

Lecture 6(b): Name Resolution

Eelco Visser

CS4200 Compiler Construction

TU Delft

October 2019

Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

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

Best EAPLS paper at ETAPS 2015

ESOP 2015

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

A Theory of Name Resolution

Pierre Neron¹, Andrew Tolmach², Eelco Visser¹, and Guido Wachsmuth¹

¹) Delft University of Technology, The Netherlands,
{p.j.m.neron, e.visser, g.wachsmuth}@tudelft.nl,
²) Portland State University, Portland, OR, USA
tolmach@pdx.edu

Abstract. We describe a language-independent theory for name binding and resolution, suitable for programming languages with complex scoping rules including both lexical scoping and modules. We formulate name resolution as a two-stage problem. First a language-independent scope graph is constructed using language-specific rules from an abstract syntax tree. Then references in the scope graph are resolved to corresponding declarations using a language-independent resolution process. We introduce a resolution calculus as a concise, declarative, and language-independent specification of name resolution. We develop a resolution algorithm that is sound and complete with respect to the calculus. Based on the resolution calculus we develop language-independent definitions of α -equivalence and rename refactoring. We illustrate the approach using a small example language with modules. In addition, we show how our approach provides a model for a range of name binding patterns in existing languages.

1 Introduction

Naming is a pervasive concern in the design and implementation of programming languages. Names identify *declarations* of program entities (variables, functions, types, modules, etc.) and allow these entities to be *referenced* from other parts of the program. Name *resolution* associates each reference to its intended declaration(s), according to the semantics of the language. Name resolution underlies most operations on languages and programs, including static checking, translation, mechanized description of semantics, and provision of editor services in IDEs. Resolution is often complicated, because it cuts across the local inductive structure of programs (as described by an abstract syntax tree). For example, the name introduced by a **let** node in an ML AST may be referenced by an arbitrarily distant child node. Languages with explicit name spaces lead to further complexity; for example, resolving a qualified reference in Java requires first resolving the class or package name to a context, and then resolving the member name within that context. But despite this diversity, it is intuitively clear that the basic concepts of resolution reappear in similar form across a broad range of lexically-scoped languages.

In practice, the name resolution rules of real programming languages are usually described using *ad hoc* and informal mechanisms. Even when a language *is* formalized, its resolution rules are typically encoded as part of static

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

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

PEPM 2016

<https://doi.org/10.1145/2847538.2847543>

A Constraint Language for Static Semantic Analysis based on Scope Graphs

Hendrik van Antwerpen
Delft University of Technology
h.vanantwerpen@student.tudelft.nl

Pierre Néron
Delft University of Technology
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University
tolmach@pdx.edu

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Abstract

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

In recent work we show how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition

of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one, none, or many declarations. A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

In this paper, we refine and extend the scope graph framework of [14] to construct a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checking step—would work. But the full story is more complicated, because sometimes name resolution also depends on type resolution. For example, in a language that uses dot notation for object field projection, determining the resolution of x in the expression $r.x$ requires first determining the object type of r , which in turn requires name resolution again. Thus, our framework requires a unified mechanism for expressing and solving arbitrarily interdependent naming and typing resolution problems.

To address this challenge, we base our framework on a language of *constraints*. Term equality constraints are a standard choice for describing type inference problems while abstracting away from the details of an AST in a particular language. Adopting constraints to describe both typing and scoping requirements has the advantage of uniform notation, and, more importantly, provides a clean way to combine naming and typing problems. In particular, we extend our previous work to support *incomplete* scope graphs, which correspond to constraint sets with (as yet) unresolved variables.

Our new framework continues to satisfy the criteria outlined above. (i) The resolution calculus and standard term equality constraint theory provide a solid language-independent theory for name and type resolution. (ii) Our framework supports type checking and inference for statically typed, monomorphic languages with user-defined types, and can also express uniqueness and completeness requirements on declarations and initializers. The framework inherits from scope graphs the ability to model a broad range of binding patterns, including many variants of lexical scoping, records, and modules. (iii) The constraint language has a declarative semantics given by a constraint satisfaction relation, which employs the resolution calculus to define the semantics of name resolution relative to a scope graph. We define a constraint resolution algorithm based on our previous name resolution algorithm,

This paper describes the next generation of the approach.

Addresses (previously) open issues in expressiveness of scope graphs for type systems:

- Structural types
- Generic types

Addresses open issue with staging of information in type systems.

Introduces Statix DSL for definition of type systems.

Prototype of Statix is available in Spoofax HEAD, but not ready for use in project yet.

OOPSLA 2018

To appear

Scopes as Types

HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands
CASPER BACH POULSEN, Delft University of Technology, Netherlands
ARJEN ROUVOET, Delft University of Technology, Netherlands
EELCO VISSER, Delft University of Technology, Netherlands

Scope graphs are a promising generic framework to model the binding structures of programming languages, bridging formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs. However, previous work on scope graphs has been limited to simple, nominal type systems. In this paper, we show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries. We introduce Statix, a new domain-specific meta-language for the specification of static semantics, based on scope graphs and constraints. We evaluate the scopes as types approach and the Statix design in case studies of the simply-typed lambda calculus with records, System F, and Featherweight Generic Java.

CCS Concepts: • **Software and its engineering** → **Semantics; Domain specific languages**;

Additional Key Words and Phrases: static semantics, type system, type checker, name resolution, scope graphs, domain-specific language

ACM Reference Format:

Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (November 2018), 30 pages. <https://doi.org/10.1145/3276484>

1 INTRODUCTION

The goal of our work is to support high-level specification of type systems that can be used for multiple purposes, including reasoning (about type safety among other things) and the implementation of type checkers [Visser et al. 2014]. Traditional approaches to type system specification do not reflect the commonality underlying the name binding mechanisms for different languages. Furthermore, operationalizing name binding in a type checker requires carefully staging the traversals of the abstract syntax tree in order to collect information before it is needed. In this paper, we introduce an approach to the declarative specification of type systems that is close in abstraction to traditional type system specifications, but can be directly interpreted as type checking rules. The approach is based on scope graphs for name resolution, and constraints to separate traversal order from solving order.


Authors’ addresses: Hendrik van Antwerpen, Delft University of Technology, Delft, Netherlands, H.vanAntwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, Netherlands, C.B.Poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, Delft, Netherlands, A.J.Rouvoet@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, Netherlands, E.Visser@tudelft.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART114
<https://doi.org/10.1145/3276484>

Documentation for NaBL2 at the [metaborg.org](http://www.metaborg.org) website.

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

 Spoofax

latest

The Spoofax Language Workbench

Examples

Publications

TUTORIALS

Installing Spoofax

Creating a Language Project

Using the API

Getting Support

REFERENCE MANUAL

Language Definition with Spoofax

Abstract Syntax with ATerms

Syntax Definition with SDF3

Static Semantics with NaBL2

1. Introduction

2. Language Reference

3. Configuration

4. Examples

5. Bibliography

6. NaBL/TS (Deprecated)

Transformation with Stratego

Dynamic Semantics with DynSem

Editor Services with ESV

Language Testing with SPT

Building Languages

Programmatic API


Developing Spoofax

Development Release

Release Archive

Migration Guides

Docs » Static Semantics Definition with NaBL2

 [Edit on GitHub](#)

Static Semantics Definition with NaBL2

Programs that are syntactically well-formed are not necessarily valid programs. Programming languages typically impose additional *context-sensitive* requirements on programs that cannot be captured in a syntax definition. Languages use names to identify reusable units that can be invoked at multiple parts in a program. In addition, statically typed languages require that expressions are consistently typed. The NaBL2 ‘Name Binding Language’ supports the specification of name binding and type checking rules of a language. NaBL2 uses a constraint-based approach, and uses scope graphs for name resolution.

