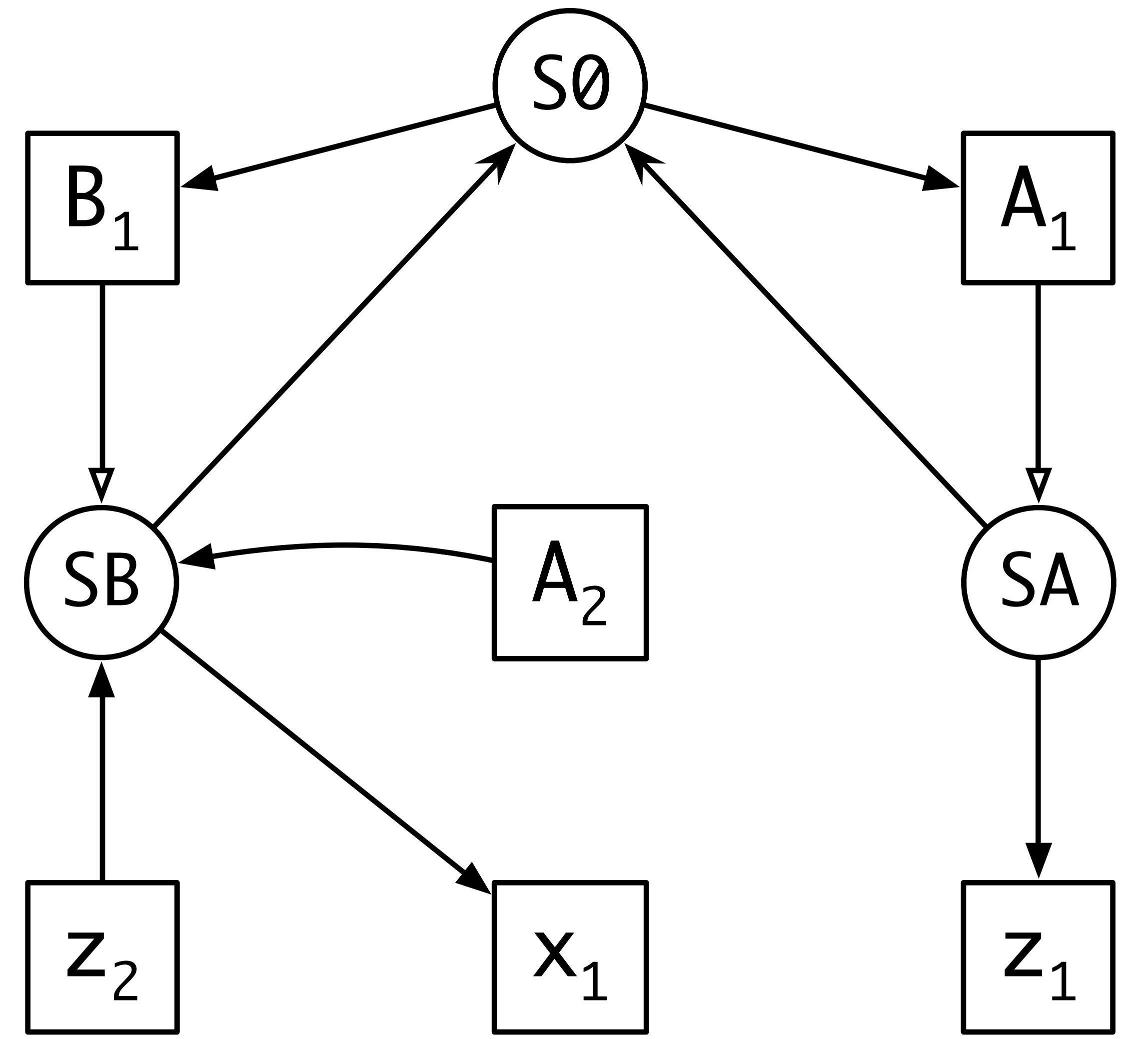
  def x_1 = 1 + z_2

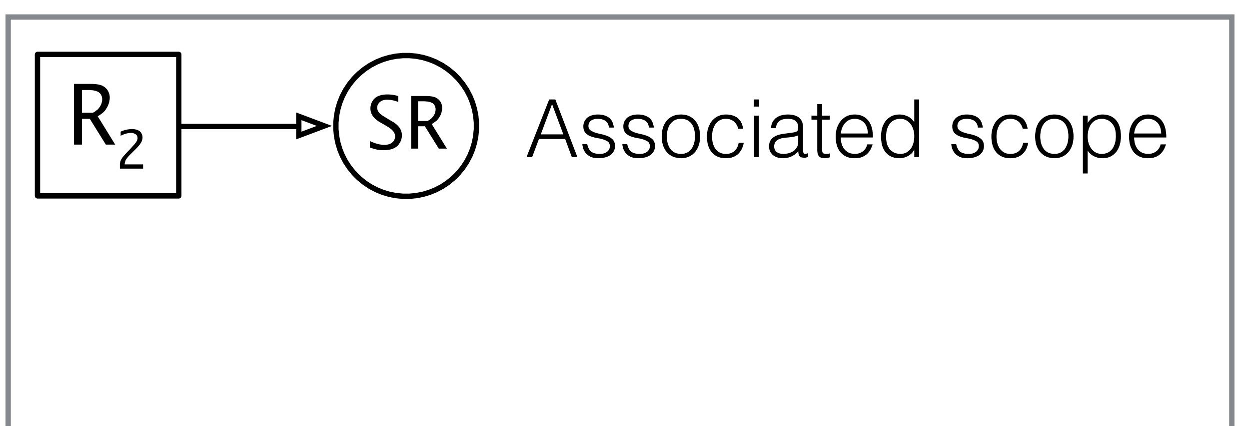
}



Imports

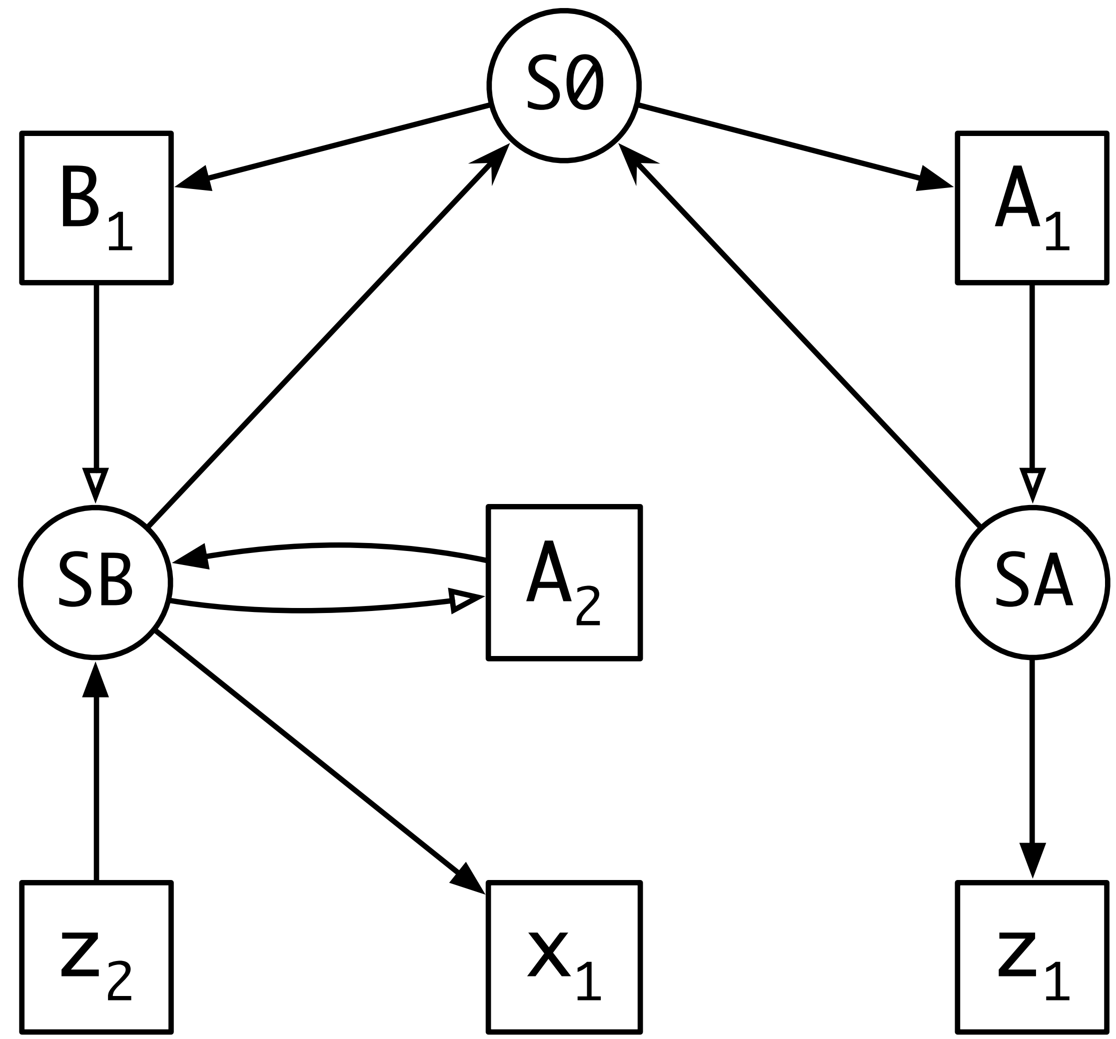
```
module A1 {                S0
  def z1 = 5               SA
}
module B1 {
  import A2                SB
  def x1 = 1 + z2
}
```

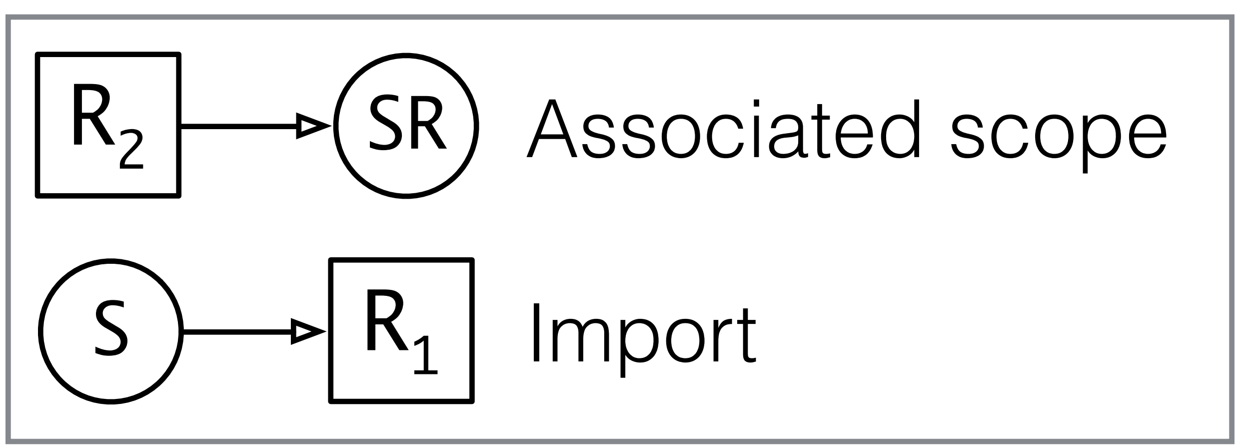




Imports

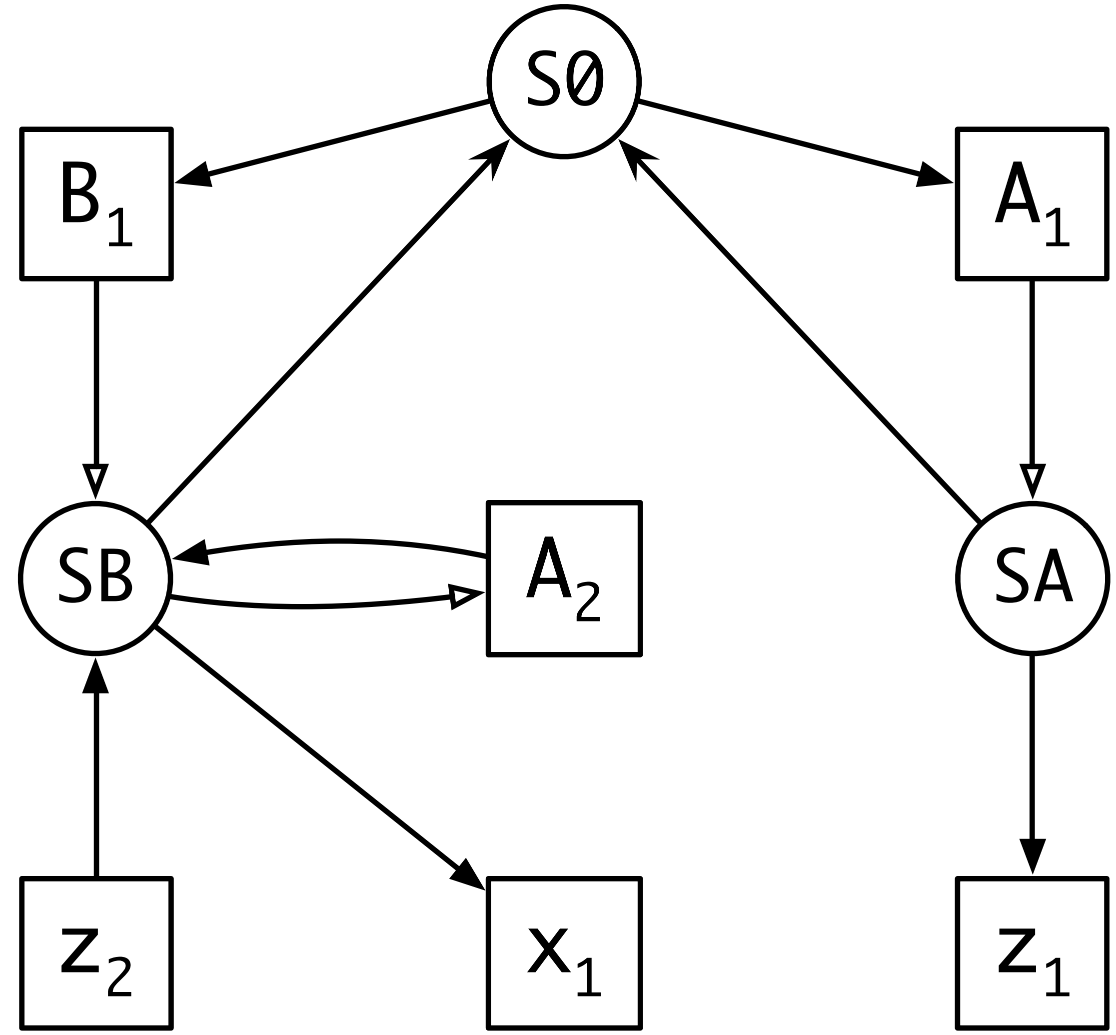
```
module A₁ {                S0
  def z₁ = 5           SA
}
module B₁ {
  import A₂            SB
  def x₁ = 1 + z₂
}
```

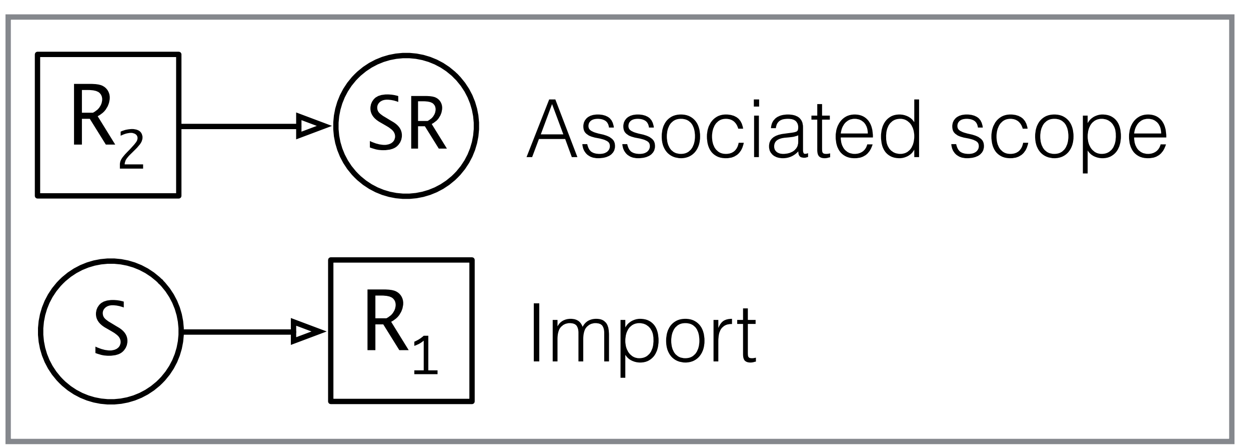




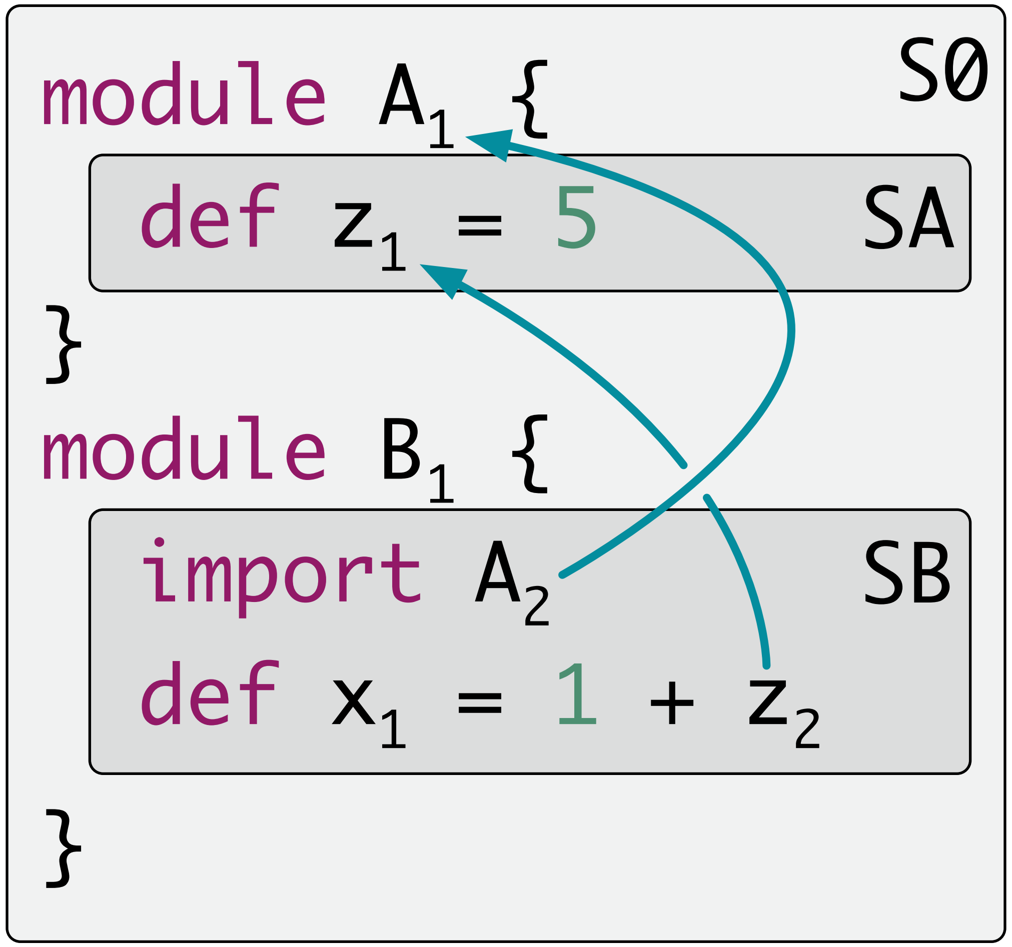
Imports

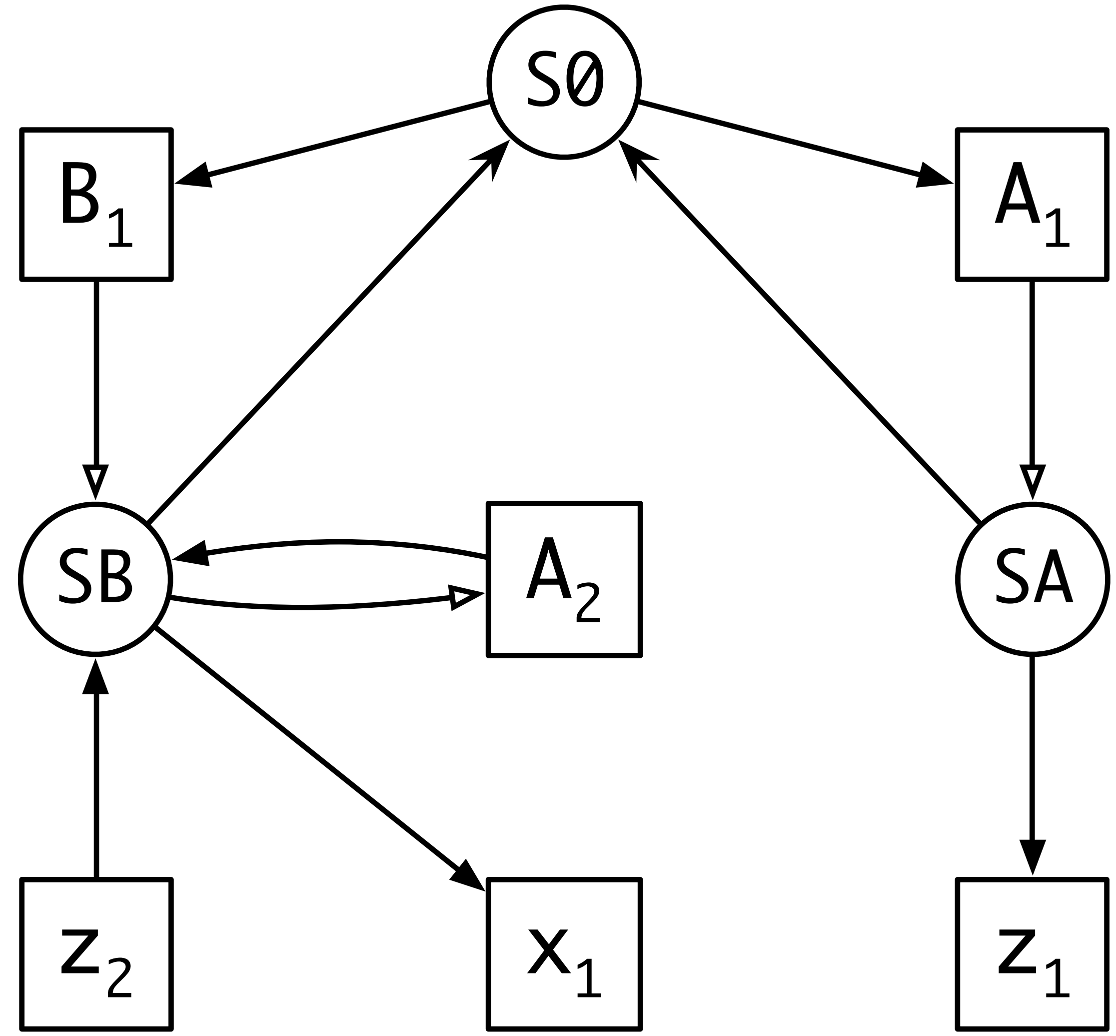
```
module A₁ {                S0
  def z₁ = 5               SA
}
module B₁ {
  import A₂                SB
  def x₁ = 1 + z₂
}
```