Table of Contents

- 1. Introduction
 - 1.1. Name Resolution with Scope Graphs
- 2. Language Reference
 - 2.1. Lexical matters
 - 2.2. Modules
 - 2.3. Signatures
 - 2.4. Rules
 - 2.5. Constraints
- 3. Configuration
 - 3.1. Prepare your project
 - 3.2. Runtime settings
 - 3.3. Customize analysis
 - 3.4. Inspecting analysis results
- 4. Examples
- 5. Bibliography
- 6. NaBL/TS (Deprecated)
 - 6.1. Namespaces
 - 6.2. Name Binding Rules
 - 6.3. Interaction with Type System

Note

The predecessor of NaBL2, the NaBL/TS name binding and type analysis meta-language is deprecated.

6

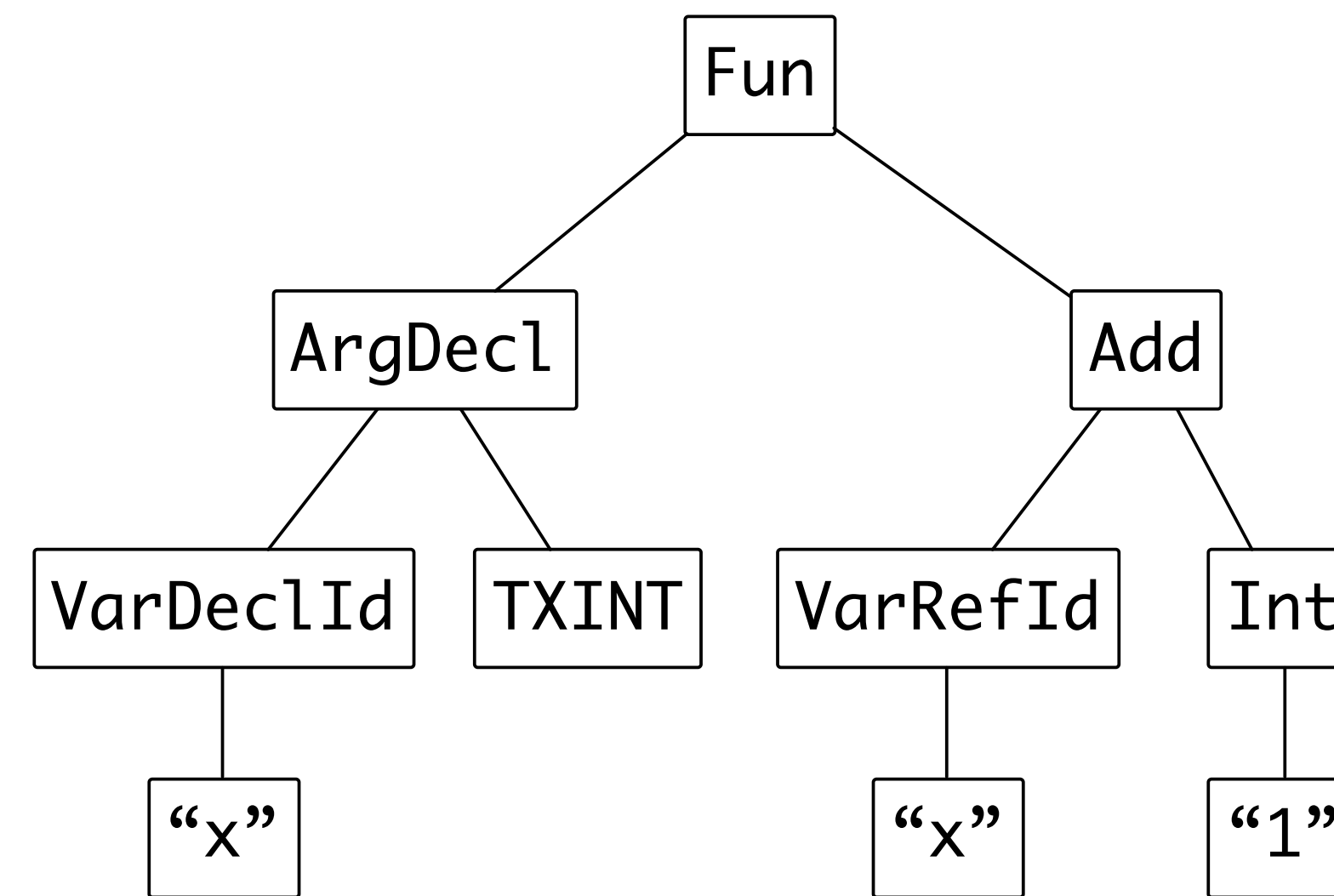
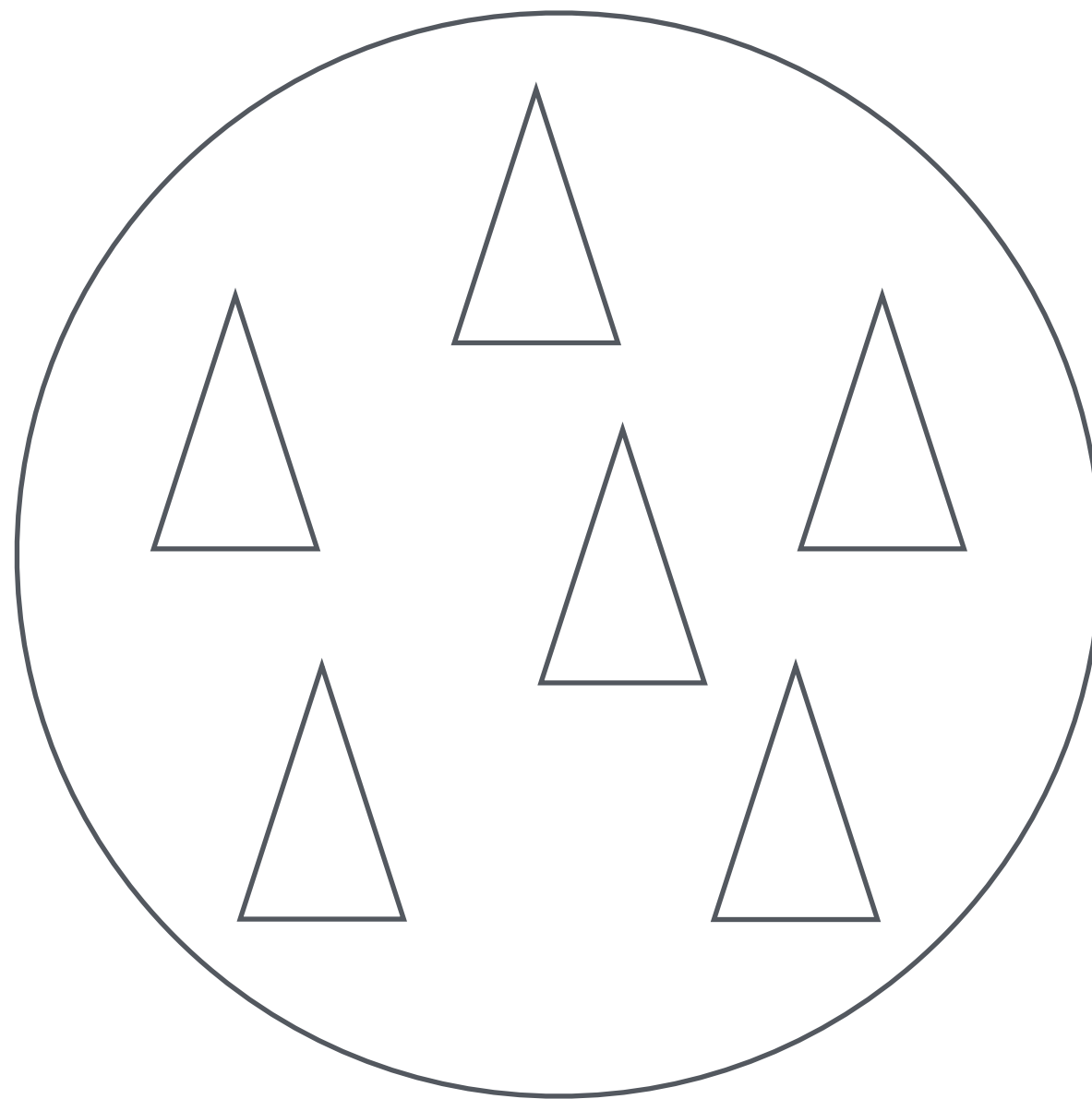
Documentation for Statix at the metaborg.org website.

?

<http://www.metaborg.org/en/latest/source/langdev/meta/lang/statix/index.html>

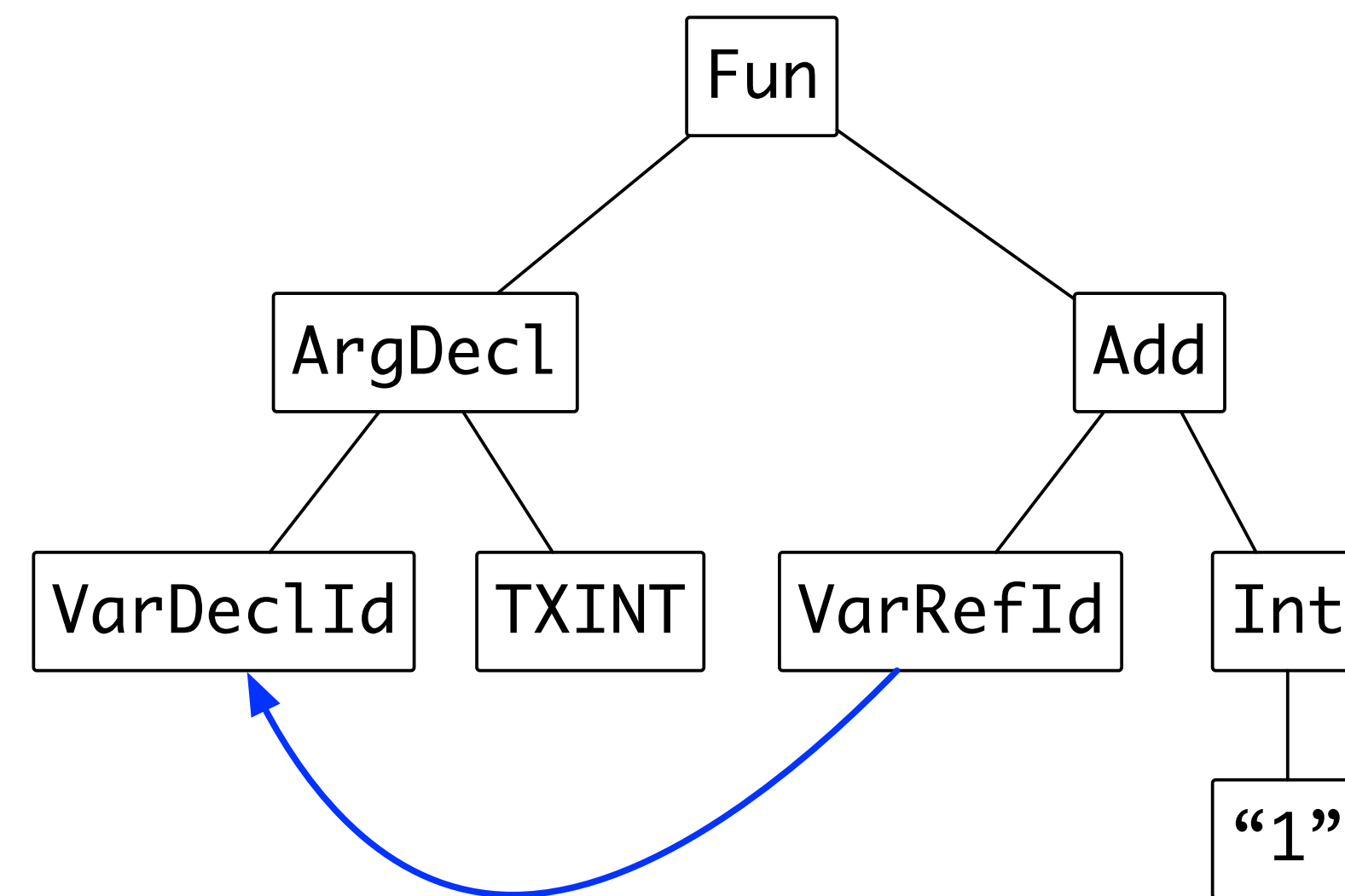
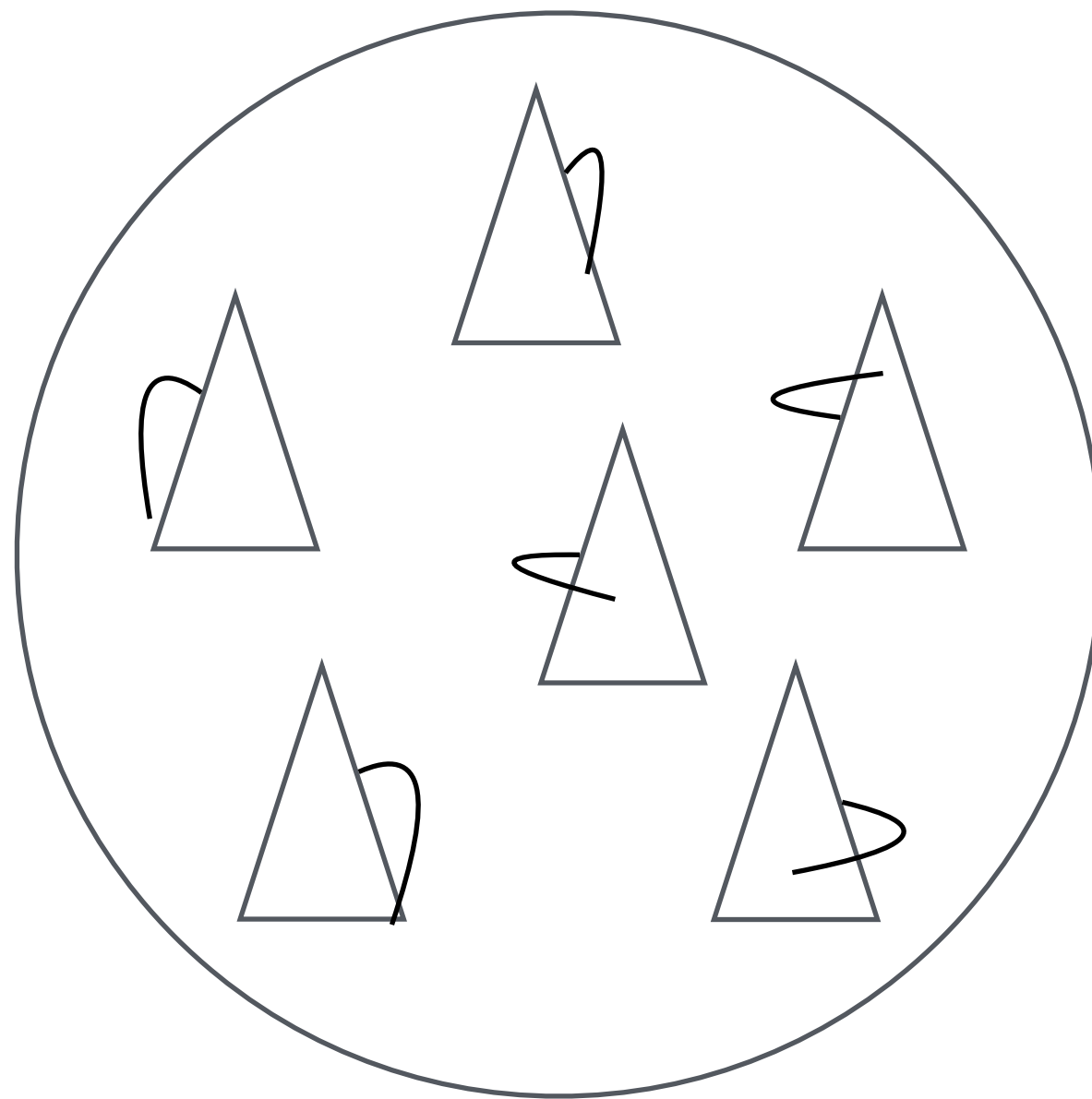
Name Binding