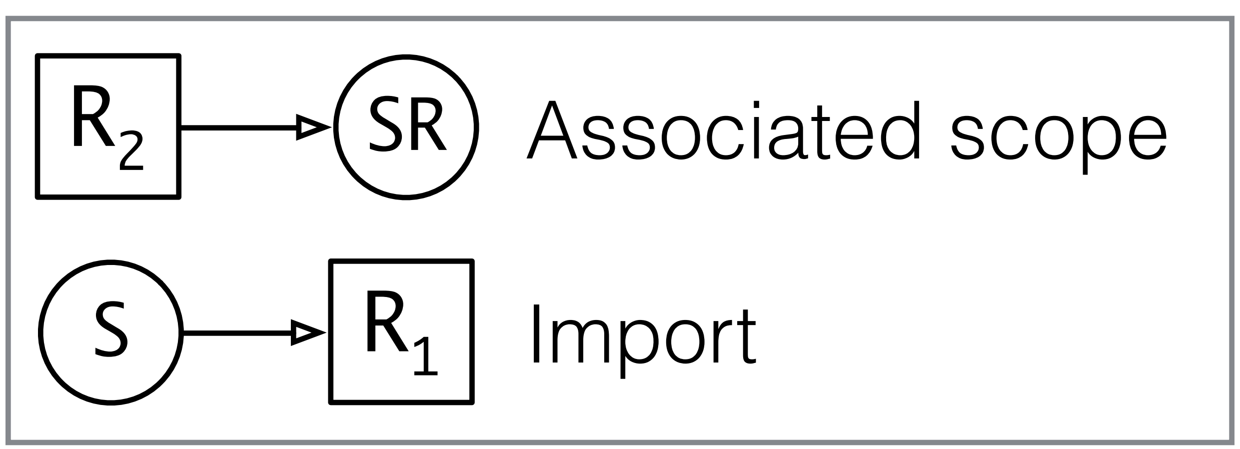




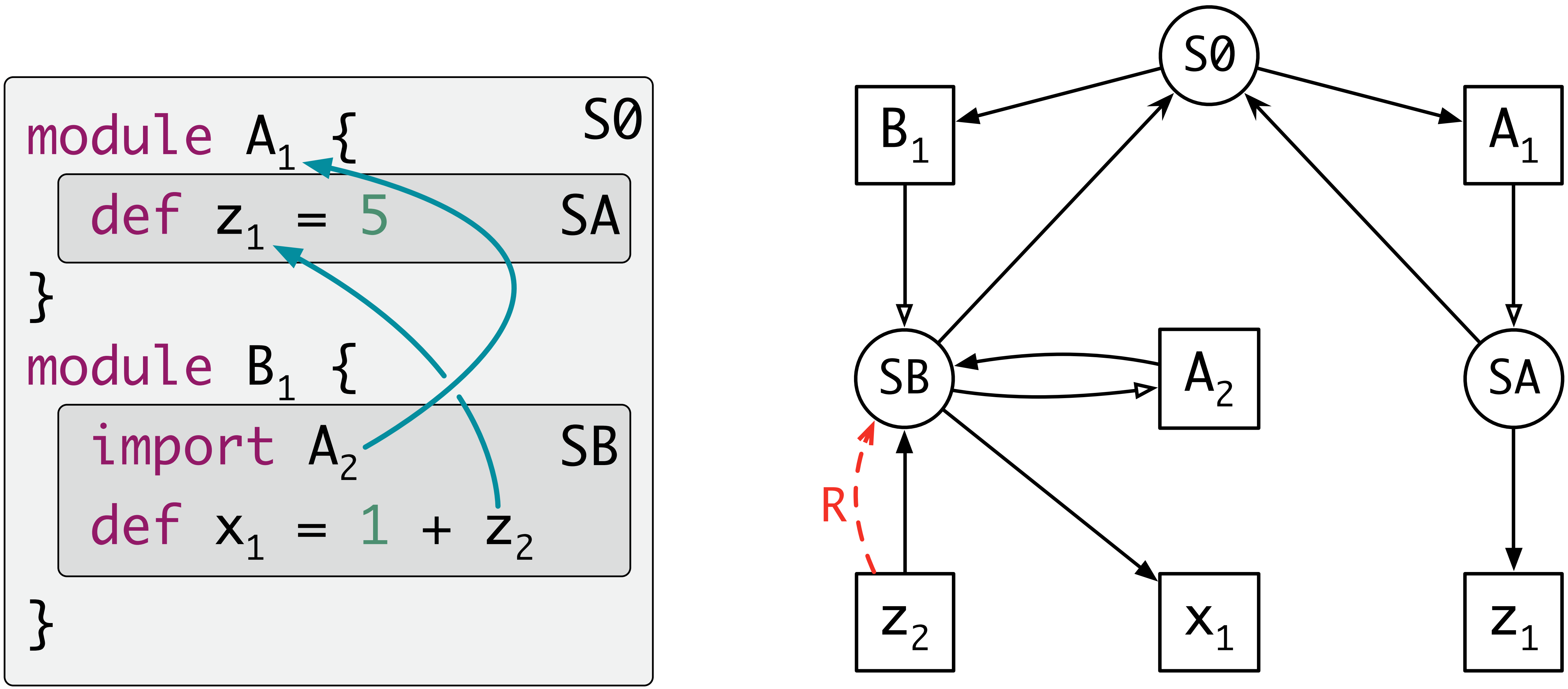
Imports

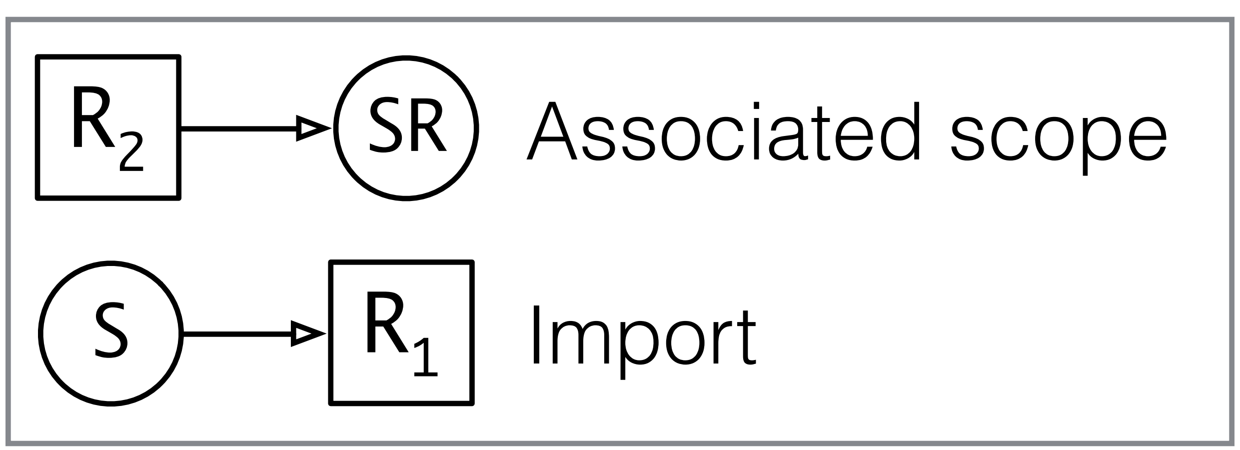

module A1 {
def z1 = 5
}
module B1 {
import A2
def x1 = 1 + z2
}
S0
SA
SB


S0
B1
A1
SB
A2
SA
z2
x1
z1

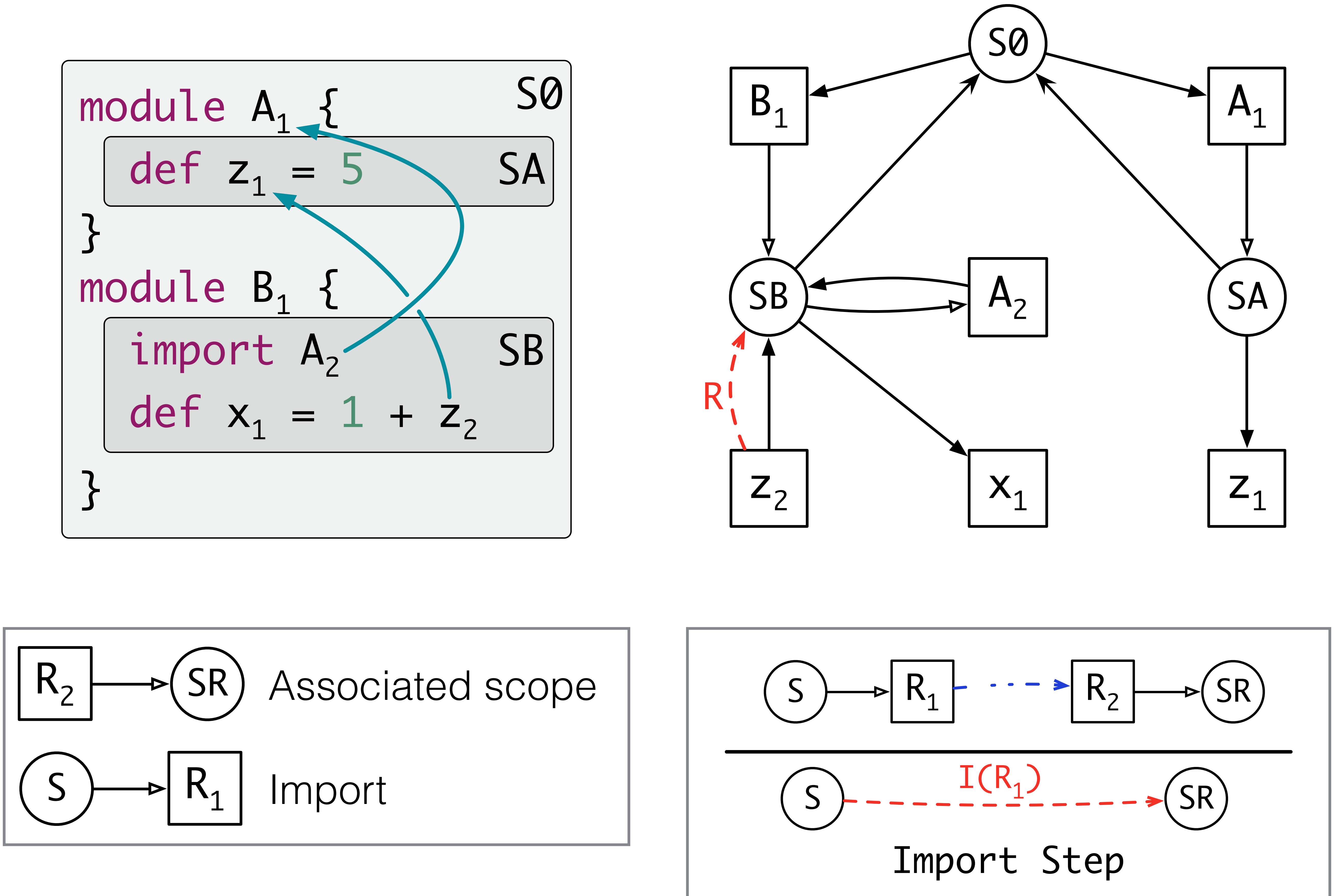

R2
SR
Associated scope
S
R1
Import

Imports


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

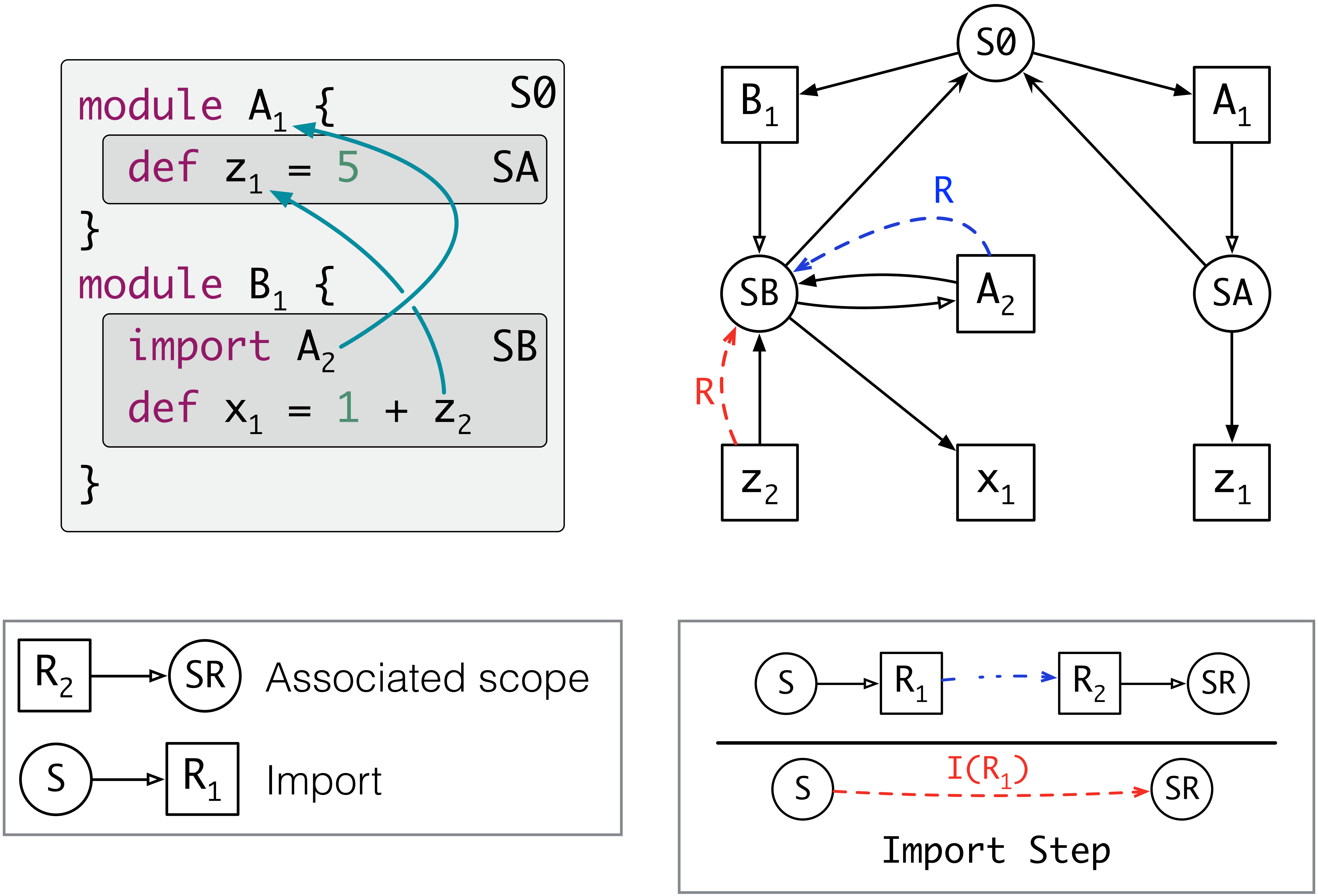

R2
SR
Associated scope
S
R1
Import

Imports



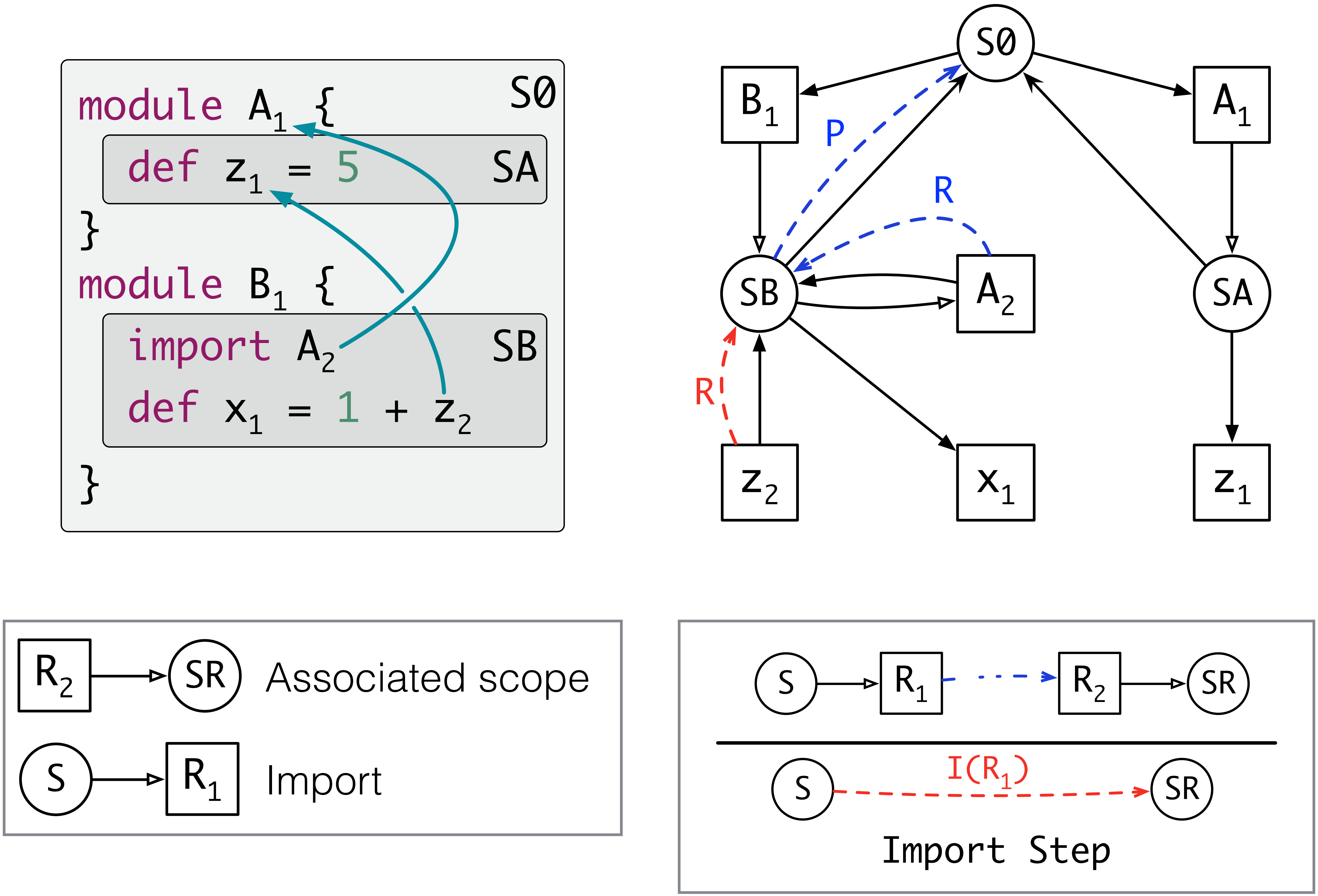
module A1 {
def z1 = 5
SA
}
S0
module B1 {
import A2
def x1 = 1 + z2
SB
}
S0
B1
A1
SB
A2
SA
R
z2
x1
z1
R2
SR
Associated scope
S
R1
Import
S
R1
R2
SR
I(R1)
S
SR
Import Step

Imports

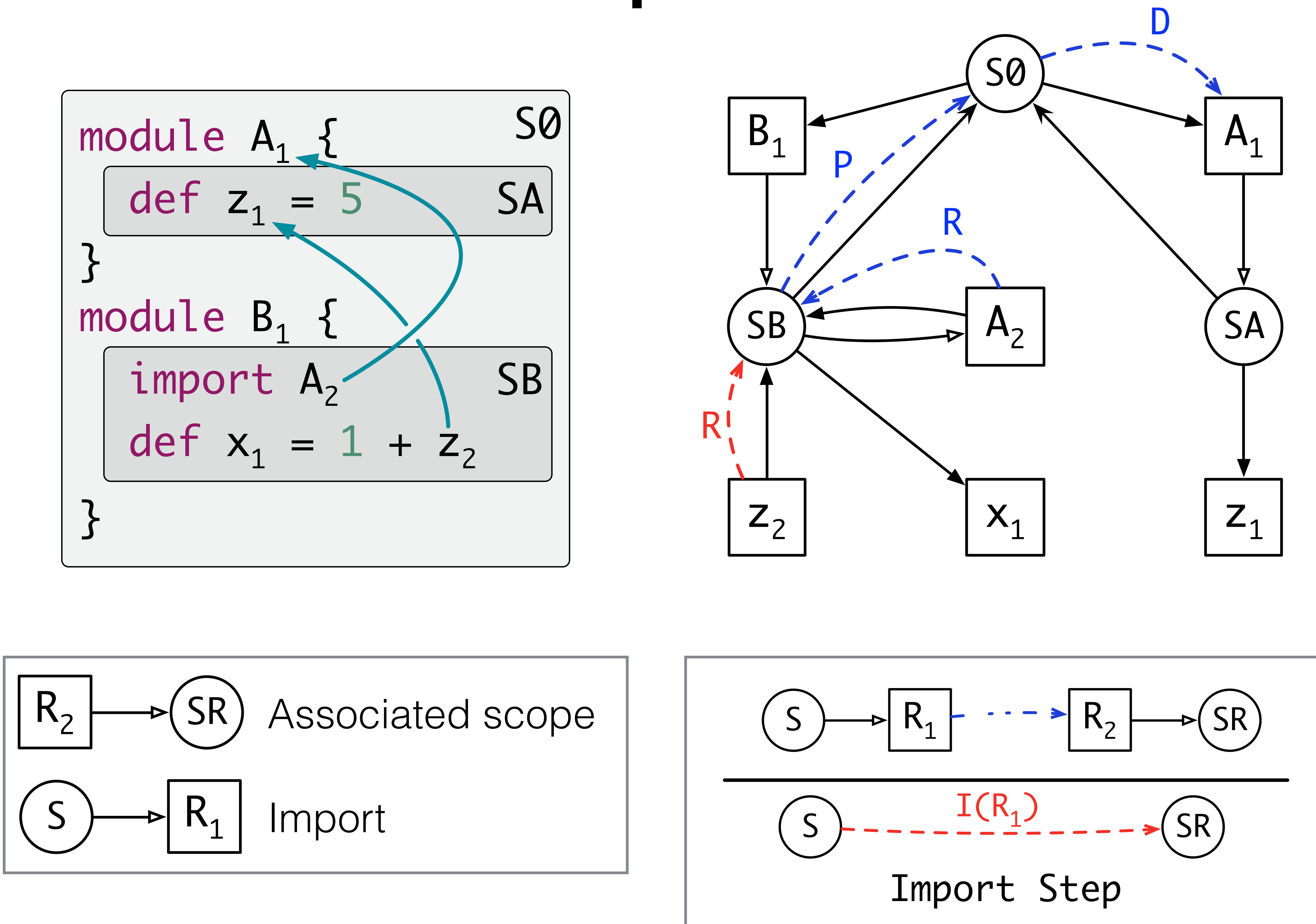


module A1 {
def z1 = 5
SA
}
S0
module B1 {
import A2
def x1 = 1 + z2
SB
}
S0
B1
A1
SB
A2
SA
R
R
z2
x1
z1
R2
SR
Associated scope
S
R1
Import
S
R1
R2
SR
S
I(R1)
SR
Import Step