Language = Set of Trees



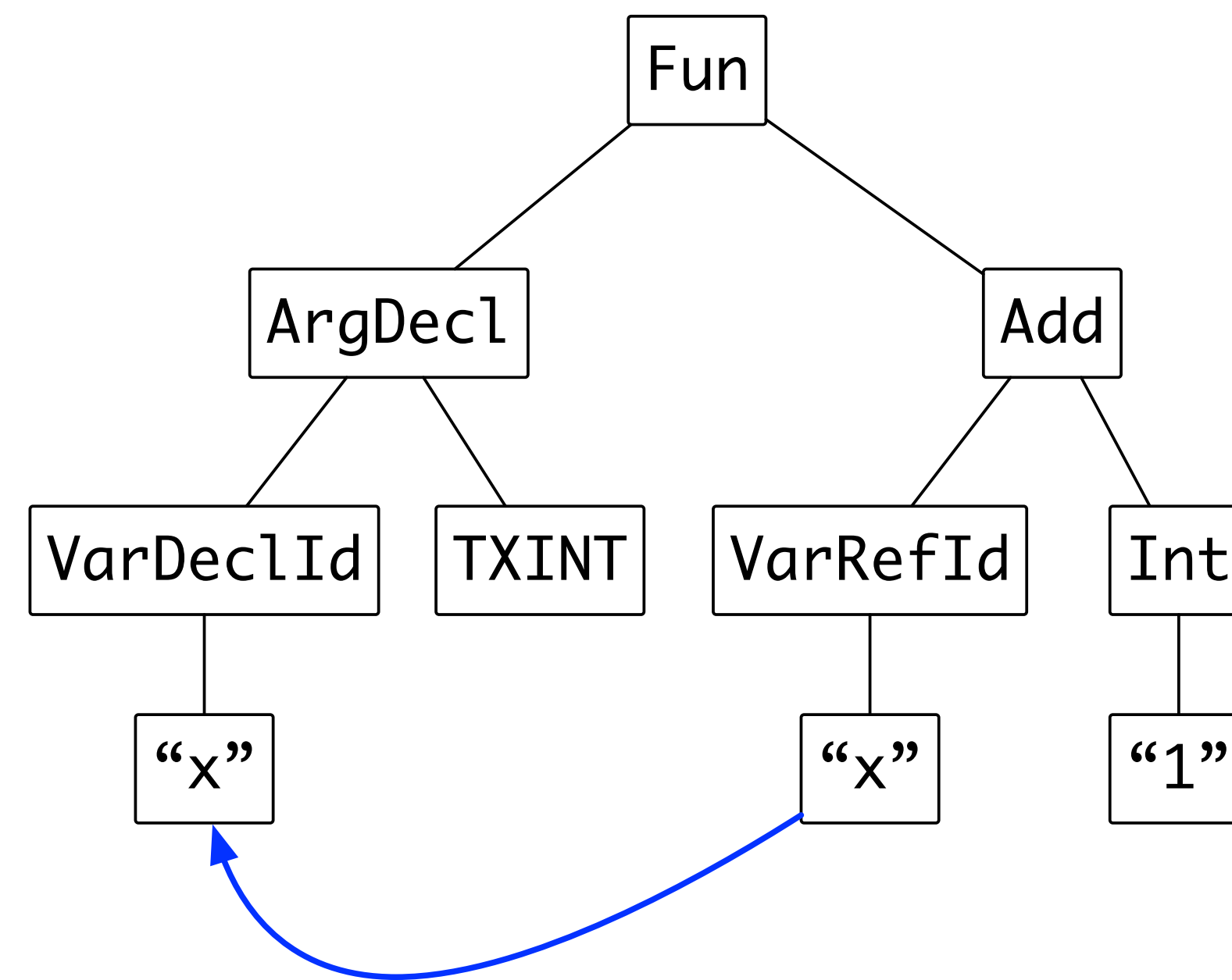
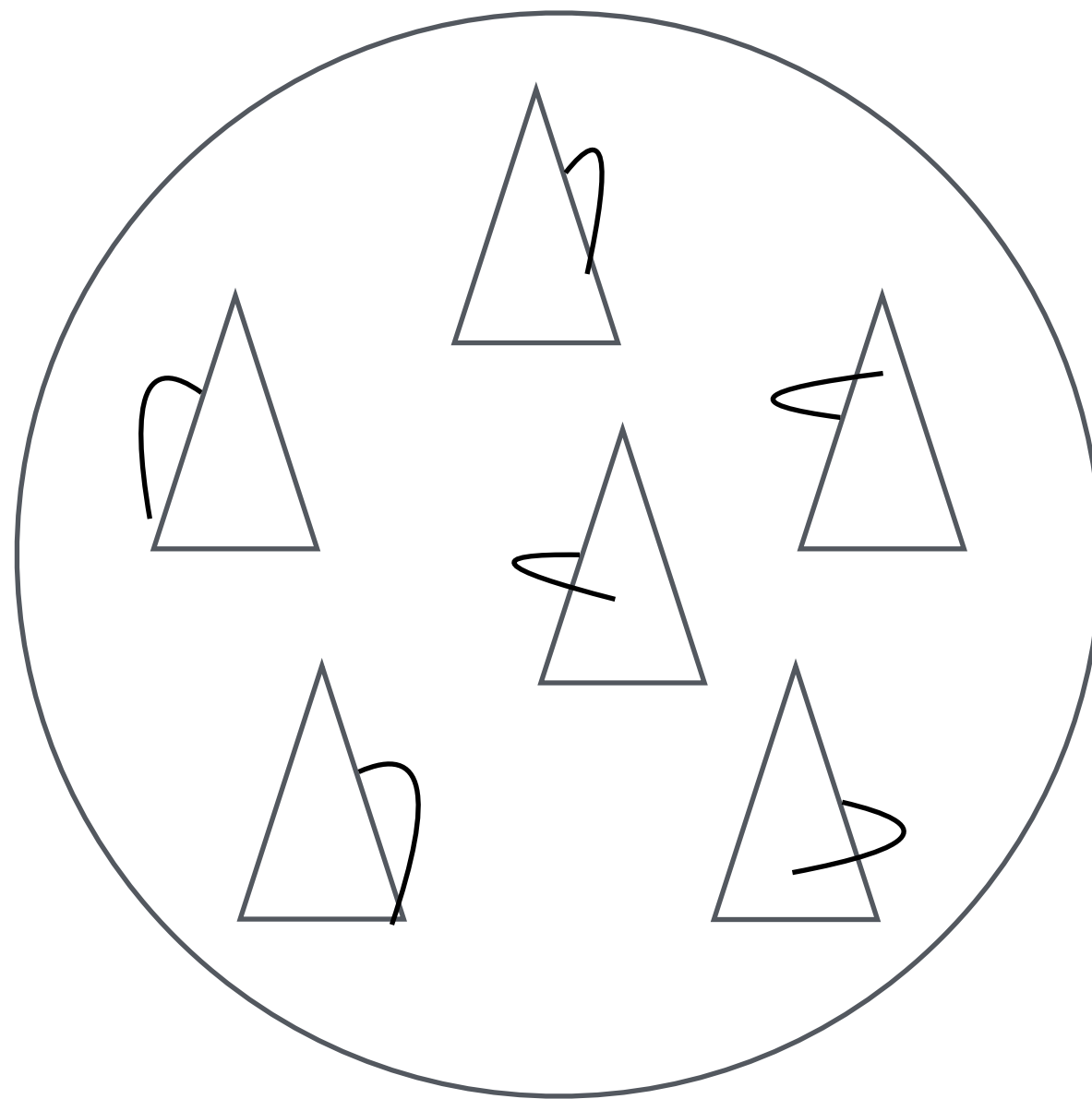
Tree is a convenient interface for transforming programs

Language = Set of *Graphs*



Edges from references to declarations

Language = Set of *Graphs*




Names are placeholders for edges in linear / tree representation

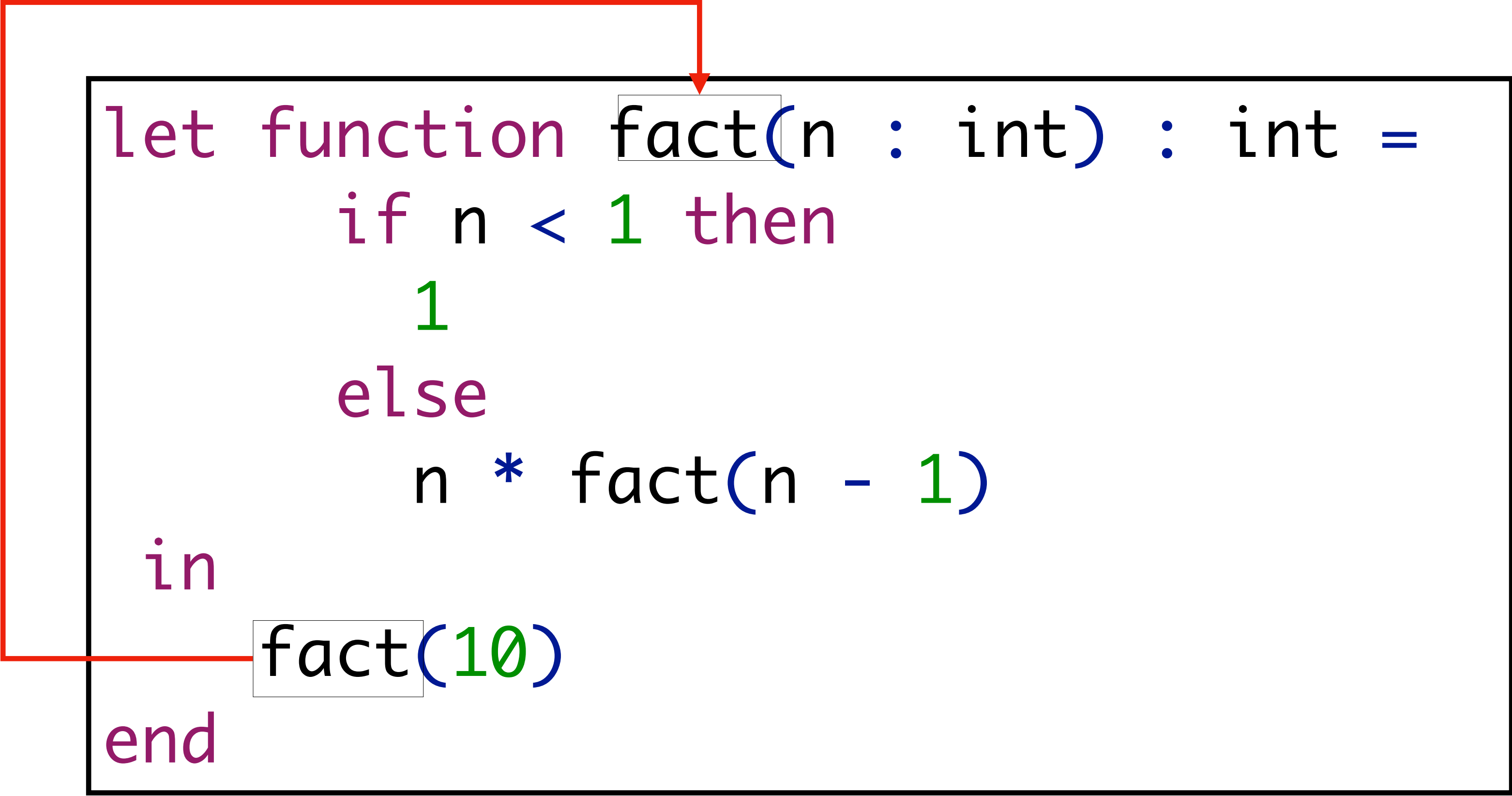
Name Binding Patterns

Variables

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

A red line with arrows illustrates the variable binding and recursive call. It starts from the 'n' in the function signature 'fact(n : int)', goes down to the 'n' in the recursive call 'fact(n - 1)', and then loops back up to the 'n' in the conditional 'if n < 1'. This indicates that the same variable 'n' is used for both the function parameter and the recursive argument.

Function Calls



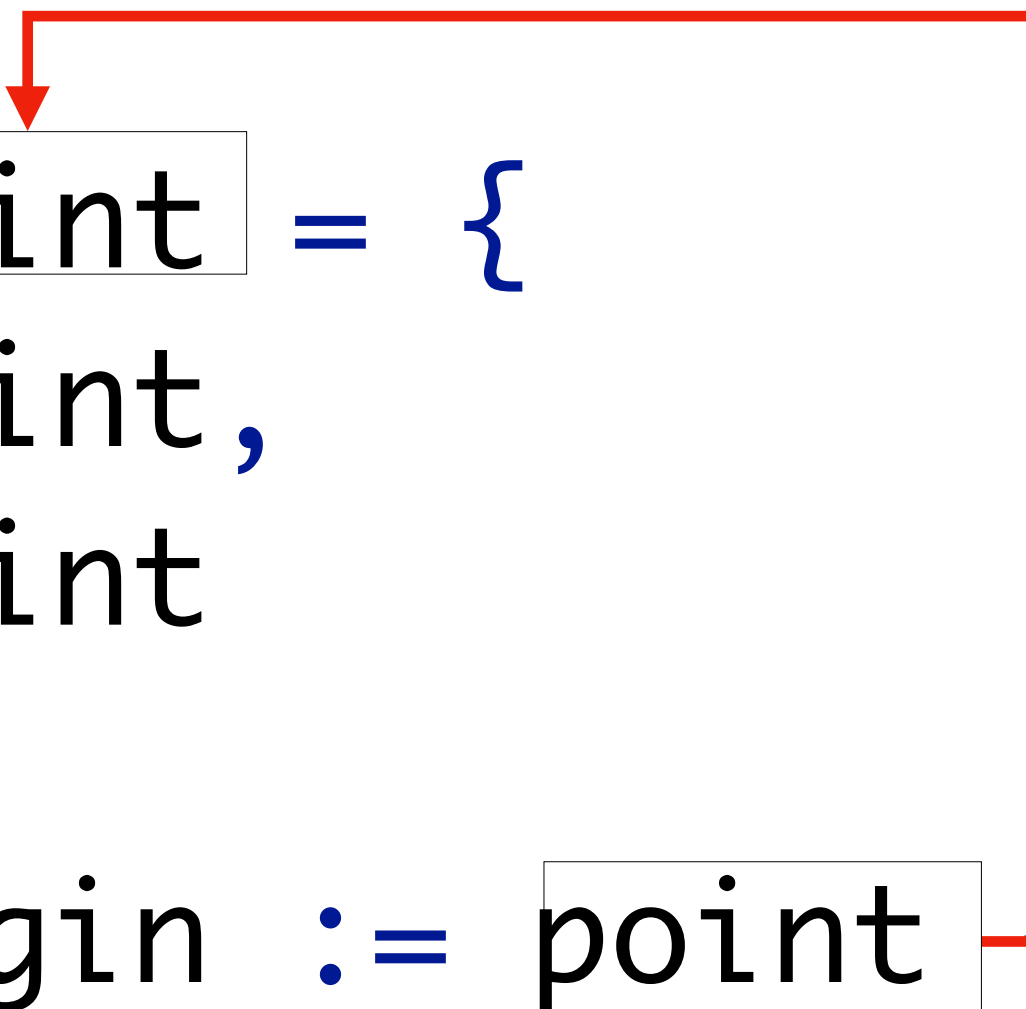
```
let function fact(n : int) : int =  
    if n < 1 then  
        1  
    else  
        n * fact(n - 1)  
    in  
    fact(10)  
end
```

The diagram illustrates a function call. A red line originates from the argument '10' in the function call 'fact(10)' and extends upwards and to the right, ending in an arrow that points to the parameter 'n' in the function definition 'fact(n : int)'. This visualizes the mapping of the argument to the parameter during a function call.