Imports

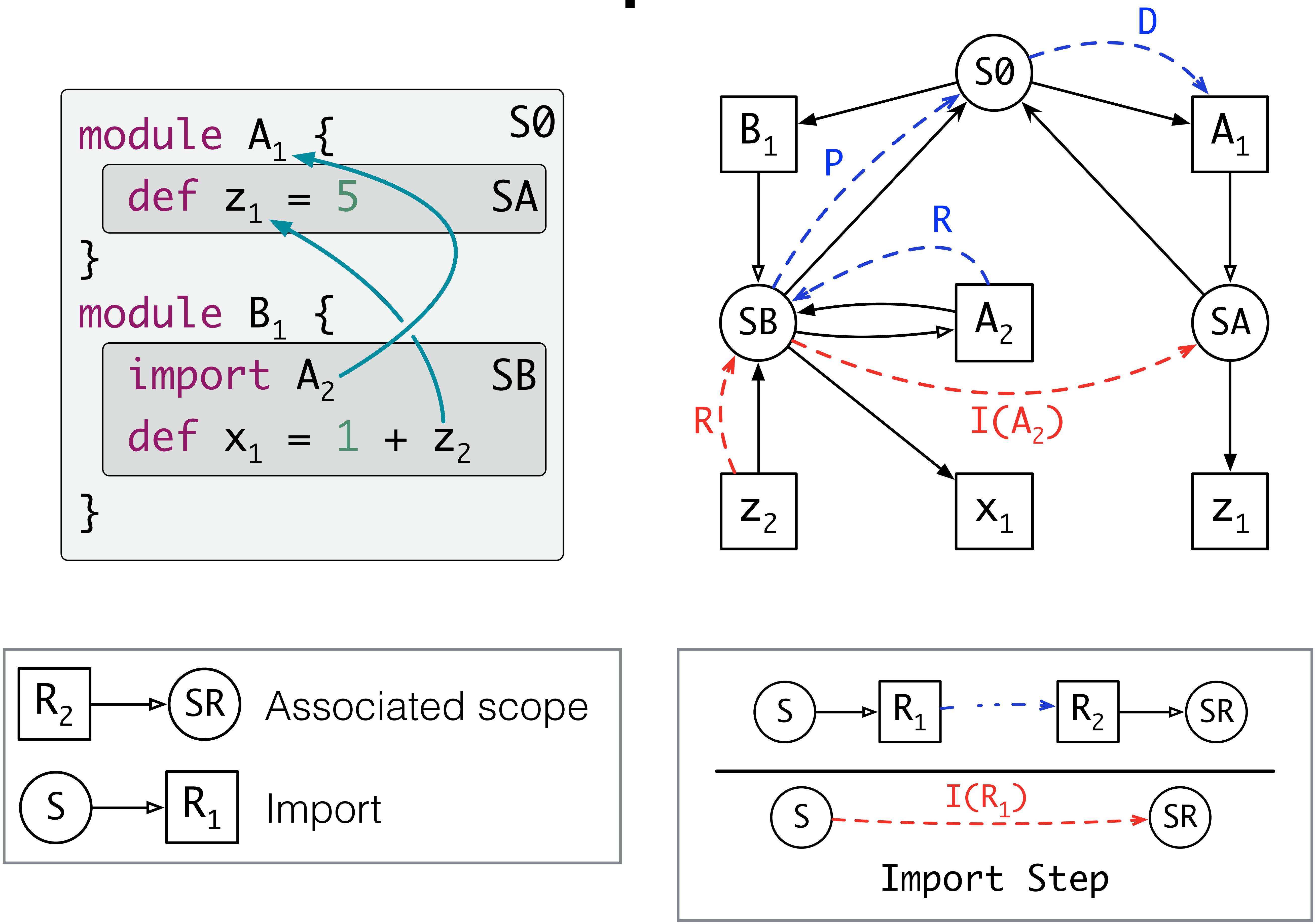

module A1 {
def z1 = 5
}
module B1 {
import A2
def x1 = 1 + z2
}
S0
SA
SB
B1
A1
A2
SB
SA
z2
x1
z1
P
R
R
Associated scope
Import
S
R1
R2
SR
I(R1)
Import Step

Imports



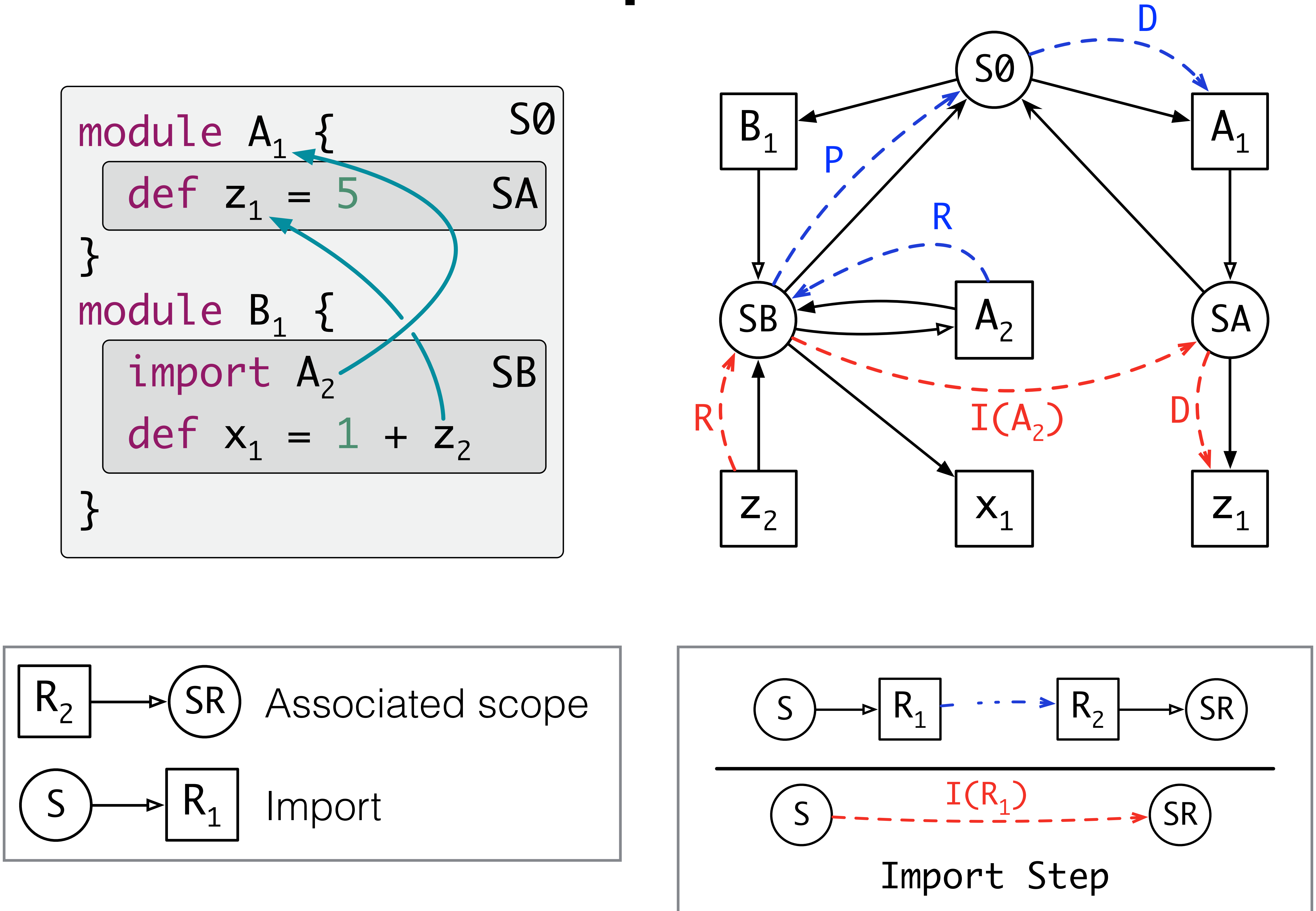
module A1 {
def z1 = 5
}
module B1 {
import A2
def x1 = 1 + z2
}
S0
SA
SB
D
P
R
R
S0
B1
A1
SB
A2
SA
z2
x1
z1
R2
SR
Associated scope
S
R1
Import
S
R1
R2
SR
S
I(R1)
SR
Import Step

Imports



module A1 {
def z1 = 5
}
module B1 {
import A2
def x1 = 1 + z2
}
S0
SA
SB
D
P
R
R
I(A2)
B1
A1
SB
A2
SA
z2
x1
z1
R2
SR
Associated scope
S
R1
Import
S
R1
R2
SR
S
I(R1)
SR
Import Step

Imports



module A1 {
def z1 = 5
SA
}
S0
module B1 {
import A2
def x1 = 1 + z2
SB
}
S0
B1
A1
SB
A2
SA
z2
x1
z1
D
P
R
R
I(A2)
D
R2
SR
Associated scope
S
R1
Import
S
R1
R2
SR
S
SR
I(R1)
Import Step

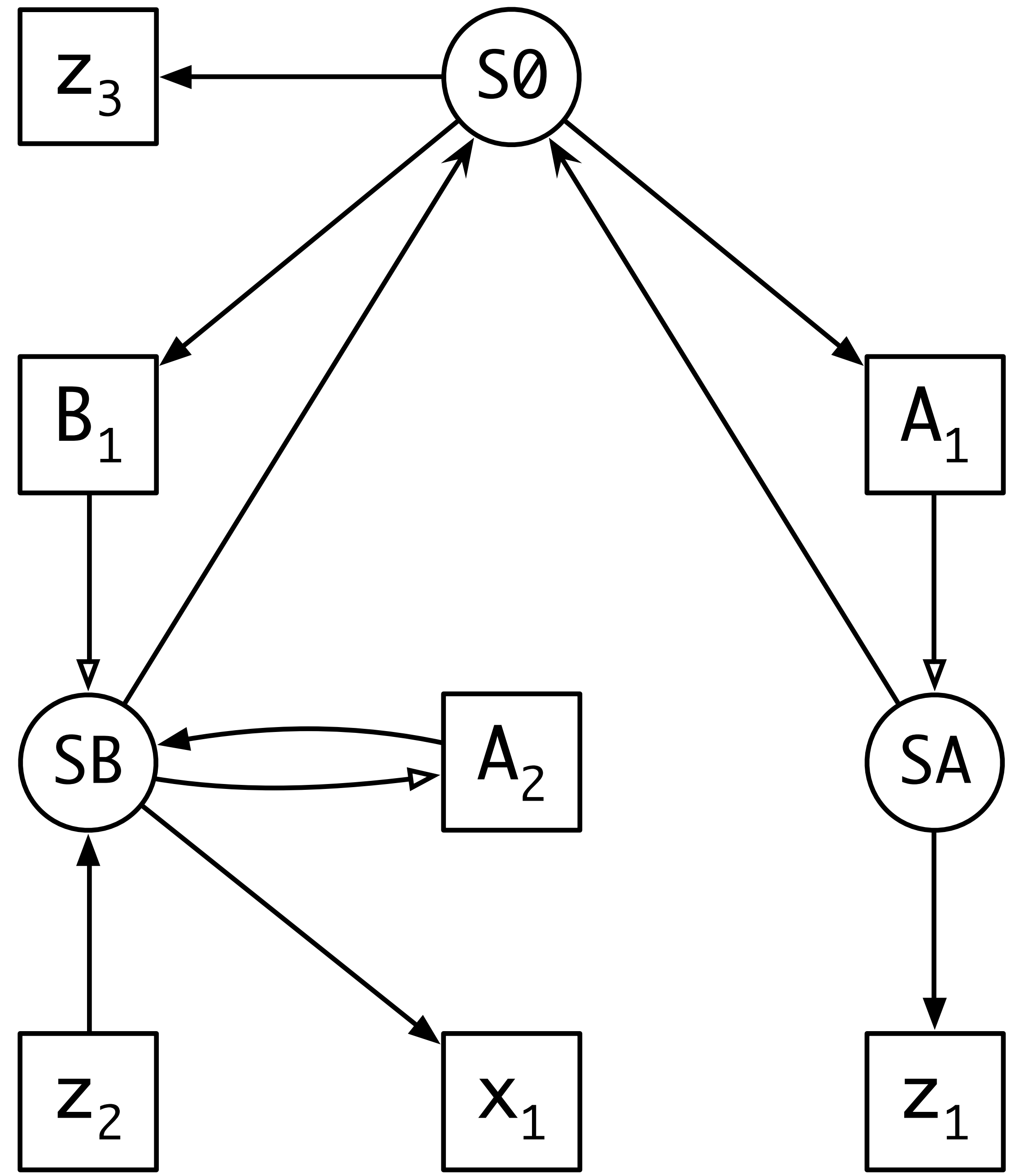
Imports shadow Parents

S0

```
def z3 = 2

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

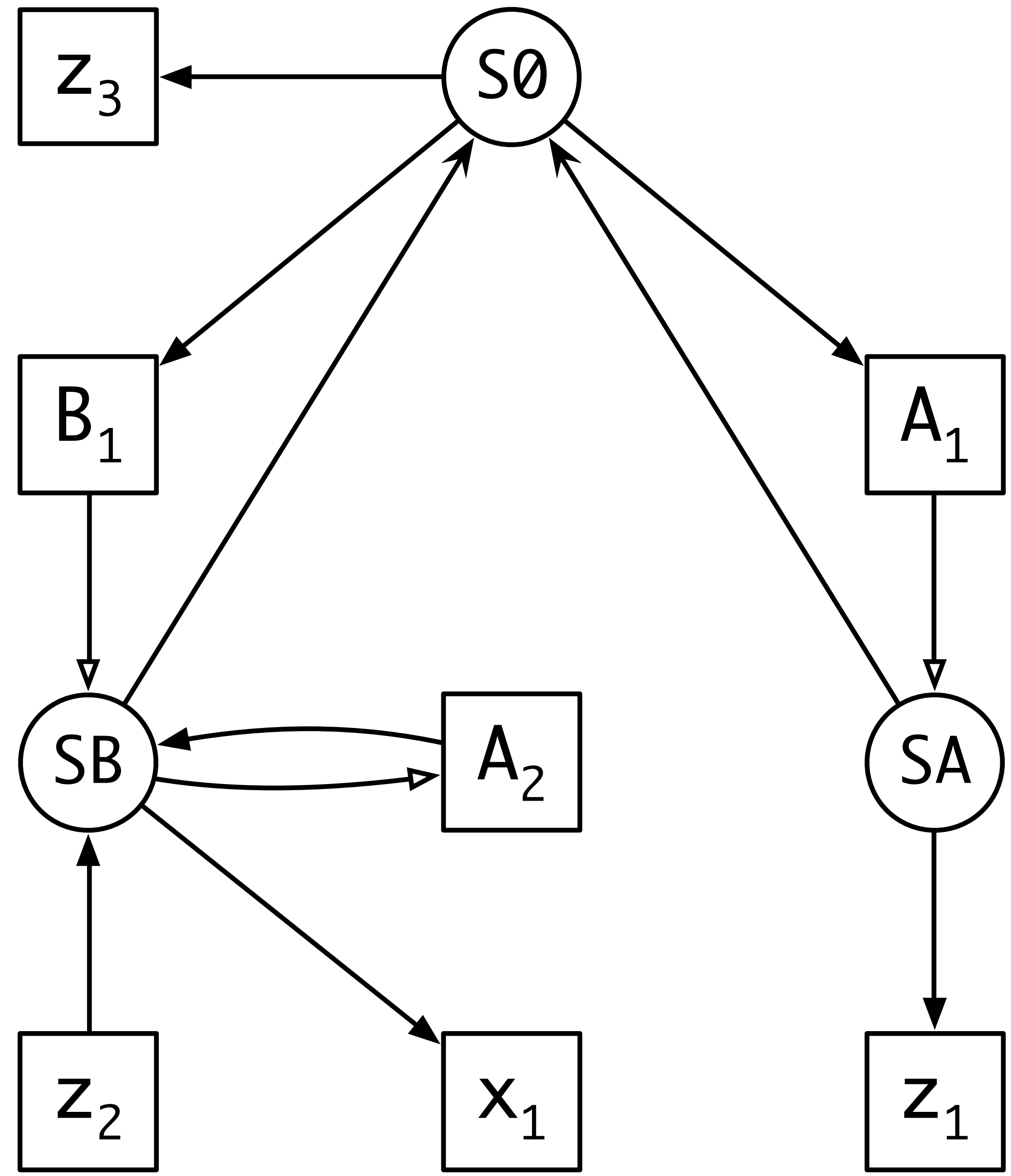
Imports shadow Parents

```
def z3 = 2                    S0

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



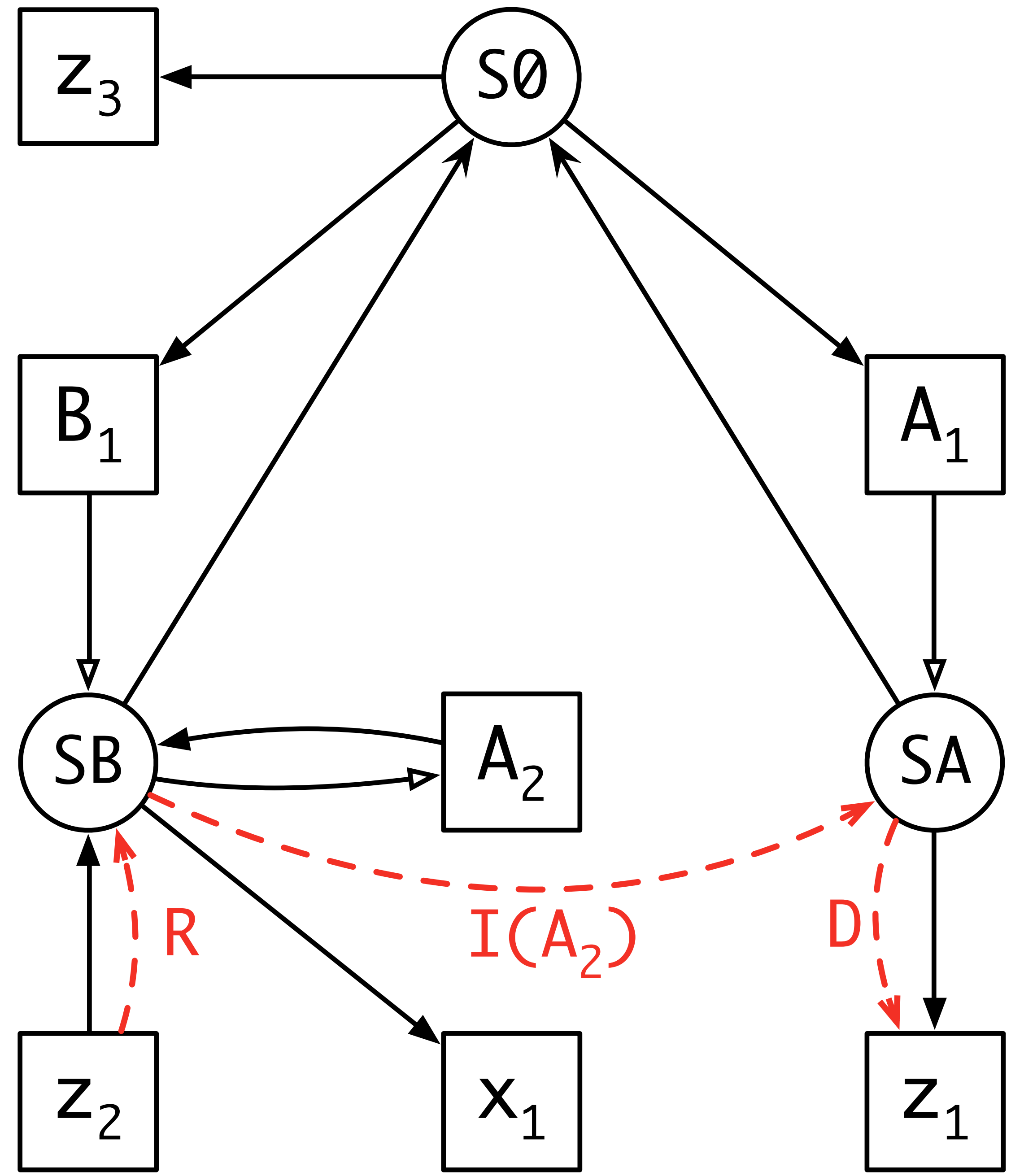
Imports shadow Parents

```
def z_3 = 2                              S0

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



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

S0

def $z_3 = 2$

module A_1 {

SA

def $z_1 = 5$

}

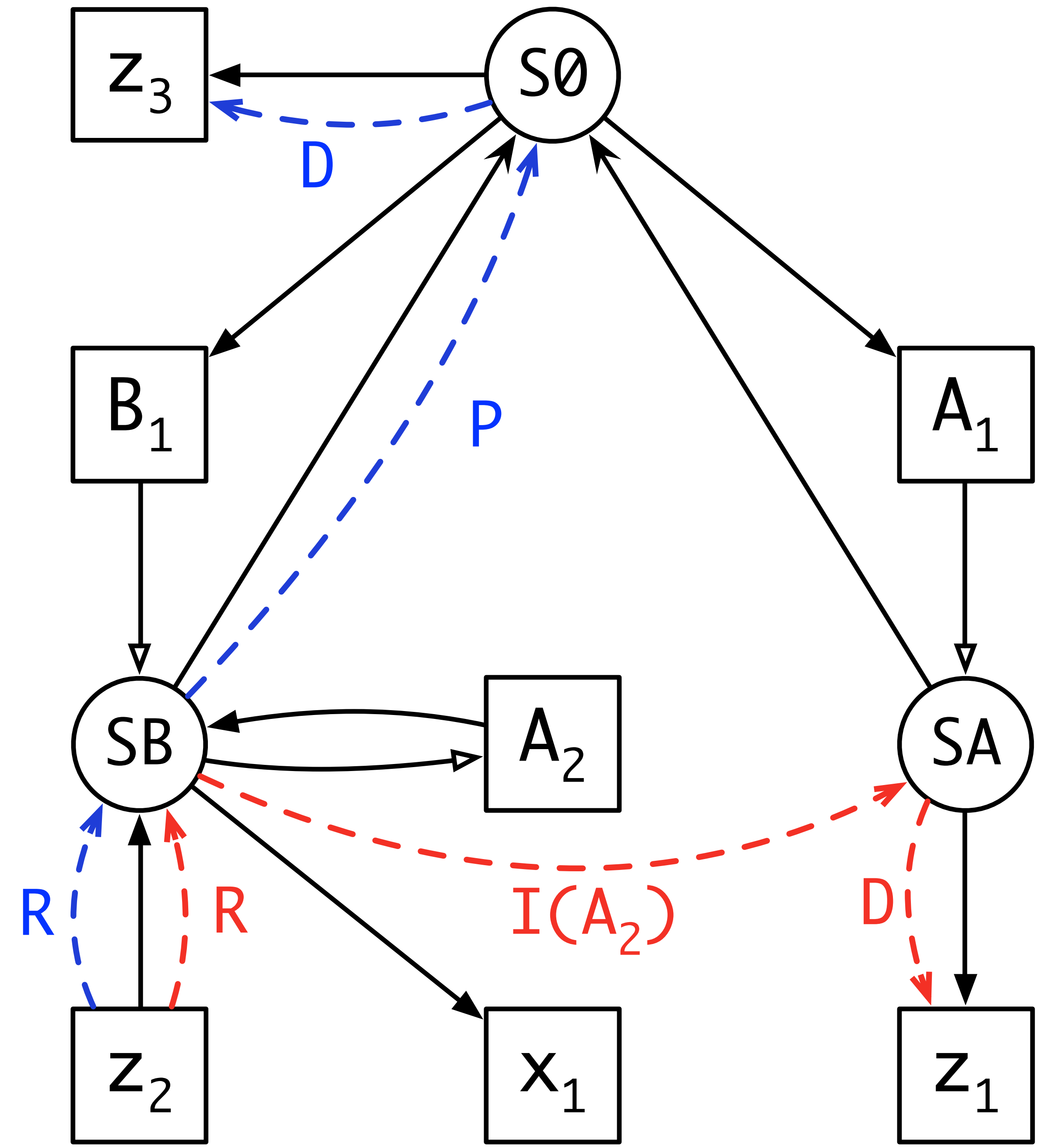
module B_1 {

SB

import A_2

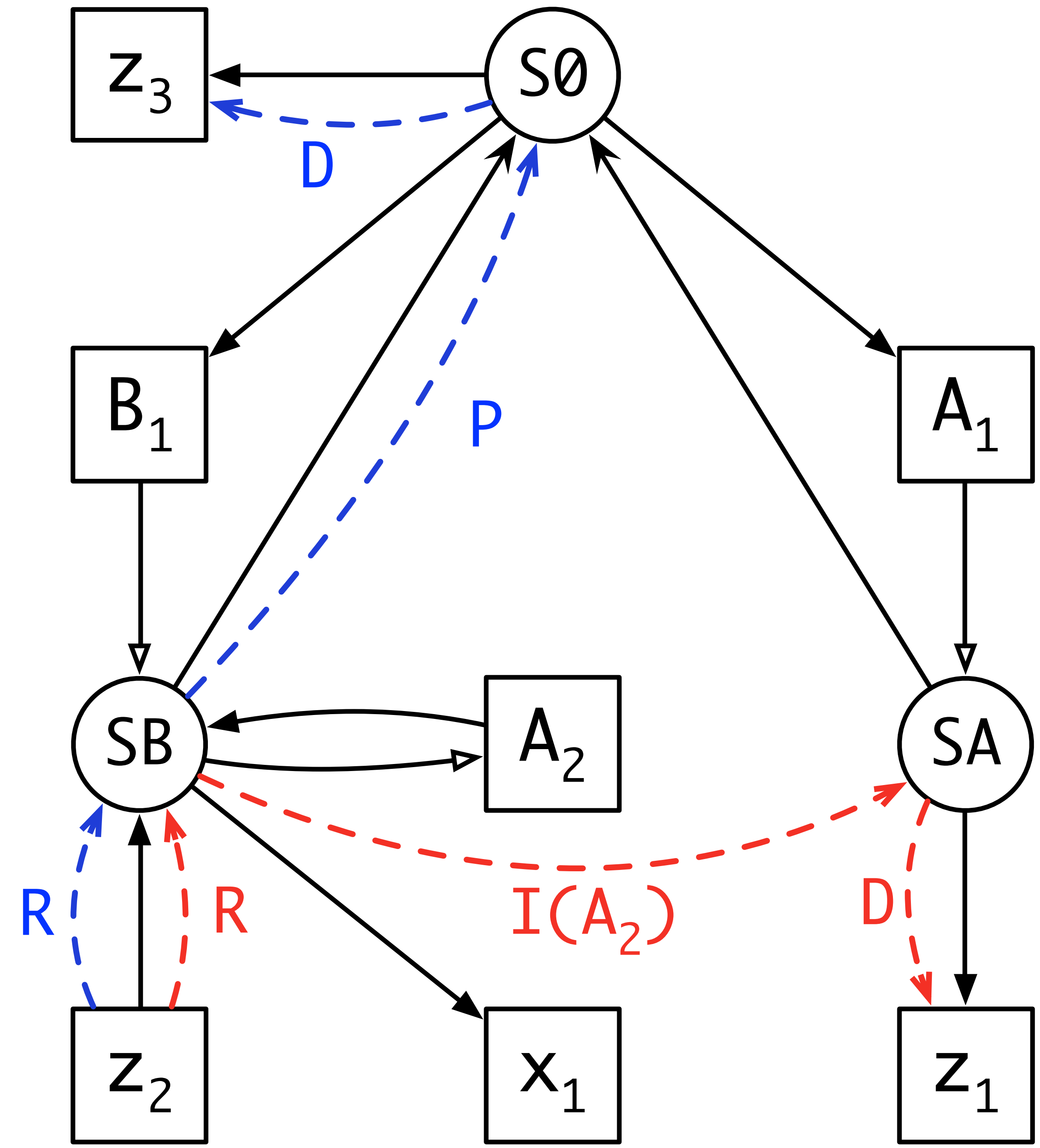
def $x_1 = 1 + z_2$

}



Imports shadow Parents

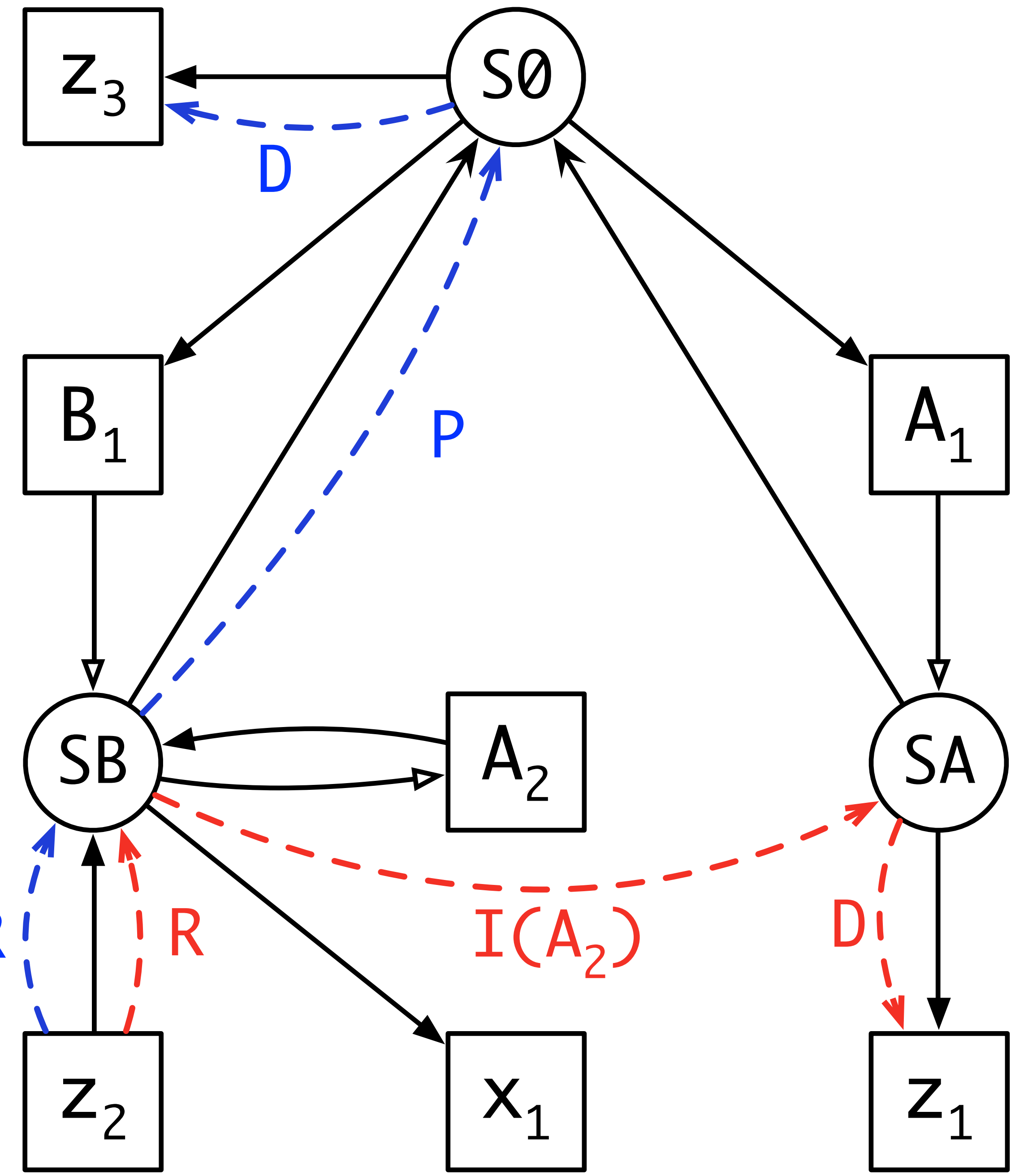
```
def z3 = 2                                S0