Nested Scopes (Shadowing)

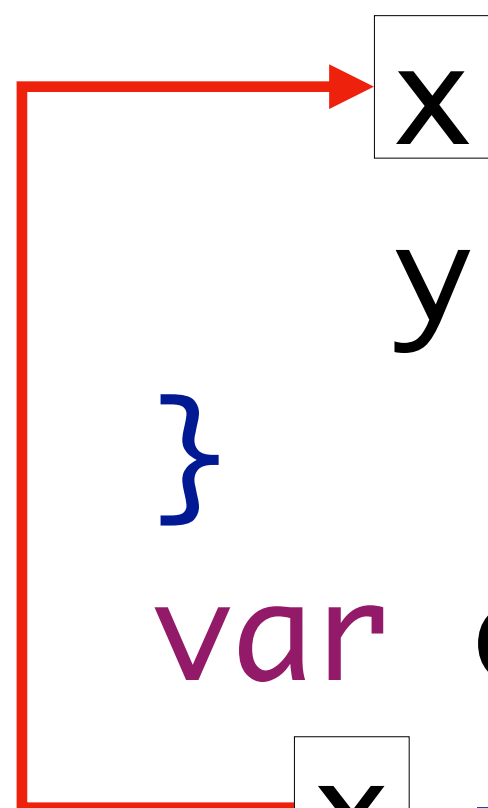
```
function prettyprint(tree: tree) : string =  
  let  
    var output := ""  
  
    function write(s: string) =  
      output := concat(output, s)  
  
    function show(n: int, t: tree) =  
      let function indent(s: string) =  
        (write("\n");  
         for i := 1 to n  
         do write(" "));  
        output := concat(output, s))  
      in if t = nil then indent(".")  
         else (indent(t.key);  
              show(n+1, t.left);  
              show(n+1, t.right))  
    end  
  
  in show(0, tree);  
  output  
end
```

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



Type References


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



Record Fields

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

The diagram illustrates the resolution of the variable `x` in the expression `origin.x`. Red arrows show the following path: from the `x` in `origin.x` to the `origin` variable, then to the `var origin` declaration, then to the `point` type definition, and finally to the `x` field in the `point` struct. A red box highlights the `x` in `origin.x`.

Type Dependent Name Resolution

Name Binding Rules

What are the name binding rules of a language?

- What are introductions of names?
- What are uses of names?
- What are the shadowing rules?
- What are the name spaces?

How can we define the name binding rules of a language?

- What is the BNF for name binding?

Name Resolution in Scope Graphs

Wanted: Separation of Concerns in Name Binding

Representation

- To conduct and represent the results of name resolution

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- ...

Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- To define name binding rules of a language

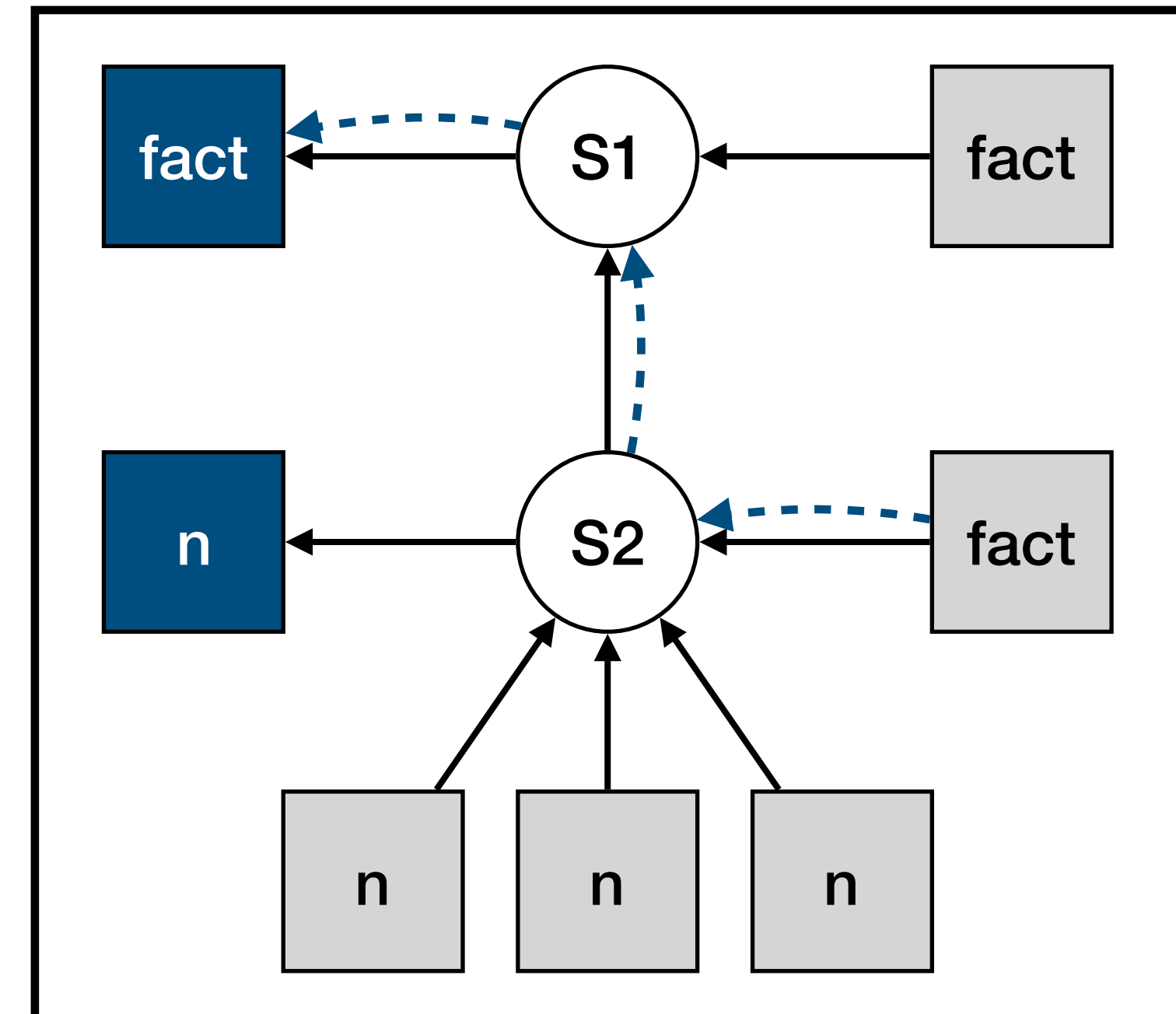
Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- ...