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

S0

$\texttt{def } z_3 = 2$

$\texttt{module } A_1 \ \{$

$\quad \texttt{def } z_1 = 5$ SA

$\}$

$\texttt{import } A_2$

$\texttt{def } x_1 = 1 + z_2$

Imports vs. Includes

S0

def z_3 = 2

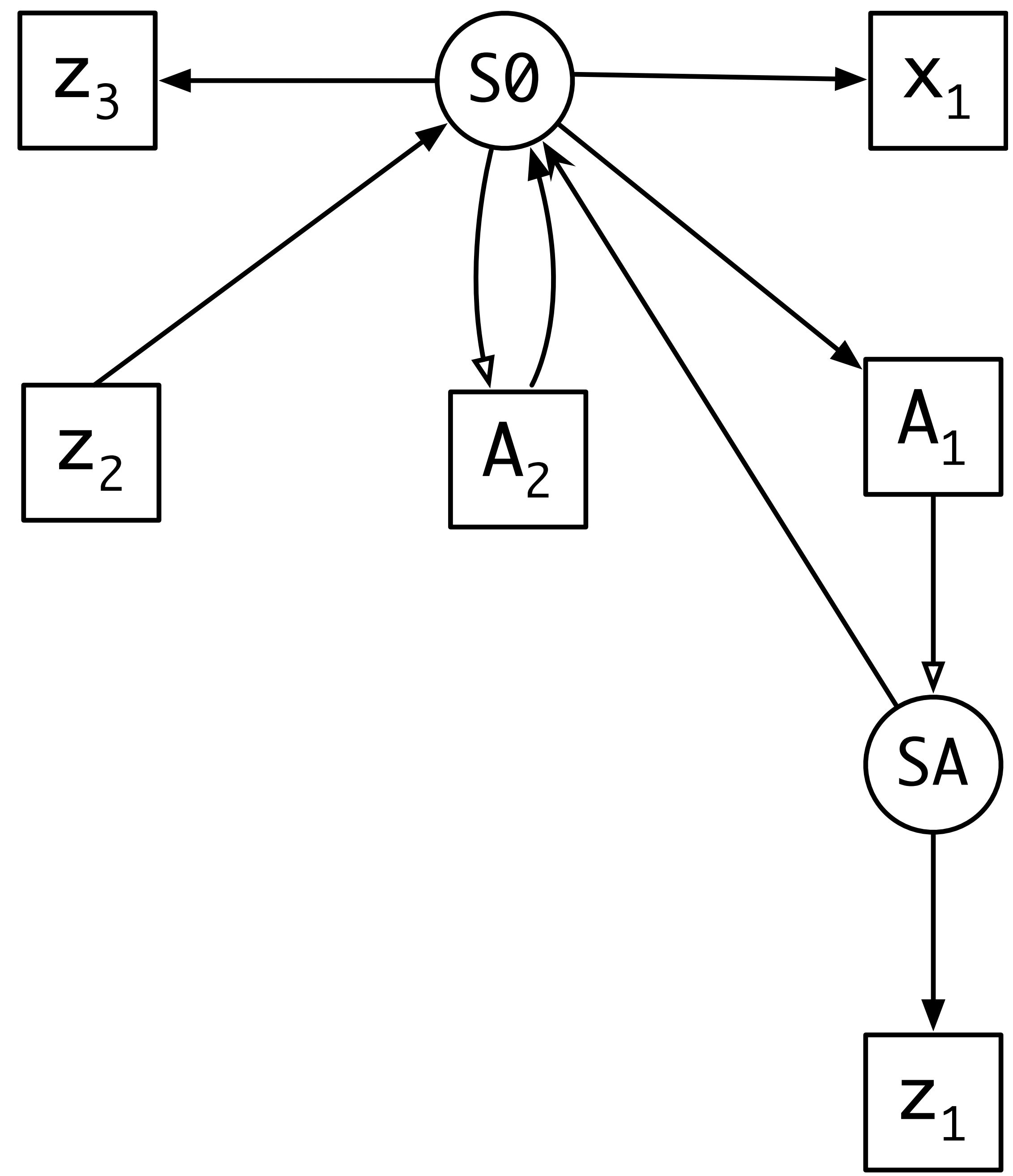
module A_1 {

def z_1 = 5 SA

}

import A_2

def x_1 = 1 + z_2



Imports vs. Includes

S0

$\texttt{def}\ z_3 = 2$

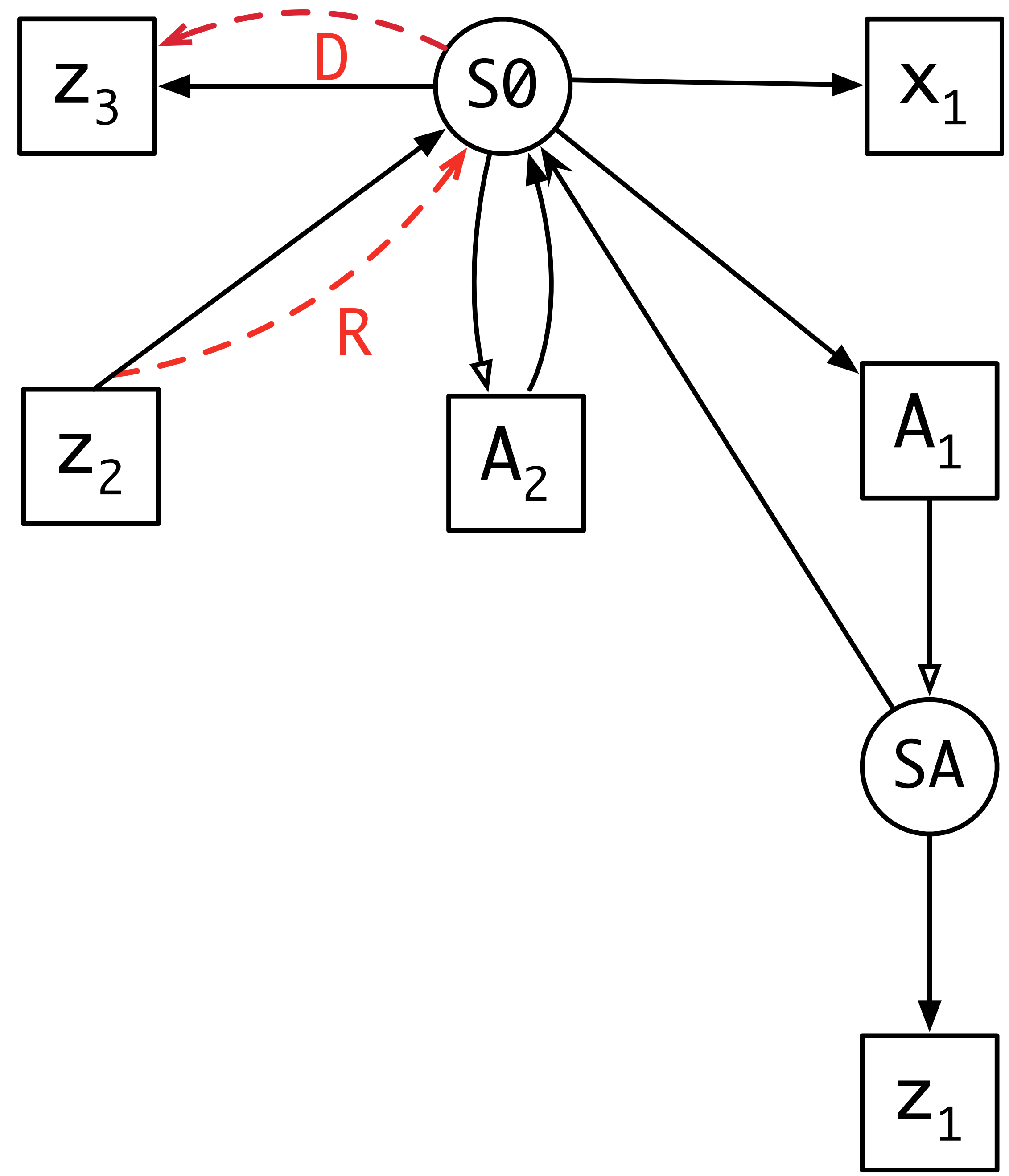
$\texttt{module}\ A_1\ \{$

$\quad \texttt{def}\ z_1 = 5$ SA

$\}$

$\texttt{import}\ A_2$

$\texttt{def}\ x_1 = 1 + z_2$

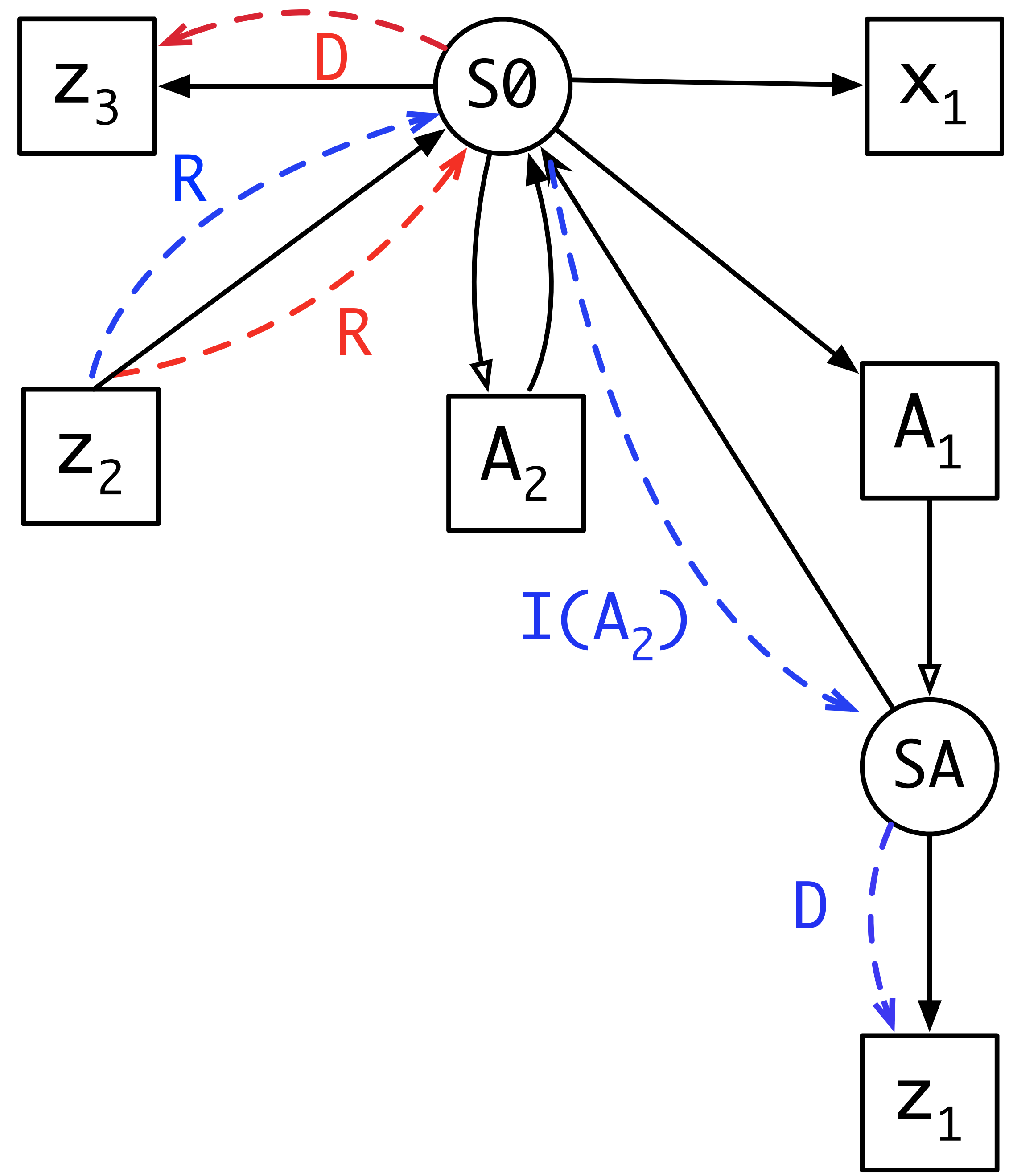


Imports vs. Includes

S0

```
def z₃ = 2

module A₁ {
  def z₁ = 5
}

import A₂
def x₁ = 1 + z₂
```

SA



Imports vs. Includes

S0

def $z_3 = 2$

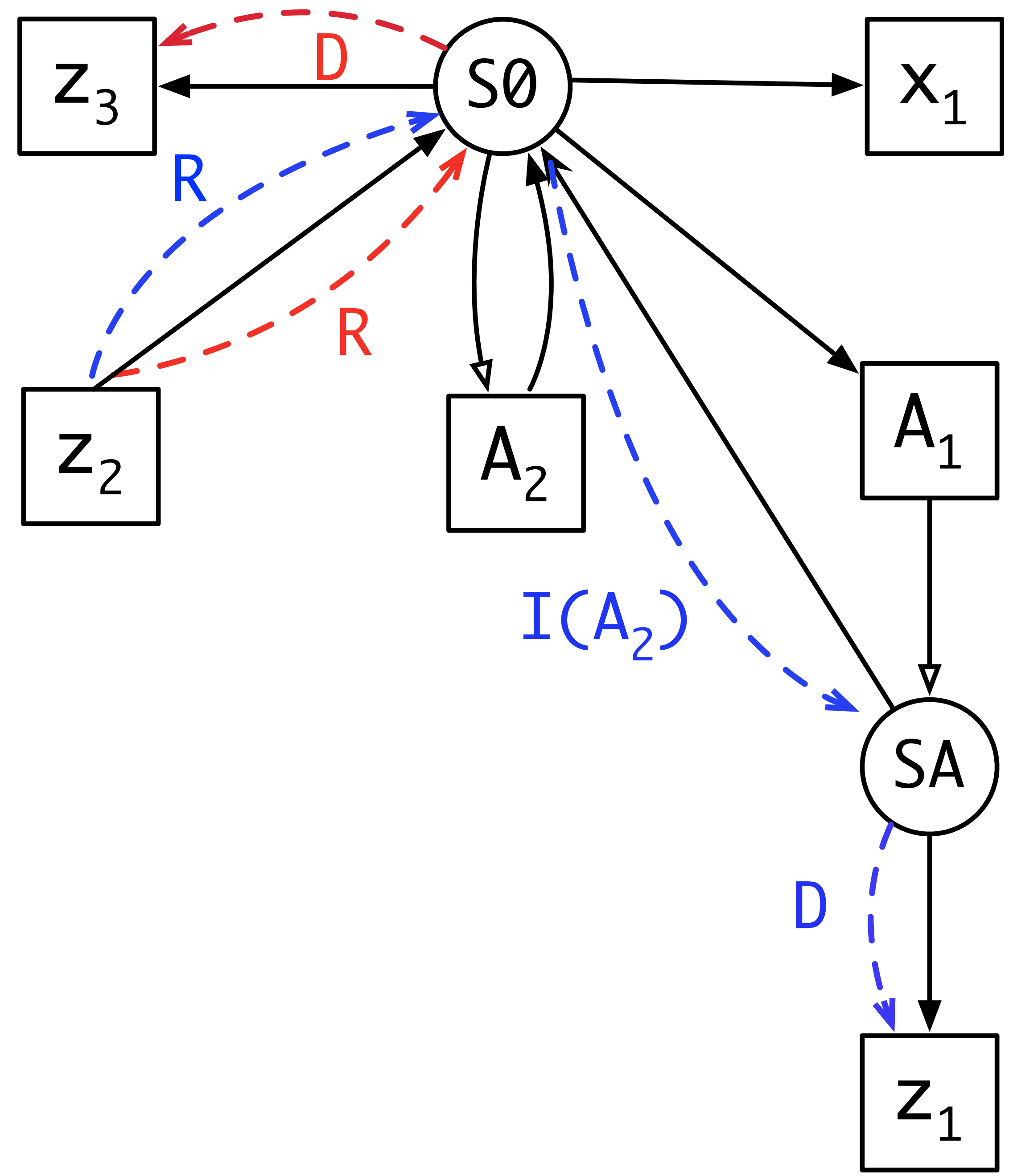
module A_1 {

def $z_1 = 5$ SA

}

import A_2

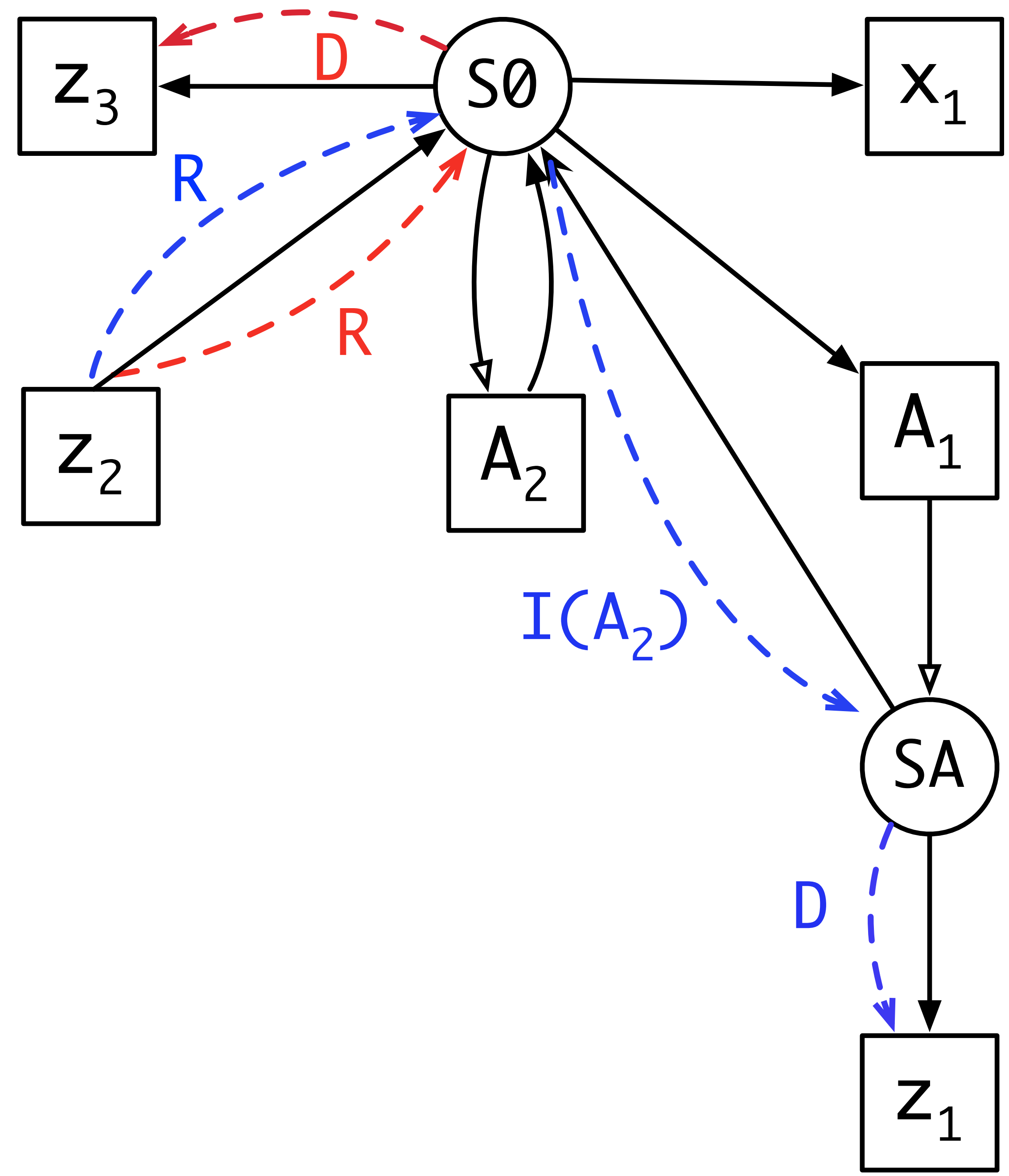
def $x_1 = 1 + z_2$



D < I(_).p’