Program

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

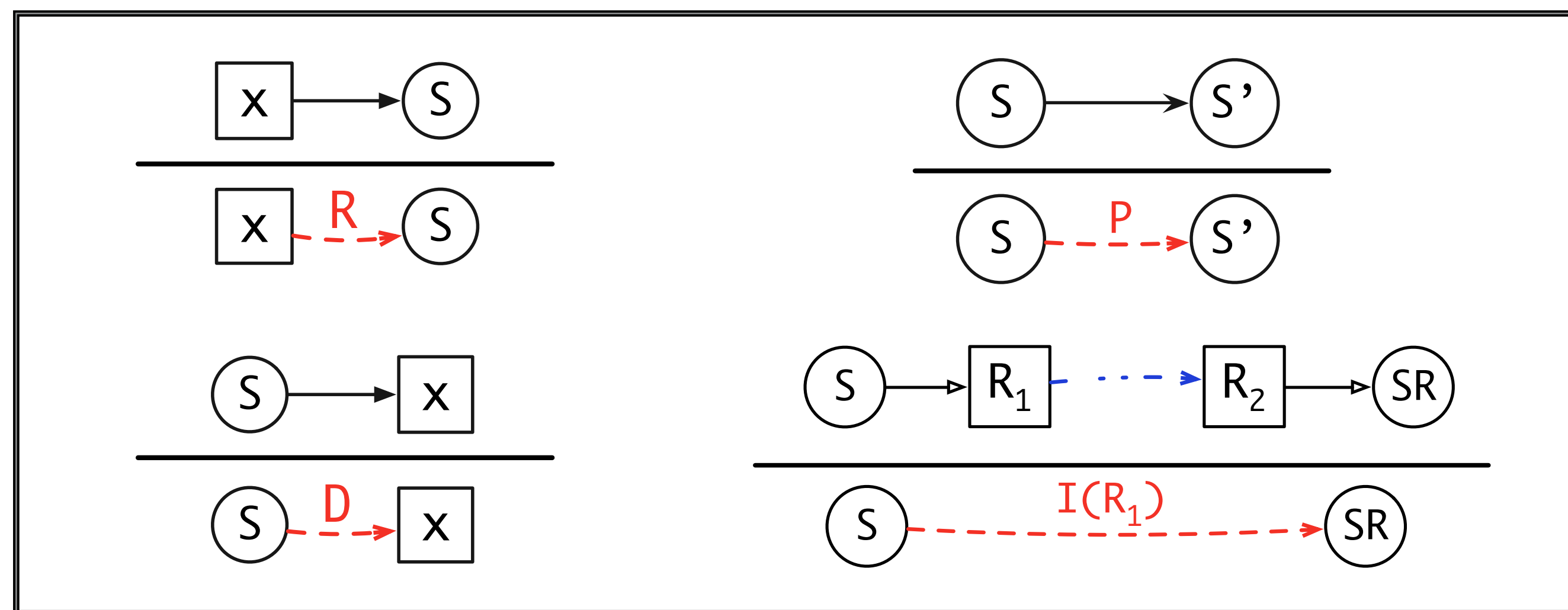
Scope Graph



Name Resolution

A Calculus for Name Resolution

Scopes, References, Declarations, Parents, Imports

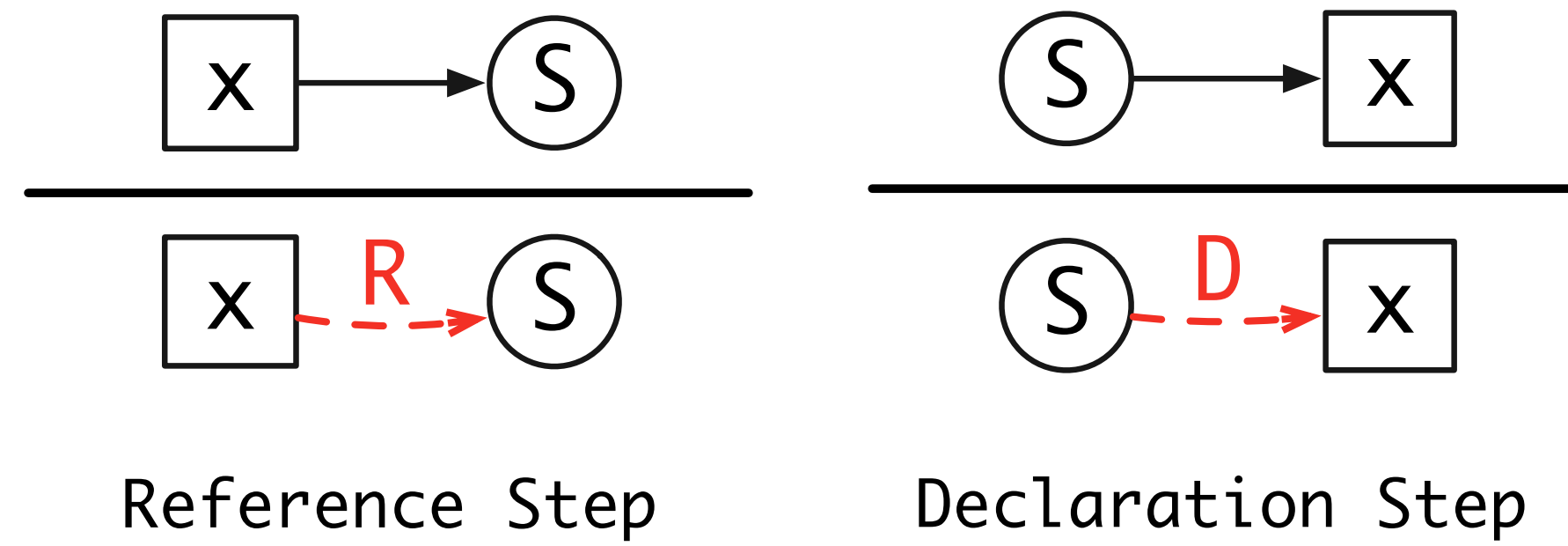
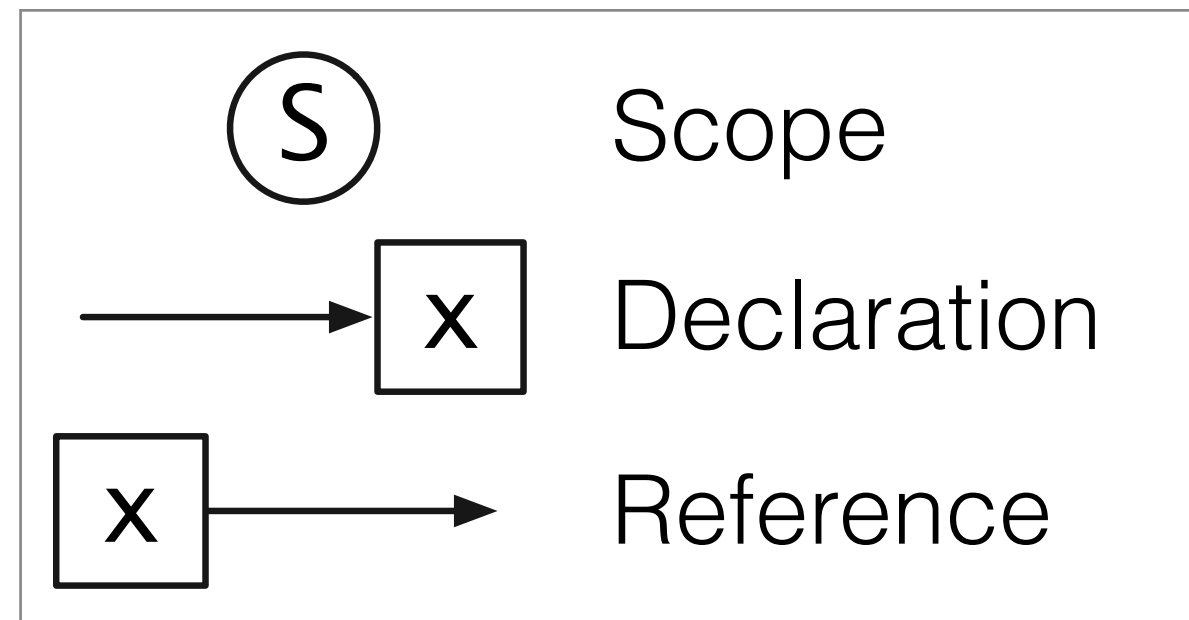
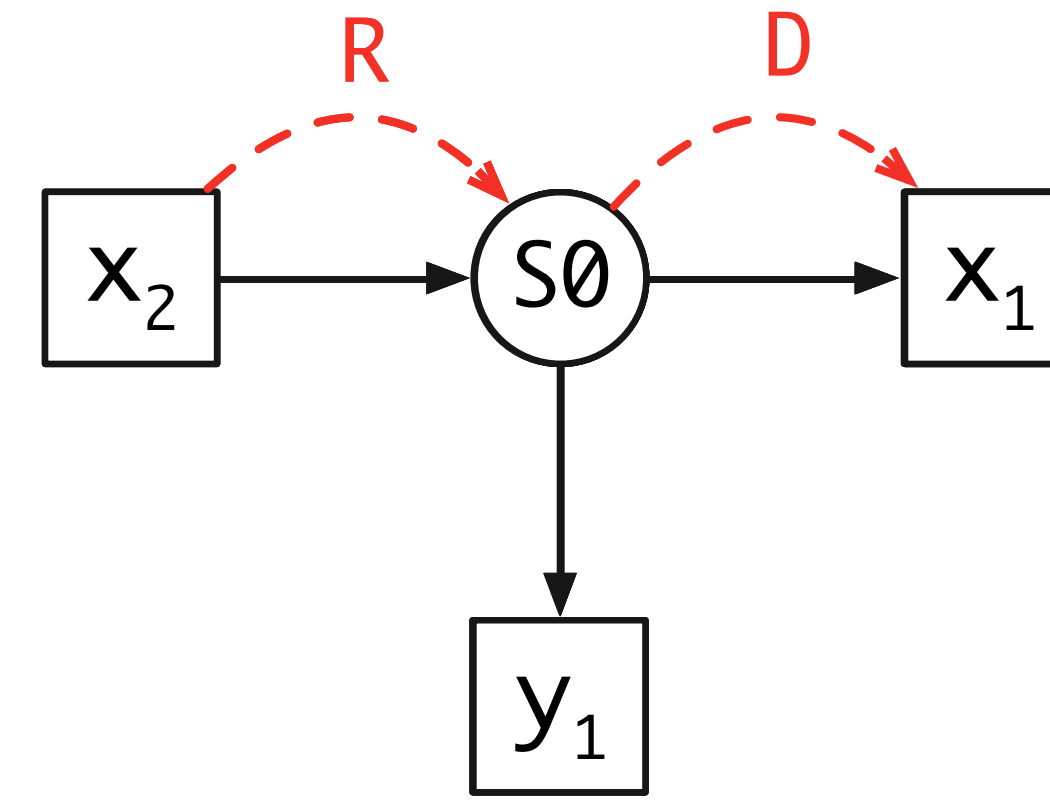