Imports vs. Includes

```
def z₃ = 2                      S0

module A₁ {
    def z₁ = 5                  SA
}

import A₂
def x₁ = 1 + z₂
```



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

Imports vs. Includes

S0

$\mathtt{def}\ z_3 = 2$

$\mathtt{module}\ A_1\ \{$

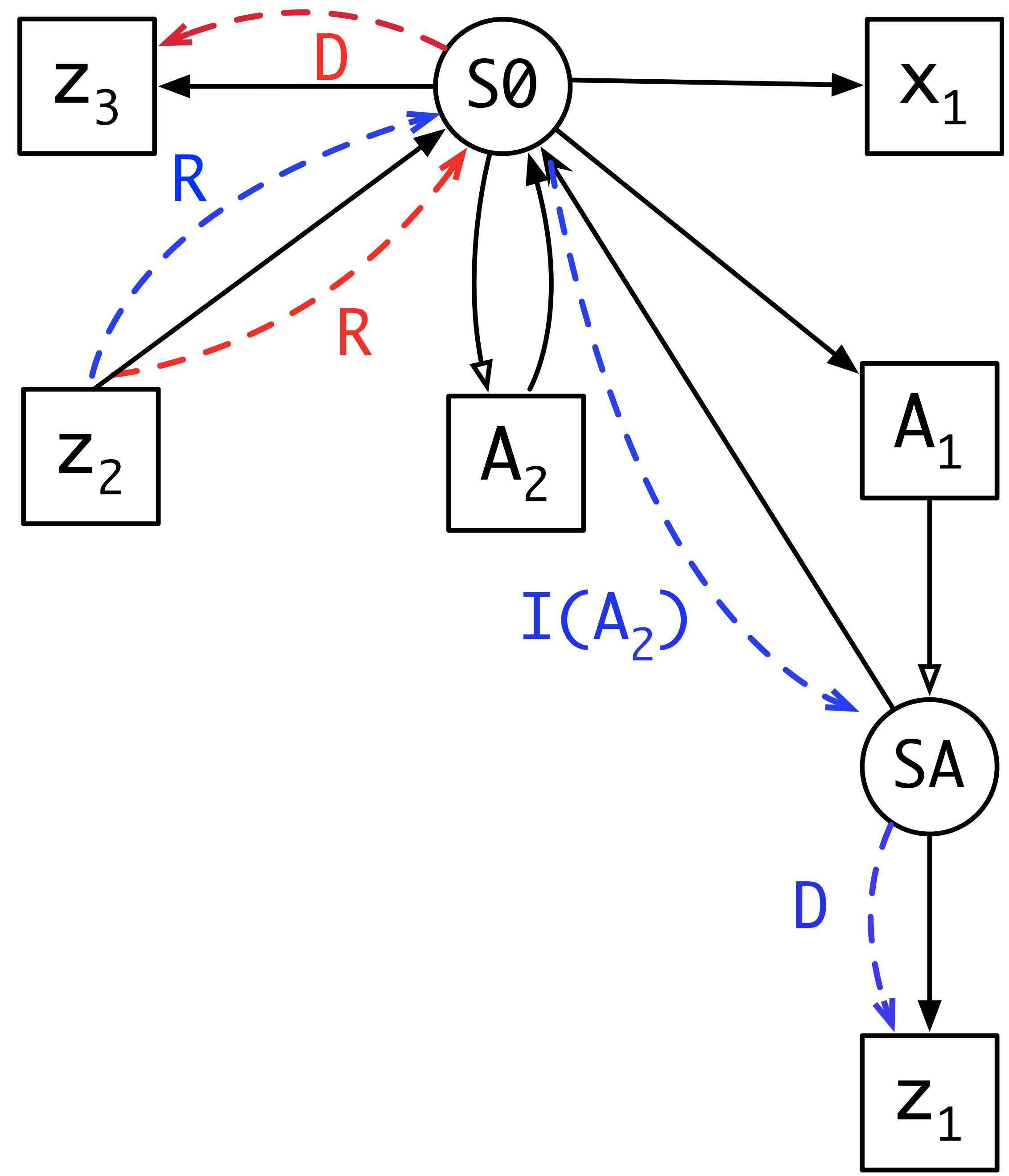
SA

$\quad \mathtt{def}\ z_1 = 5$

$\}$

$\mathtt{include}\ A_2$

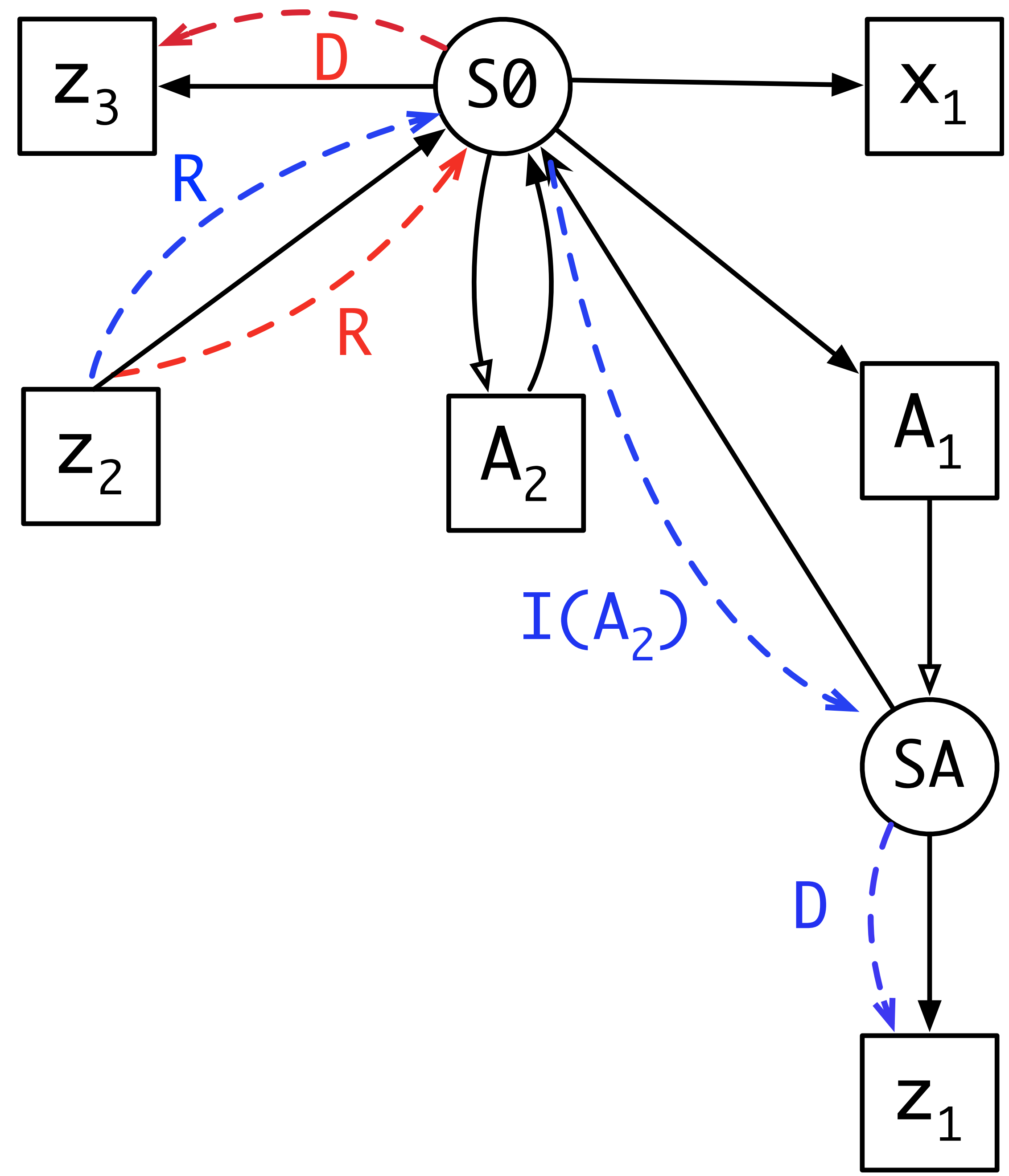
$\mathtt{def}\ x_1 = 1 + z_2$



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

Imports vs. Includes

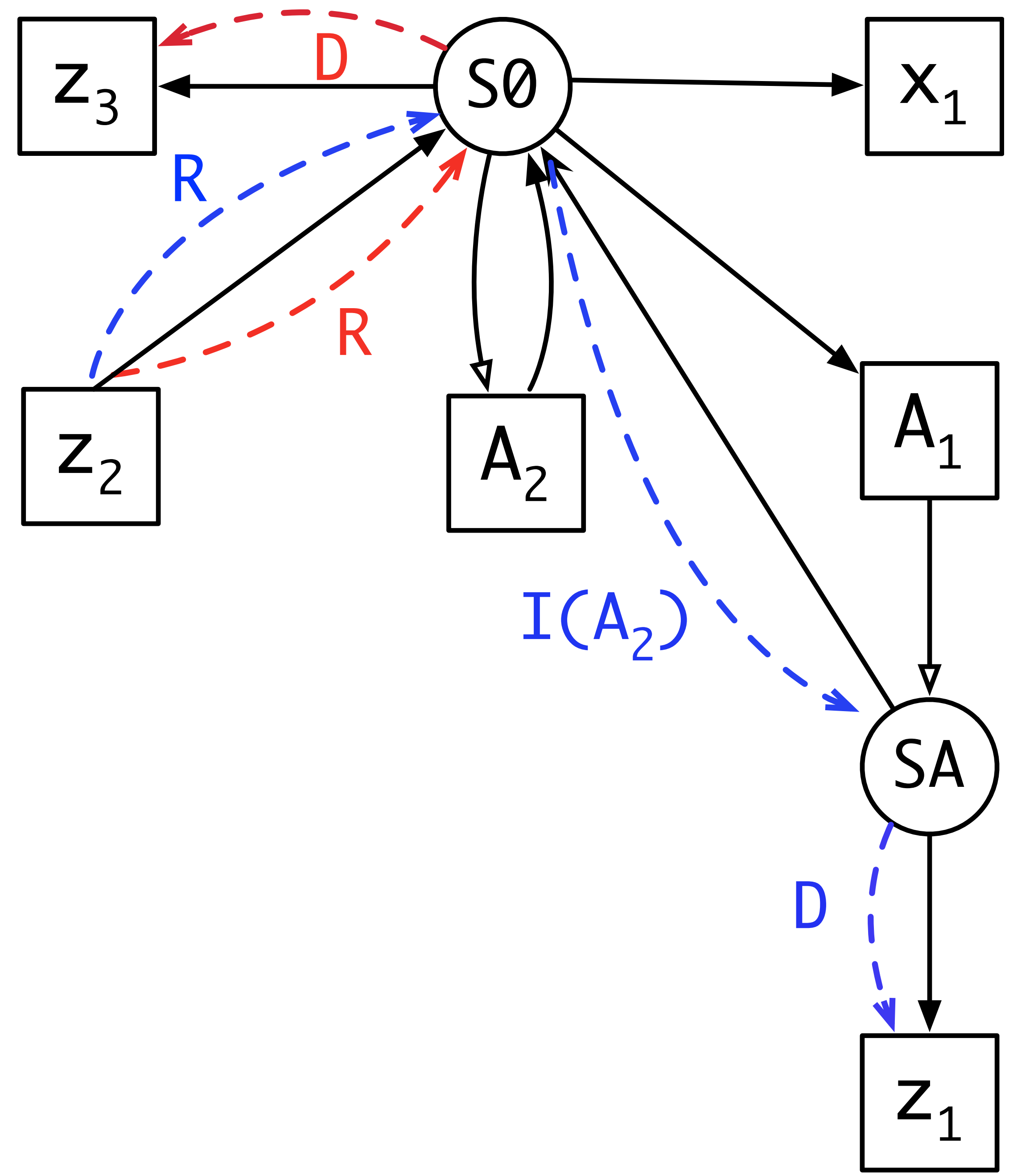
```
def z₃ = 2                 S0

module A₁ {
  def z₁ = 5               SA
}

include A₂
def x₁ = 1 + z₂
```



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

Imports vs. Includes

```
def z3 = 2                S0

module A1 {
    def z1 = 5            SA
}

include A2
def x1 = 1 + z2
```



$$\cancel{\dfrac{}{D < I(\_).p'}}$$

Qualified Names

S0

```
module N₁ {
    def s₁ = 5
}

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

Qualified Names

S0

$\texttt{module}\ N_1\ \{$

$\quad \texttt{def}\ s_1 = 5$

$\}$

$\texttt{module}\ M_1\ \{$

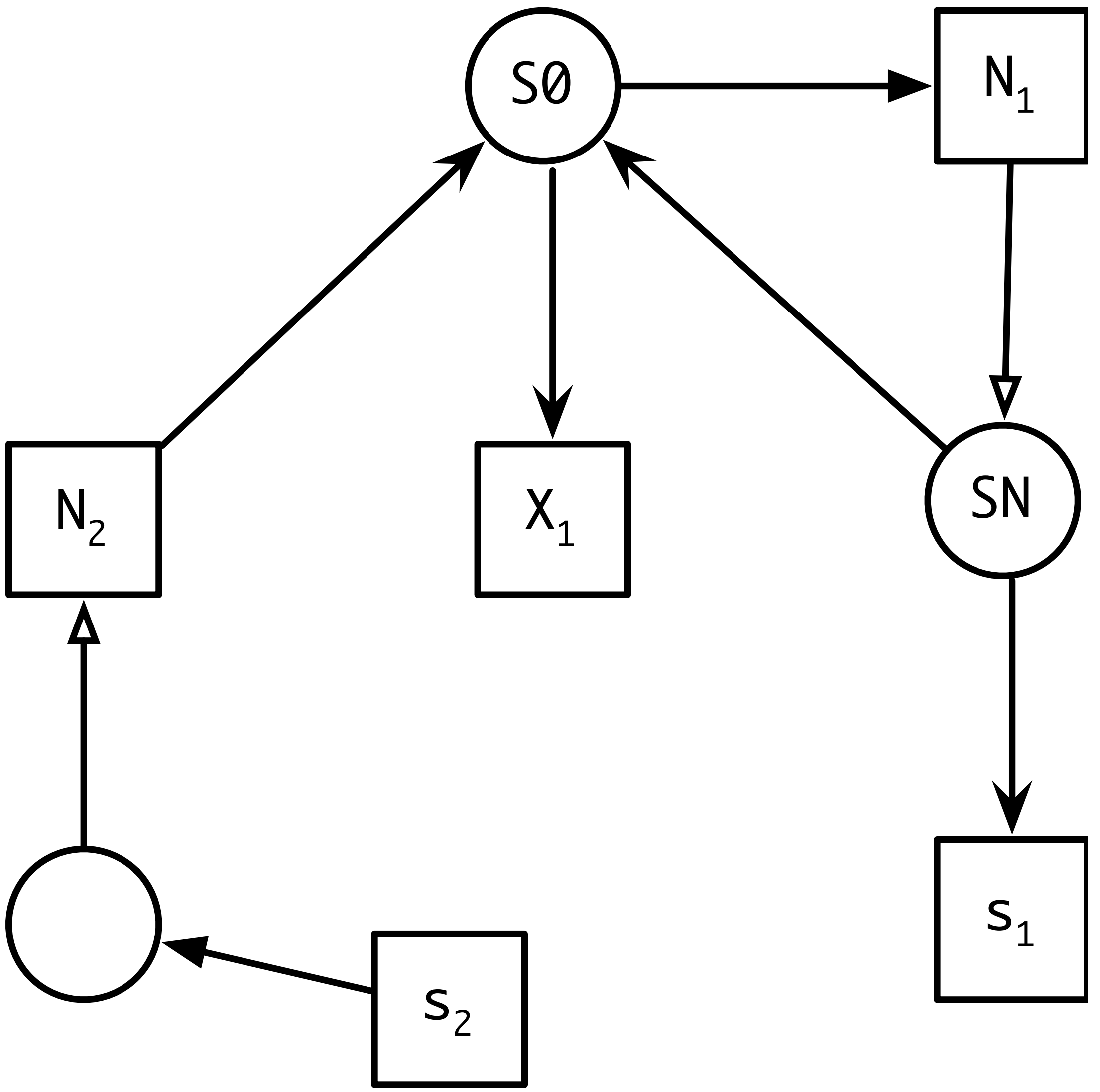
$\quad \texttt{def}\ x_1 = 1 + N_2.s_2$

$\}$

Qualified Names

S0

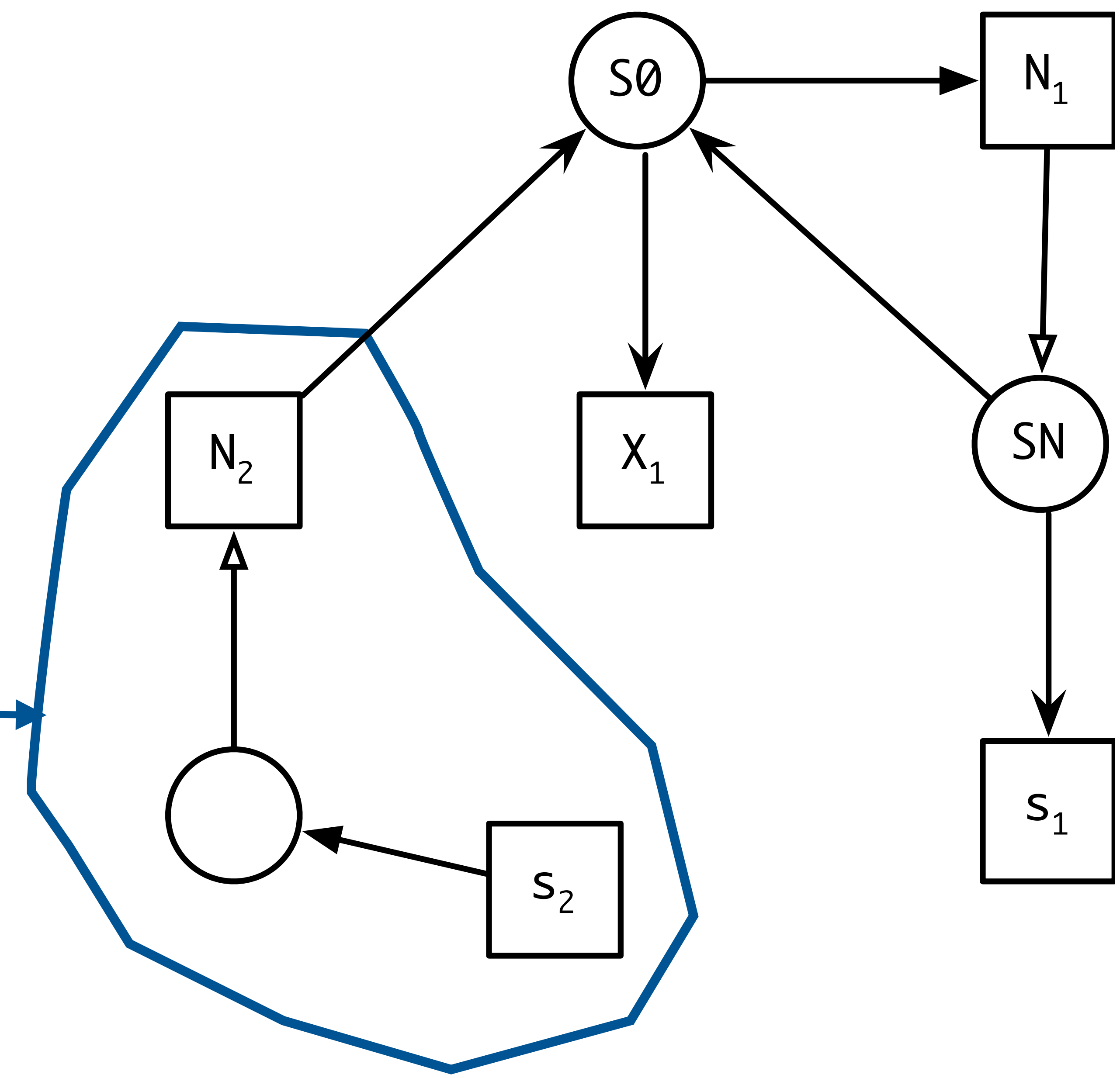
```
module N₁ {
    def s₁ = 5
}

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



Qualified Names

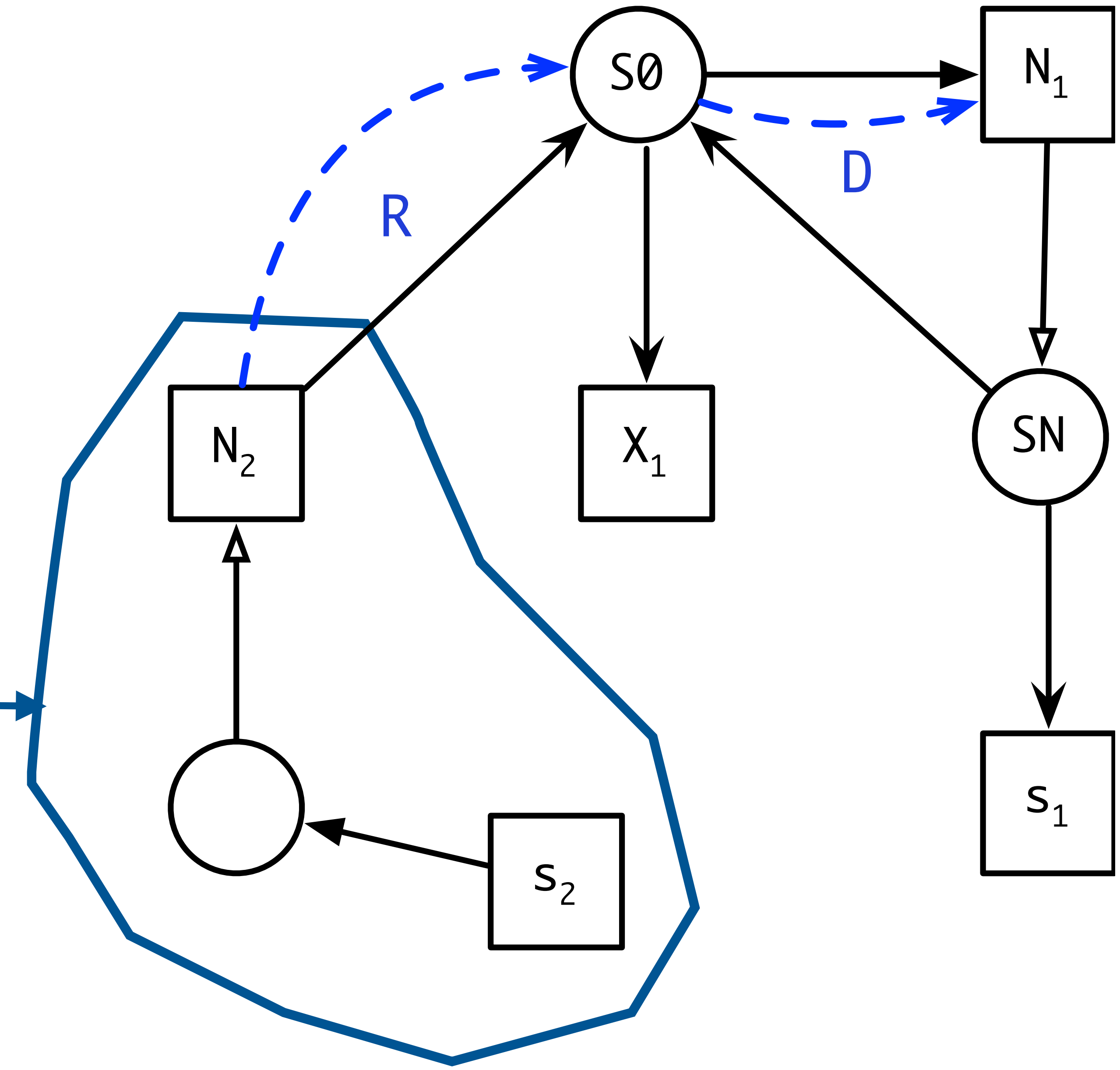
S0

```
module N₁ {
    def s₁ = 5
}

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



Qualified Names

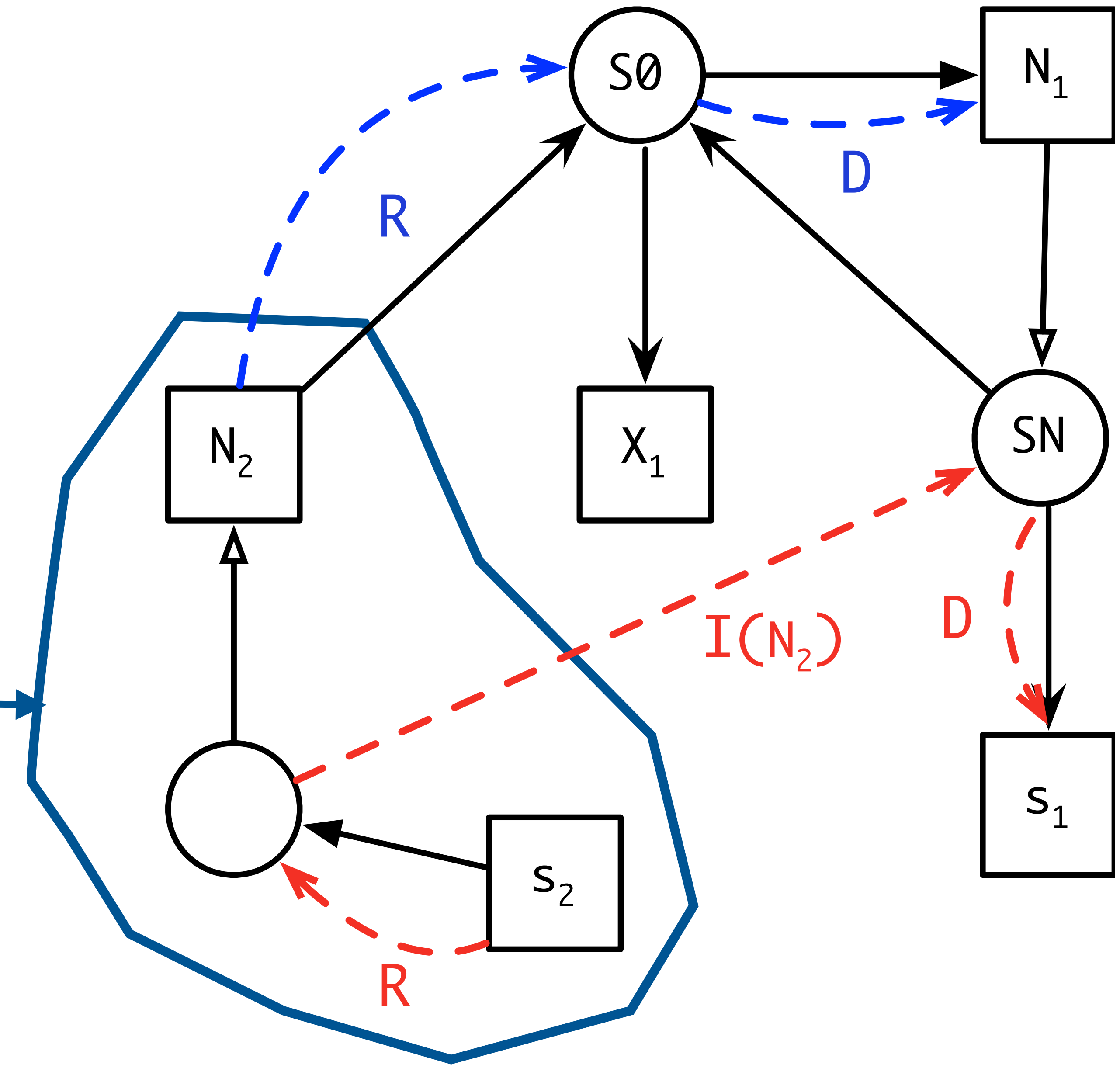
```
module N₁ {
    def s₁ = 5
}

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



Qualified Names

S0

```
module N1 {
    def s1 = 5
}

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

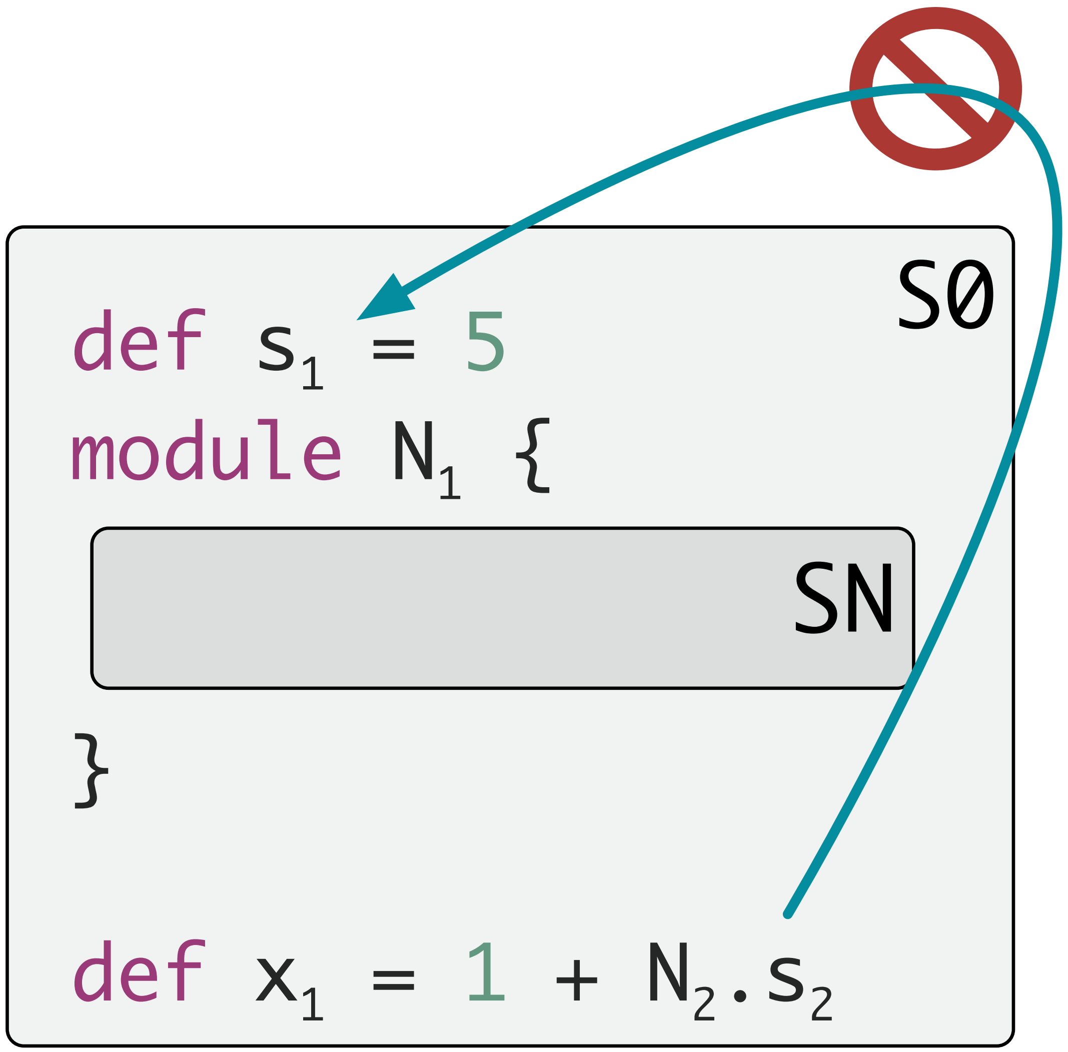


Import Parents

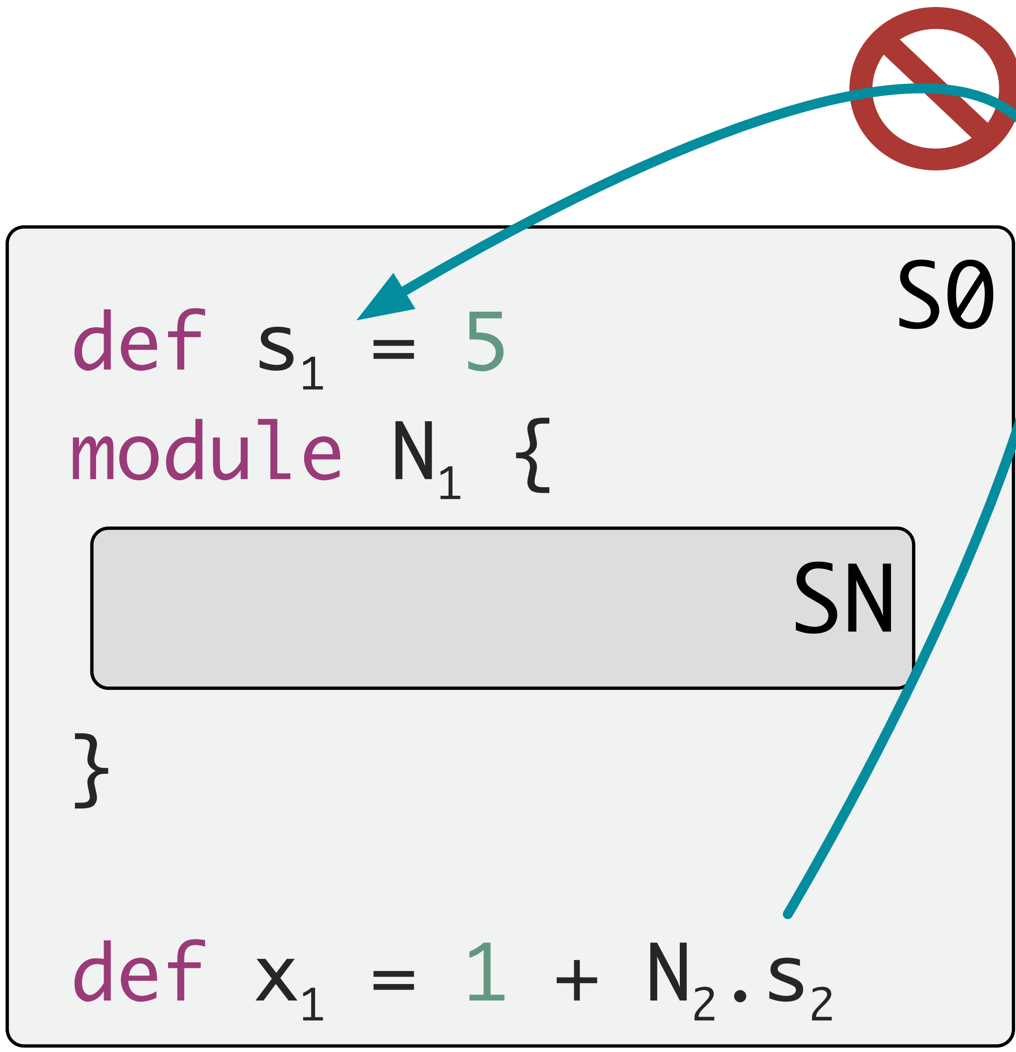
S0

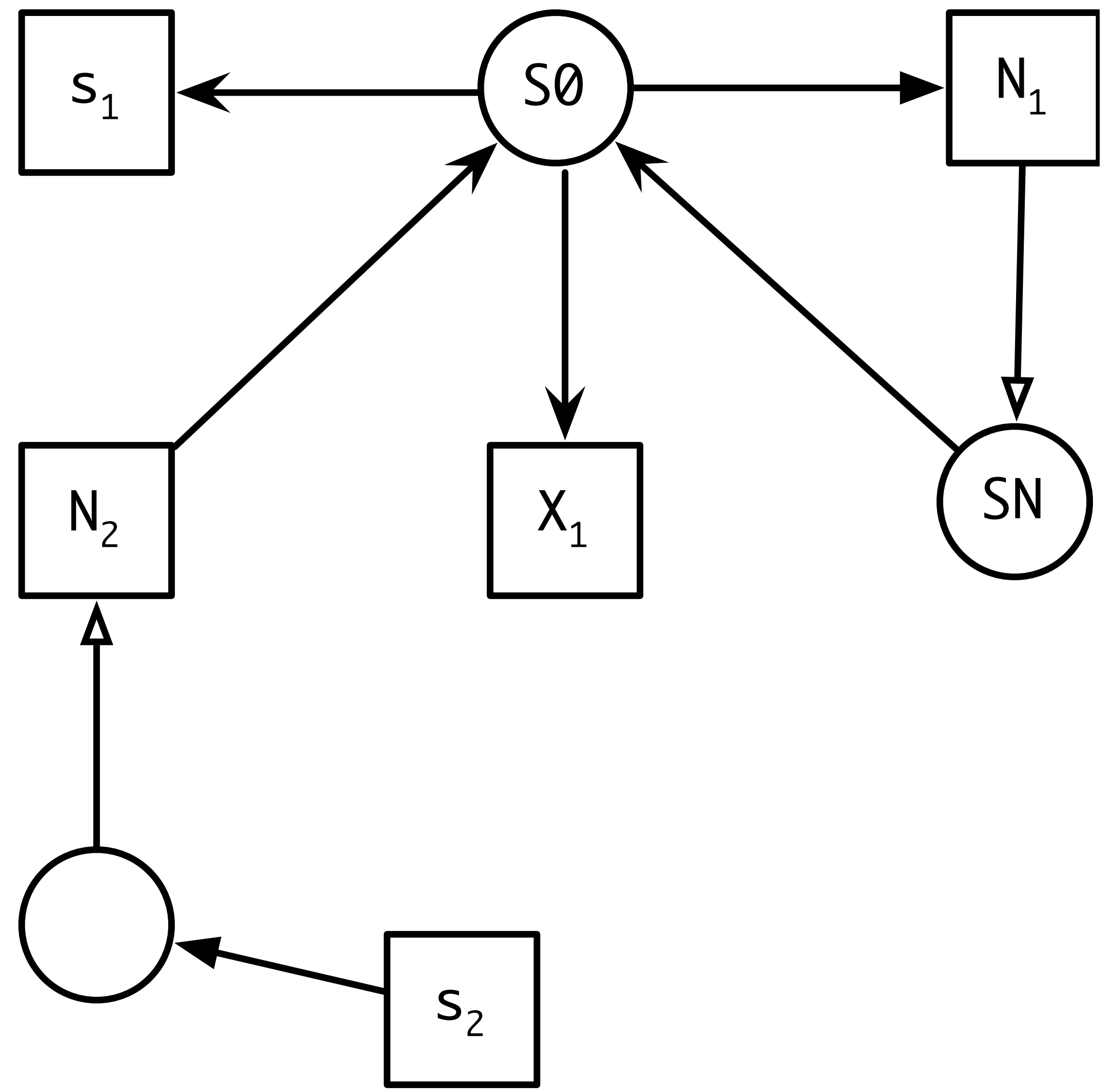
```
def s₁ = 5
module N₁ {
    SN
}

def x₁ = 1 + N₂.s₂
```

Import Parents

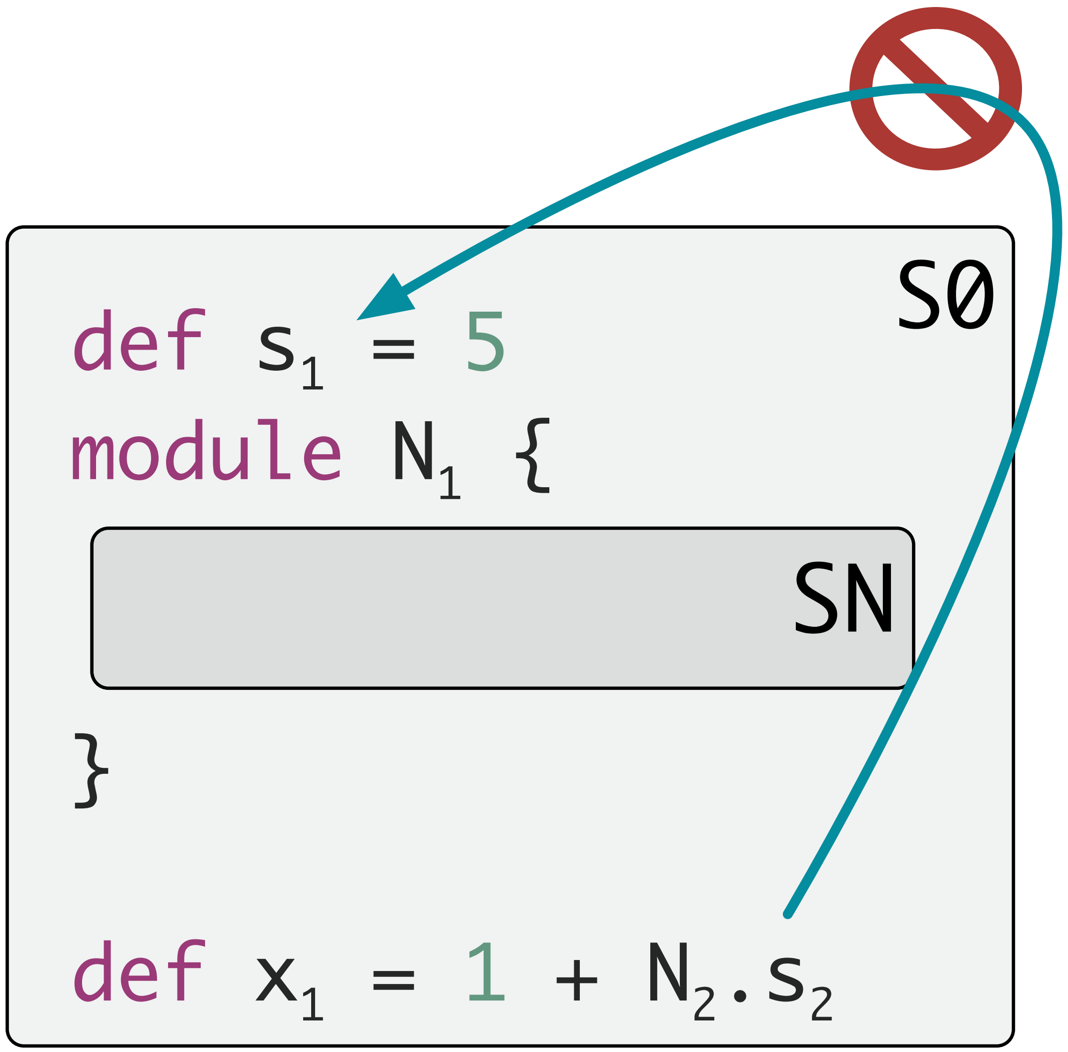


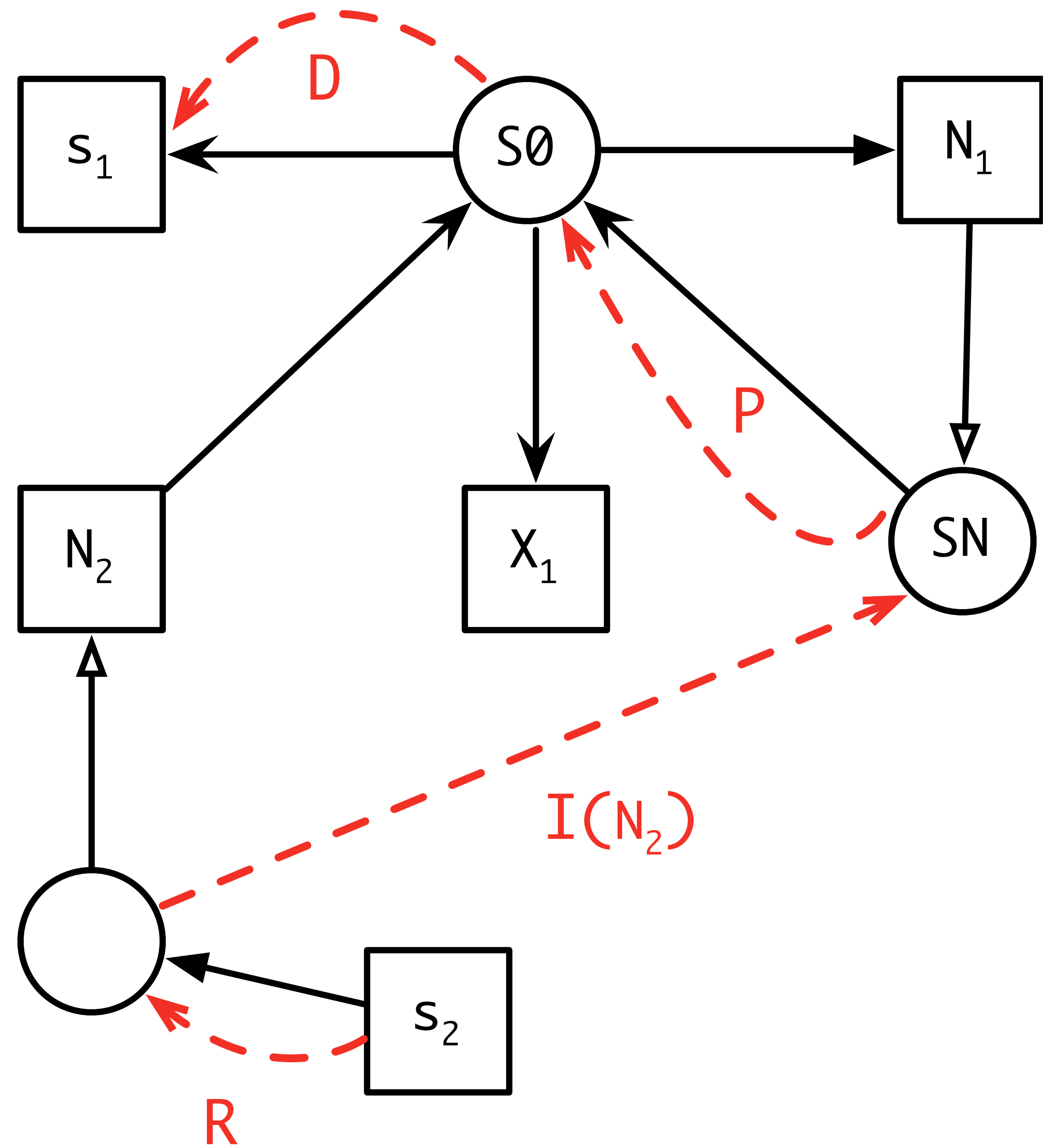
Import Parents



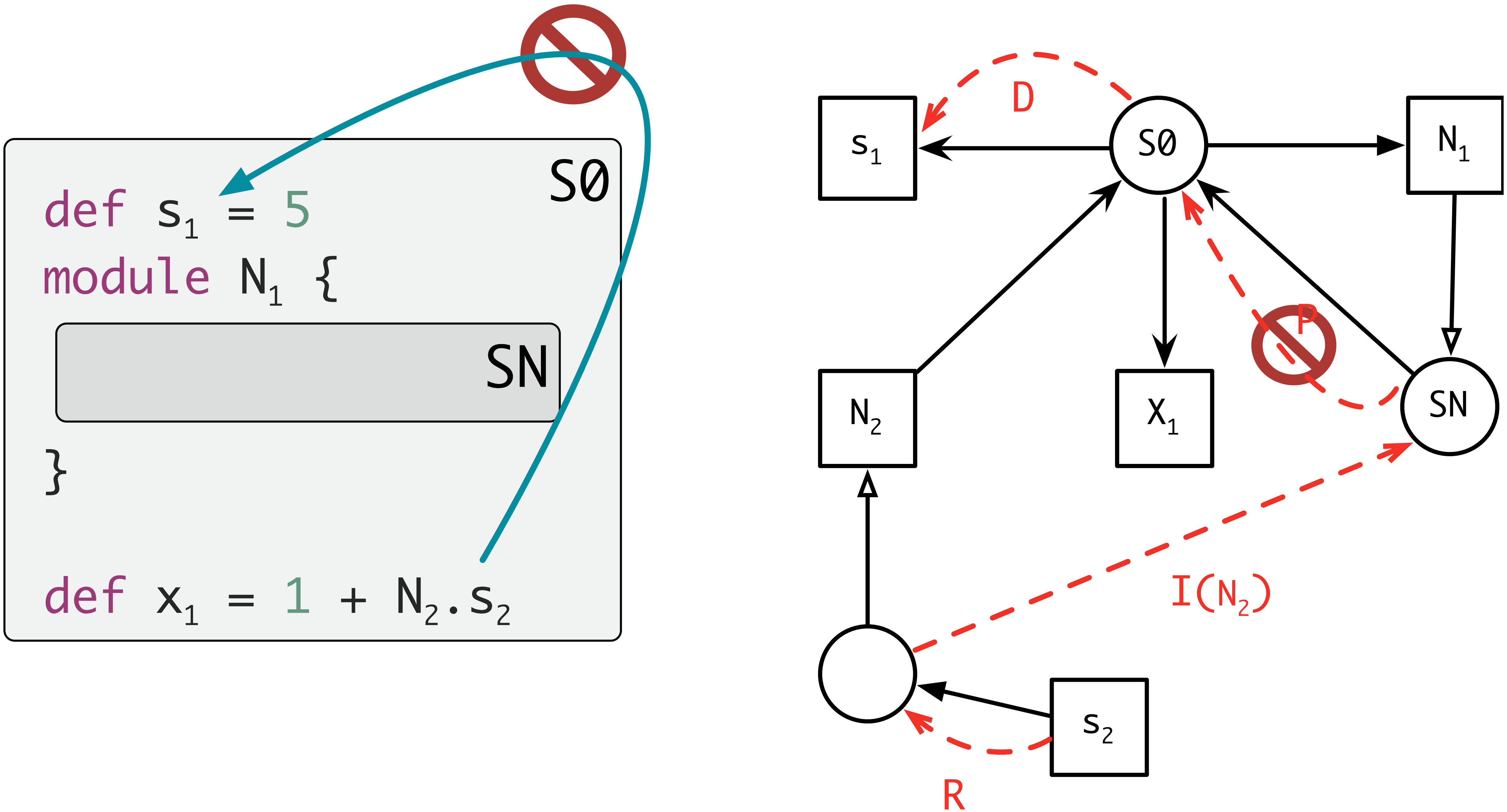


Import Parents

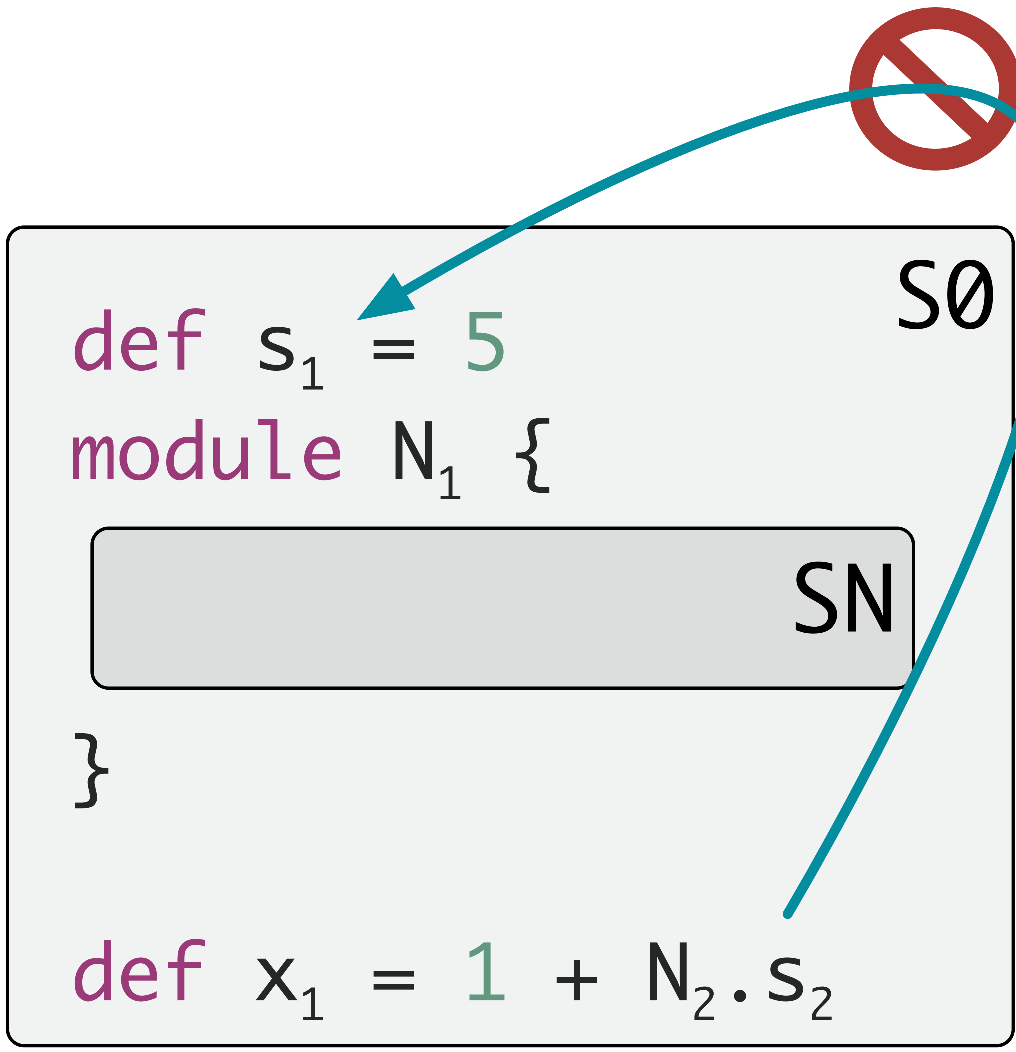


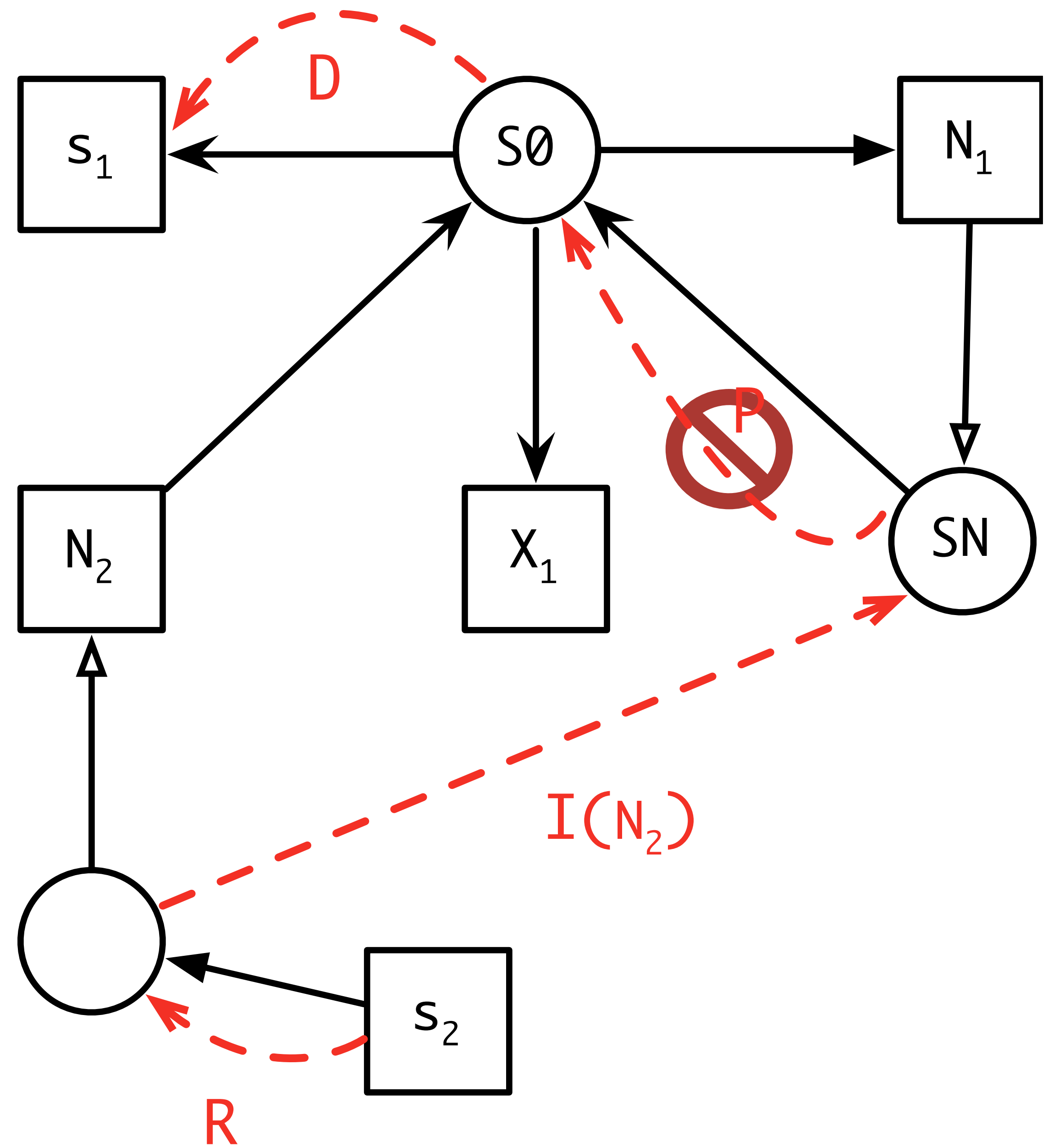


Import Parents



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

module A_1 {

  def z_1 = 5 SA

}

module B_1 {

  import A_2 SB

}

module C_1 {

  import B_2 SC

  def x_1 = 1 + z_2

}

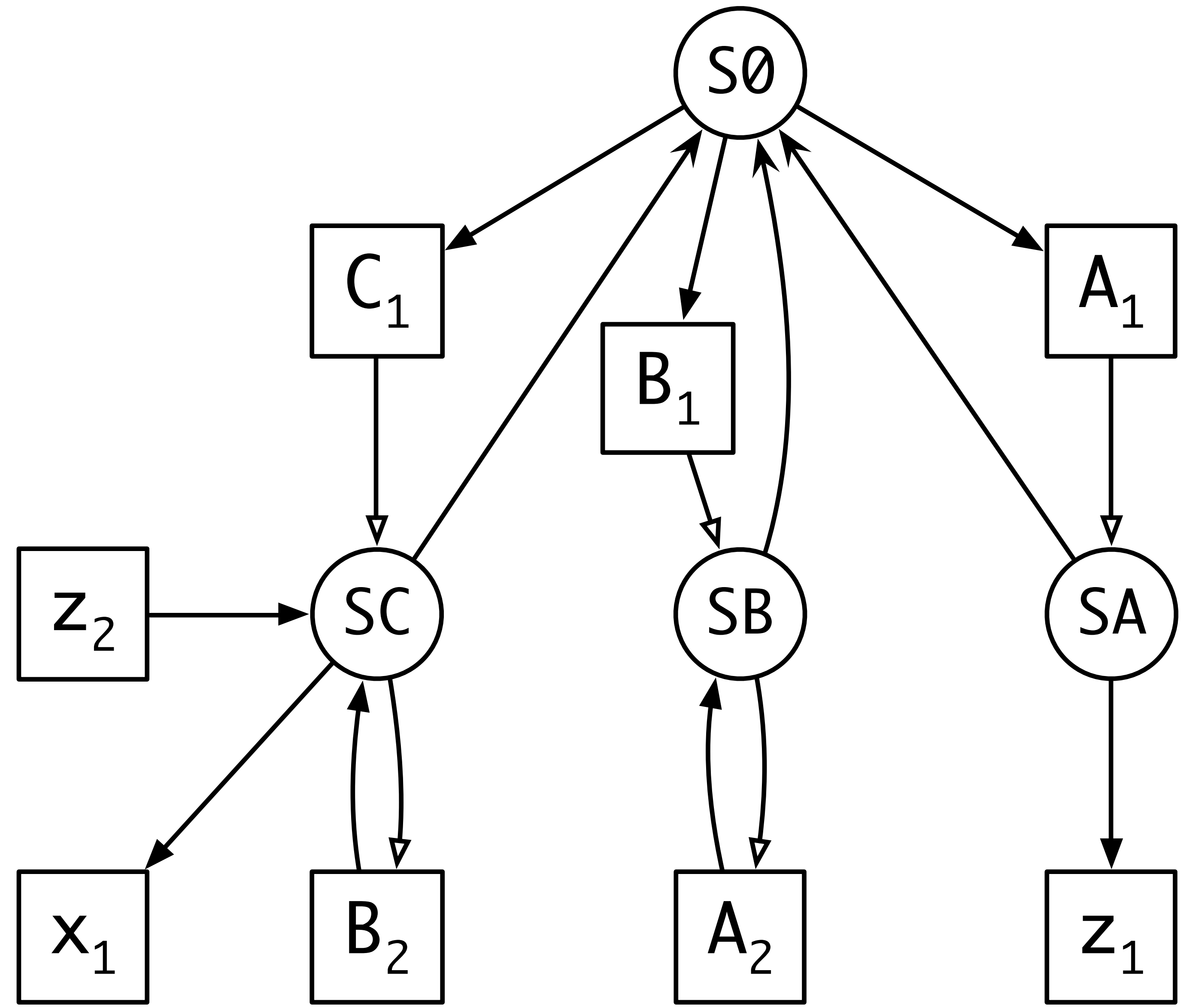
Transitive vs. Non-Transitive

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

??

Transitive vs. Non-Transitive

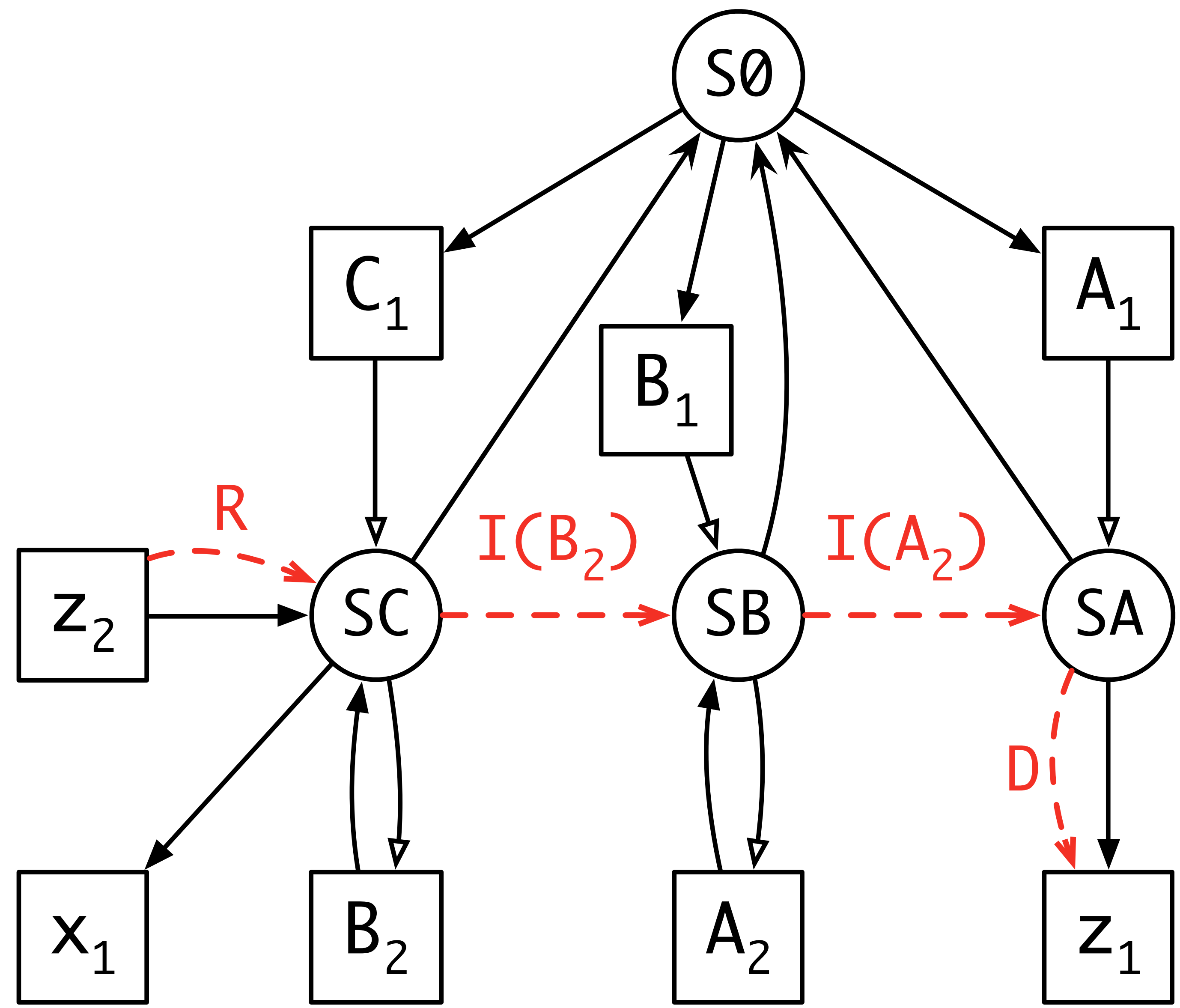
```
module A_1 {
    def z_1 = 5                  SA
}
                        ??
module B_1 {
    import A_2                   SB
}
module C_1 {
    import B_2                   SC
    def x_1 = 1 + z_2
}
```



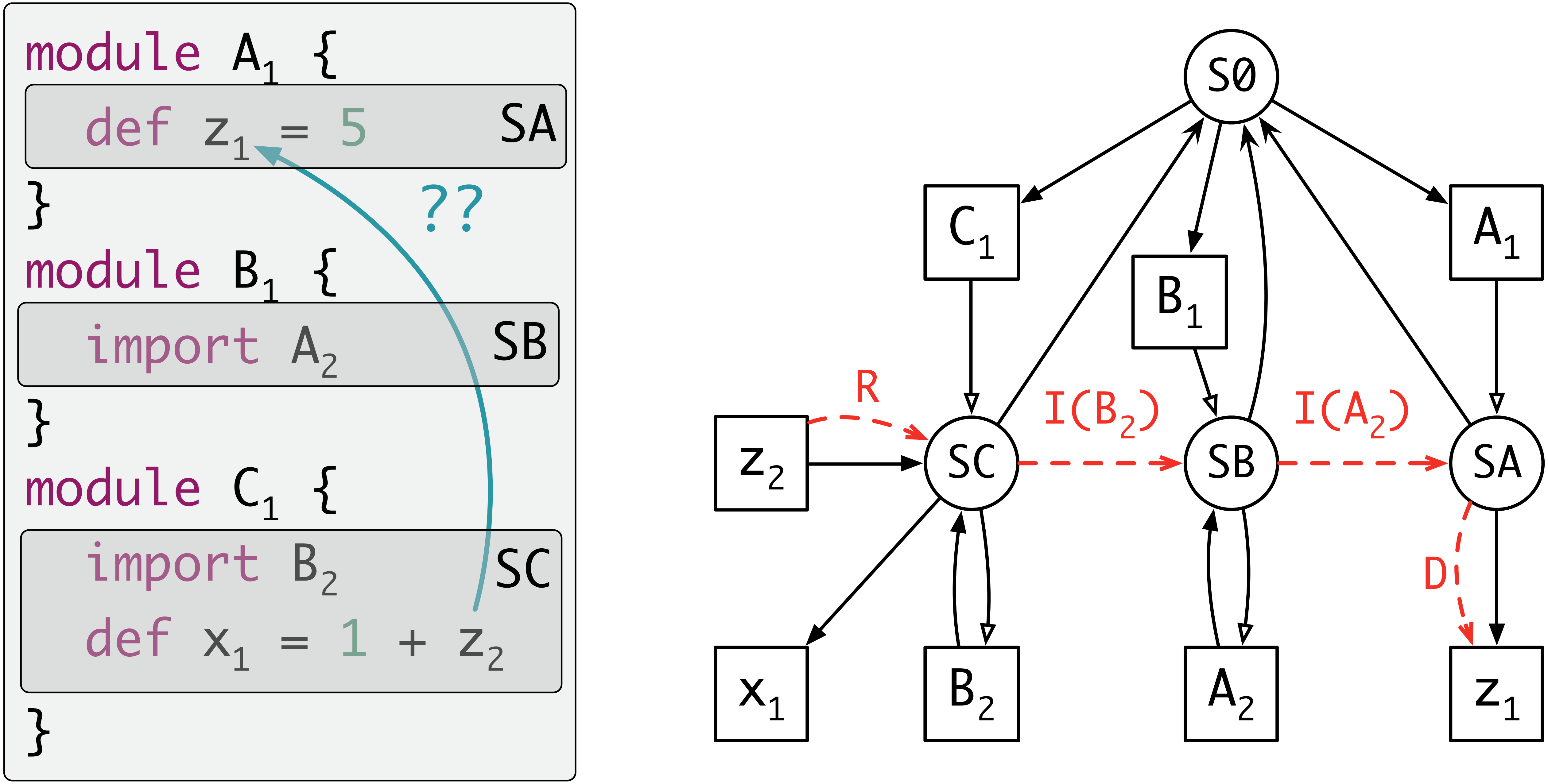
Transitive vs. Non-Transitive

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

??



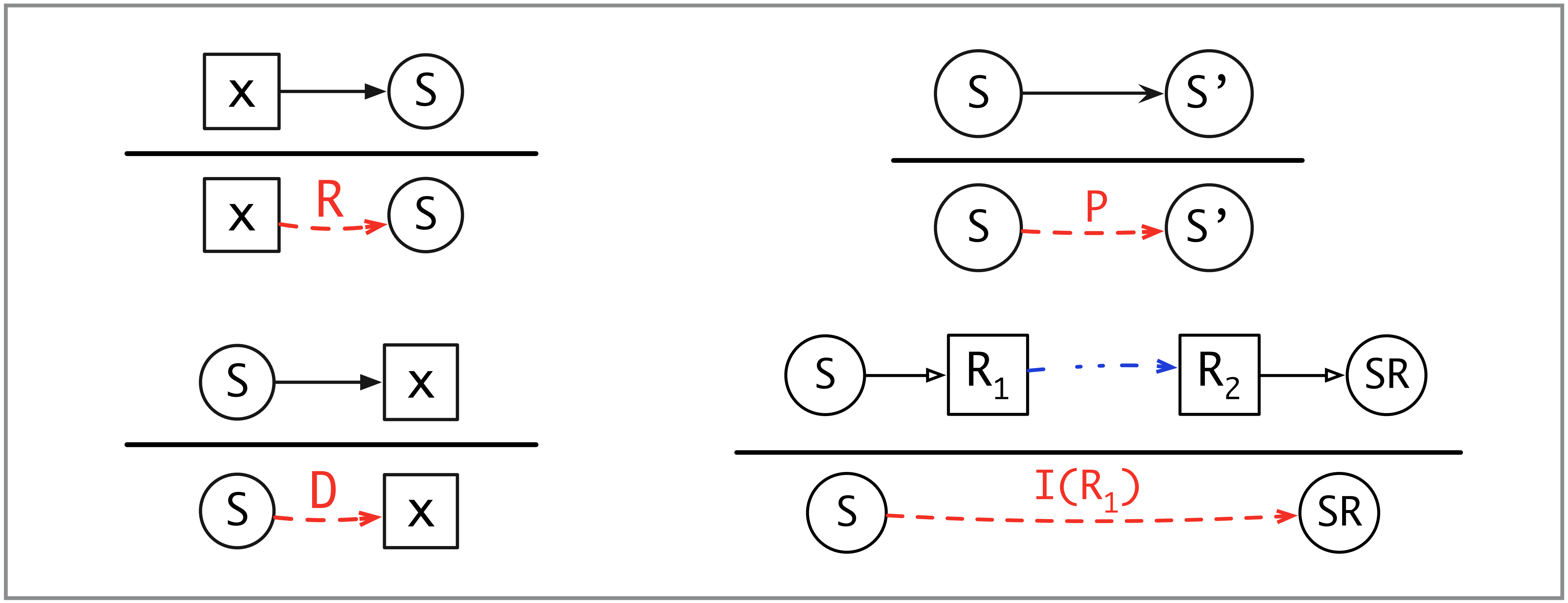
Transitive vs. Non-Transitive



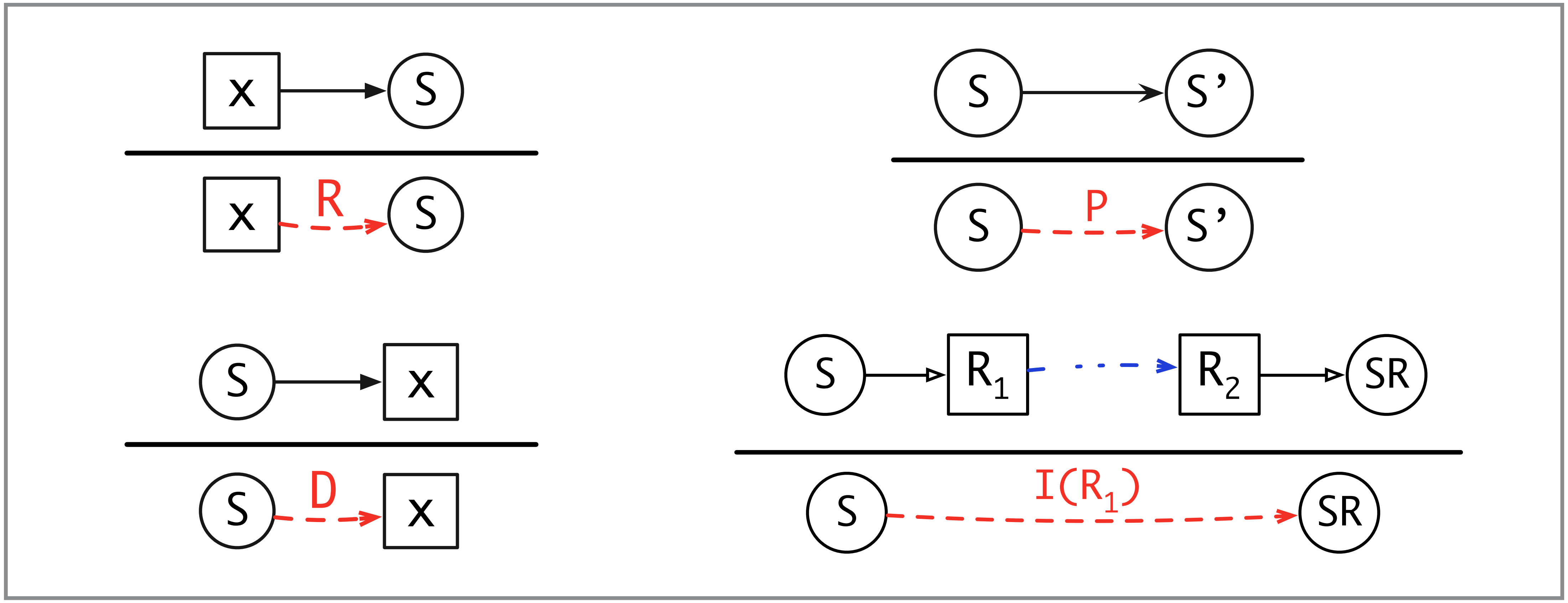
With transitive imports, a well formed path is `R.P*.I(_)*.D`

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

A Calculus for Name Resolution



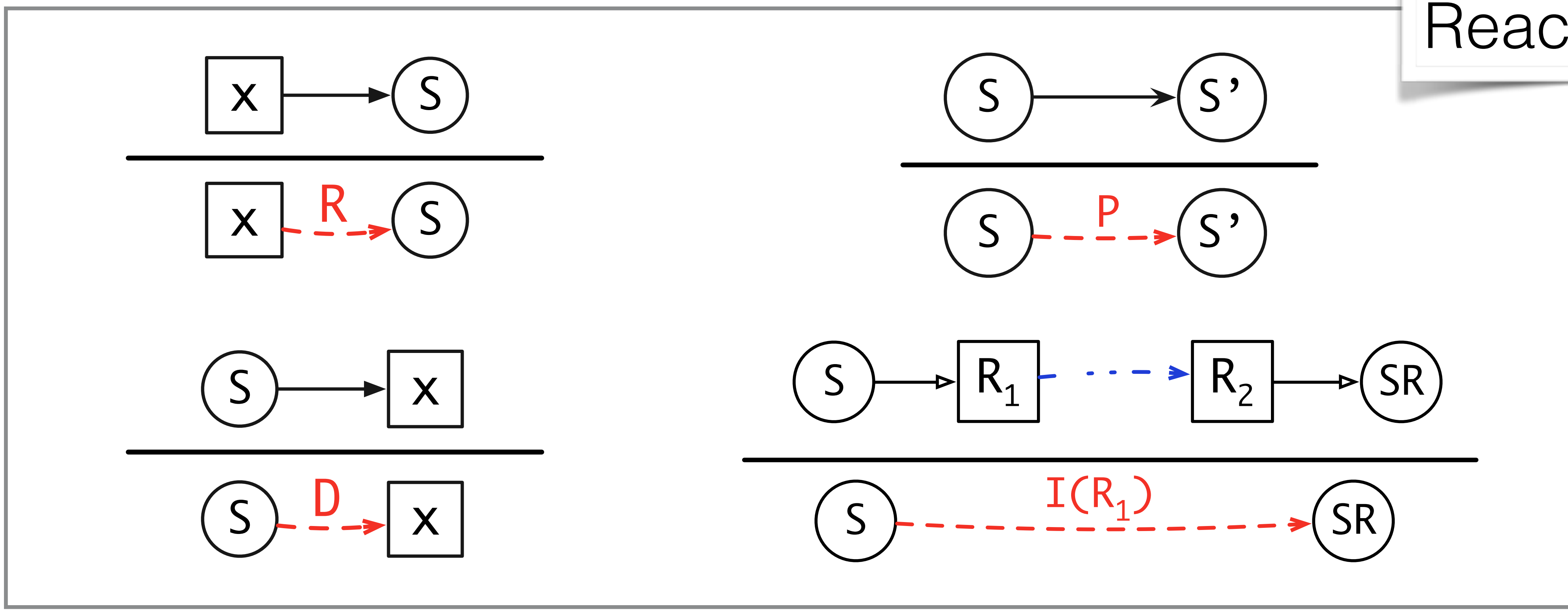
A Calculus for Name Resolution



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

A Calculus for Name Resolution

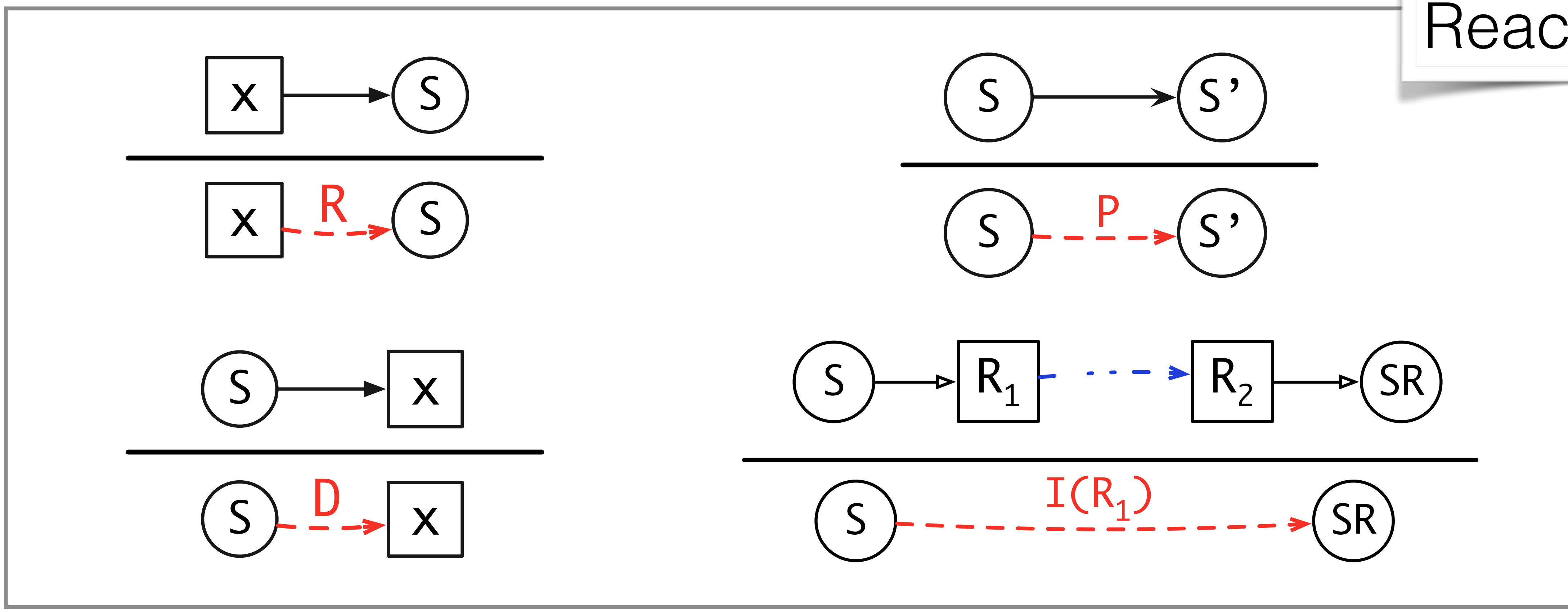
Reachability



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

A Calculus for Name Resolution

Reachability



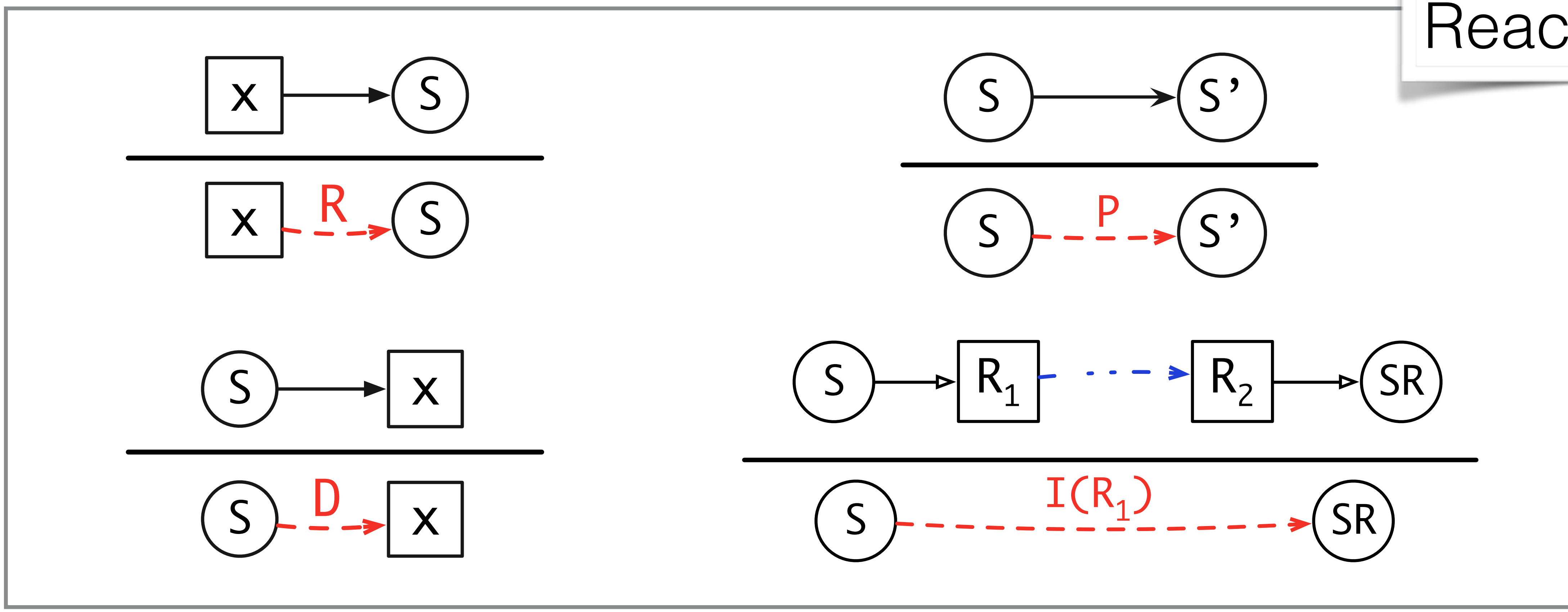
Well formed path: R.P*.I(_)*.D

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

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

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

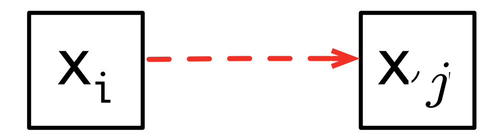
Resolution paths

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

Resolution calculus, formally

Resolution paths

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



Resolution calculus, formally

Resolution paths

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

$$\boxed{\mathsf{x}_i} \dashrightarrow \boxed{\mathsf{x}_{\prime j}}$$

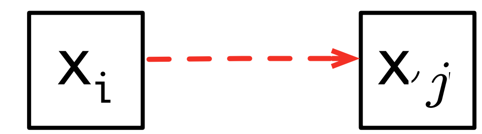
Visibility paths

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

Resolution calculus, formally

Resolution paths

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



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

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

$$\boxed{\mathsf{x_i}} \dashrightarrow \boxed{\mathsf{x}_{j}}$$

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

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

Resolution calculus, formally

Resolution paths

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

$$\boxed{x_i} \dashrightarrow \boxed{x_j}$$

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

$$
\begin{aligned}
E_D(S) &:= \mathcal{D}(S) \\
E_P(S) &:= E_V(\mathcal{P}(S)) \\
E_I(S) &:= \bigcup \left\{ E_L(S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \land y_j^{\mathsf{D}}{:}S_y \in \mathit{Resolve}(y_i^{\mathsf{R}}) \right\} \\
E_L(S) &:= E_D(S) \triangleleft E_I(S) \\
E_V(S) &:= E_L(S) \triangleleft E_P(S) \\
\mathit{Resolve}(x_i^{\mathsf{R}}) &:= \{ x_j^{\mathsf{D}} \mid \exists S \ s.t.\ x_i^{\mathsf{R}} \in \mathcal{R}(S) \land x_j^{\mathsf{D}} \in E_V(S) \}
\end{aligned}
$$

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

$$
\begin{aligned}
E_D(S) &:= \mathcal{D}(S) \\
E_P(S) &:= E_V(\mathcal{P}(S)) \\
E_I(S) &:= \bigcup \left\{ E_L(S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \land y_j^{\mathsf{D}}{:}S_y \in \mathit{Resolve}(y_i^{\mathsf{R}}) \right\} \\
E_L(S) &:= E_D(S) \triangleleft E_I(S) \\
E_V(S) &:= E_L(S) \triangleleft E_P(S) \\
\mathit{Resolve}(x_i^{\mathsf{R}}) &:= \{x_j^{\mathsf{D}} \mid \exists S \ s.t. \ x_i^{\mathsf{R}} \in \mathcal{R}(S) \land x_j^{\mathsf{D}} \in E_V(S)\}
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

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

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

Resolution Algorithm v3

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

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

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

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

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

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

Lemma: visibility paths never have cycles

Resolution Algorithm v3

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

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

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

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

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

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

Lemma: visibility paths never have cycles

Theorem: algorithm is sound and complete for calculus:

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

Language-independent α-equivalence

Program similarity

$\texttt{P} \simeq \texttt{P'}$ 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

$$\frac{\boxed{\texttt{x}_\texttt{i}} \dashrightarrow \boxed{\texttt{x}_{\texttt{i'}}}}{i \overset{\texttt{P}}{\sim} i'} \qquad \frac{i' \overset{\texttt{P}}{\sim} i}{i \overset{\texttt{P}}{\sim} i'} \qquad \frac{i \overset{\texttt{P}}{\sim} i' \quad i' \overset{\texttt{P}}{\sim} i''}{i \overset{\texttt{P}}{\sim} i''} \qquad \frac{}{i \overset{\texttt{P}}{\sim} i}$$

Language-independent α-equivalence

Program similarity

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

Position equivalence

$$\frac{\boxed{\texttt{x}_{\texttt{i}}} \dashrightarrow \boxed{\texttt{x}_{\texttt{i'}}}}{i \overset{\texttt{P}}{\sim} i'} \qquad \frac{i' \overset{\texttt{P}}{\sim} i}{i \overset{\texttt{P}}{\sim} i'} \qquad \frac{i \overset{\texttt{P}}{\sim} i' \quad i' \overset{\texttt{P}}{\sim} i''}{i \overset{\texttt{P}}{\sim} i''} \qquad \frac{}{i \overset{\texttt{P}}{\sim} i}$$

Alpha equivalence

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

(with some further details about free variables)

Preserving ambiguity

module $\mathtt{A}_{1}$ {
  def $\mathtt{x}_{2}$:= 1
}
module $\mathtt{B}_{3}$ {
  def $\mathtt{x}_{4}$:= 2
}
module $\mathtt{C}_{5}$ {
  import $\mathtt{A}_{6}$ $\mathtt{B}_{7}$;
  def $\mathtt{y}_{8}$:= $\mathtt{x}_{9}$
}
module $\mathtt{D}_{10}$ {
  import $\mathtt{A}_{11}$;
  def $\mathtt{y}_{12}$:= $\mathtt{x}_{13}$
}
module $\mathtt{E}_{14}$ {
  import $\mathtt{B}_{15}$;
  def $\mathtt{y}_{16}$:= $\mathtt{x}_{17}$
}

P1

module $\mathtt{AA}_{1}$ {
  def $\mathtt{z}_{2}$:= 1
}
module $\mathtt{BB}_{3}$ {
  def $\mathtt{z}_{4}$:= 2
}
module $\mathtt{C}_{5}$ {
  import $\mathtt{AA}_{6}$ $\mathtt{BB}_{7}$;
  def $\mathtt{s}_{8}$:= $\mathtt{z}_{9}$
}
module $\mathtt{D}_{10}$ {
  import $\mathtt{AA}_{11}$;
  def $\mathtt{u}_{12}$:= $\mathtt{z}_{13}$
}
module $\mathtt{E}_{14}$ {
  import $\mathtt{BB}_{15}$;
  def $\mathtt{v}_{16}$:= $\mathtt{z}_{17}$
}

P2

module $\mathtt{A}_{1}$ {
  def $\mathtt{z}_{2}$:= 1
}
module $\mathtt{B}_{3}$ {
  def $\mathtt{x}_{4}$:= 2
}
module $\mathtt{C}_{5}$ {
  import $\mathtt{A}_{6}$ $\mathtt{B}_{7}$;
  def $\mathtt{y}_{8}$:= $\mathtt{z}_{9}$
}
module $\mathtt{D}_{10}$ {
  import $\mathtt{A}_{11}$;
  def $\mathtt{y}_{12}$:= $\mathtt{z}_{13}$
}
module $\mathtt{E}_{14}$ {
  import $\mathtt{B}_{15}$;
  def $\mathtt{y}_{16}$:= $\mathtt{x}_{17}$
}

P3

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

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

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

```
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

```
def x : int = 6
def f = fun (y : int) { x + y }
```

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

$$\texttt{record } A_1 \ \{\ x_1 : \texttt{int}\ \}$$

$$\texttt{record } B_1 \ \{\ a_1 : A_2\ ;\ x_2 : \texttt{bool}\}$$

$$\texttt{def } z_1 : B_2 = \ldots$$

$$\texttt{def } y_1 = z_2.x_3$$

$$\texttt{def } y_2 = z_3.a_2.x_4$$

Types give Binding

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

$\texttt{record}\ A_1\ \{\ x_1 : \texttt{int}\ \}$

$\texttt{record}\ B_1\ \{\ a_1 : A_2\ ;\ x_2 : \texttt{bool}\}$

$\texttt{def}\ z_1 : B_2 = \ldots$

$\texttt{def}\ y_1 = z_2.x_3$

$\texttt{def}\ y_2 = z_3.a_2.x_4$

Types give Binding

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

$$\texttt{record}\ A_1\ \{\ x_1 : \texttt{int}\ \}$$
$$\texttt{record}\ B_1\ \{\ a_1 : A_2\ ;\ x_2 : \texttt{bool}\}$$

$$\texttt{def}\ z_1 : B_2 = \ ...$$

$$\texttt{def}\ y_1 = z_2.x_3$$

$$\texttt{def}\ y_2 = z_3.a_2.x_4$$

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

$$\texttt{record } A_1 \ \{\ x_1 : \texttt{int}\ \}$$

$$\texttt{record } B_1 \ \{\ a_1 : A_2 \ ;\ x_2 : \texttt{bool}\}$$

$$\texttt{def } z_1 : B_2 = \ldots$$

$$\texttt{def } y_1 = z_2.x_3$$

$$\texttt{def } y_2 = z_3.a_2.x_4$$

Types give Binding

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

```
record A_1 { x_1 : int }
record B_1 { a_1 : A_2 ; x_2 : bool}

def z_1 : B_2 = ...
def y_1 = z_2.x_3
def y_2 = z_3.a_2.x_4
```

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

$$
\begin{array}{l}
\texttt{record}\ A_1\ \{\ x_1 : \texttt{int}\ \} \\
\texttt{record}\ B_1\ \{\ a_1 : A_2\ ;\ x_2 : \texttt{bool}\ \} \\
\\
\texttt{def}\ z_1 : B_2 = \ldots \\
\texttt{def}\ y_1 = z_2.x_3 \\
\texttt{def}\ y_2 = z_3.a_2.x_4
\end{array}
$$

Types give Binding

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

$\texttt{record}\ A_1\ \{\ x_1 : \texttt{int}\ \}$

$\texttt{record}\ B_1\ \{\ a_1 : A_2\ ;\ x_2 : \texttt{bool}\}$

$\texttt{def}\ z_1 : B_2 = \ldots$

$\texttt{def}\ y_1 = z_2.x_3$

$\texttt{def}\ y_2 = z_3.a_2.x_4$

Types give Binding

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

$\texttt{record}\ A_1\ \{\ x_1 : \texttt{int}\ \}$

$\texttt{record}\ B_1\ \{\ a_1 : A_2\ ;\ x_2 : \texttt{bool}\}$

$\texttt{def}\ z_1 : B_2 = \ldots$

$\texttt{def}\ y_1 = z_2.x_3$

$\texttt{def}\ y_2 = z_3.a_2.x_4$

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

Also in the works...

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

Use Scope Graphs to Describe Name Resolution

Use Scope Graphs to Describe Name Resolution !!