Path in scope graph connects reference to declaration

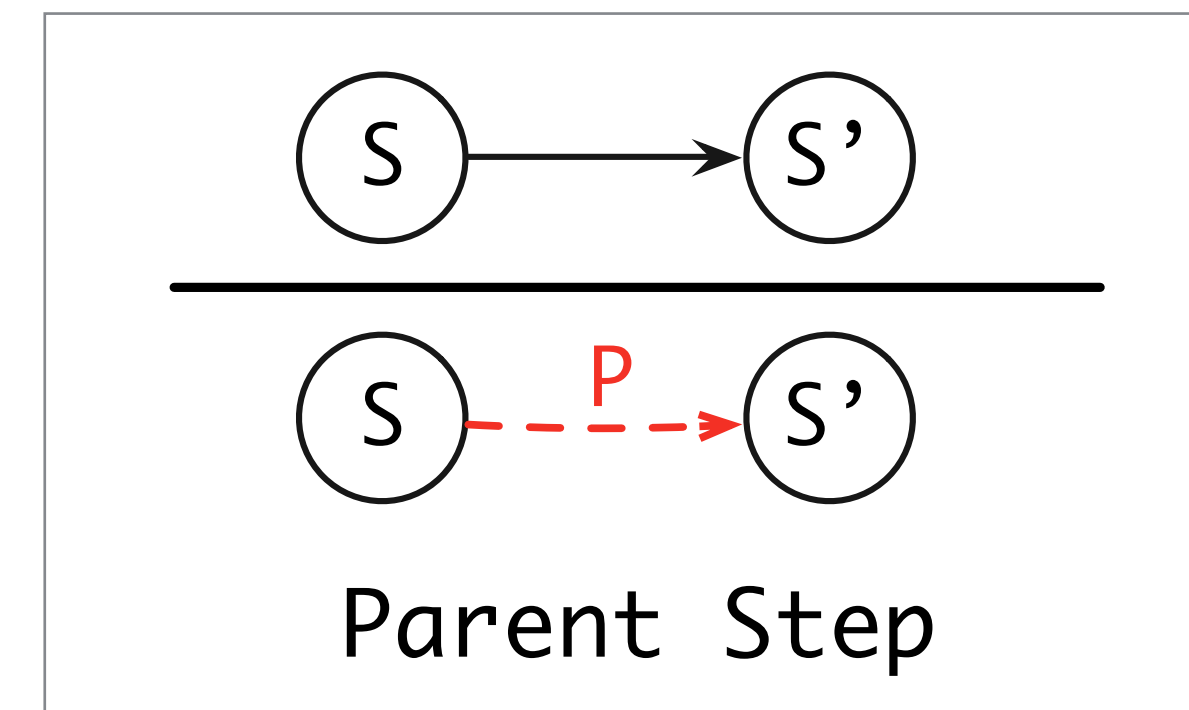
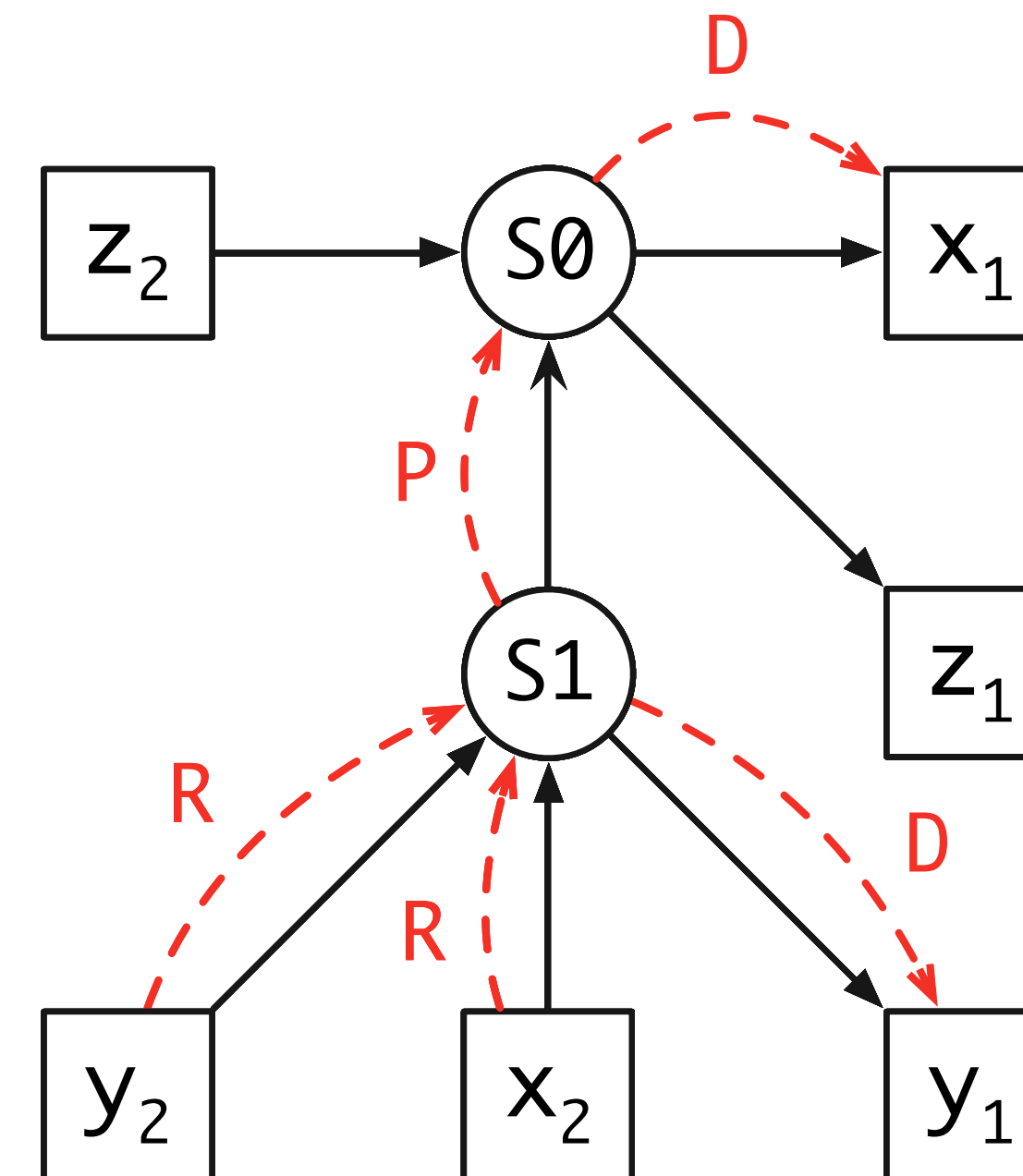
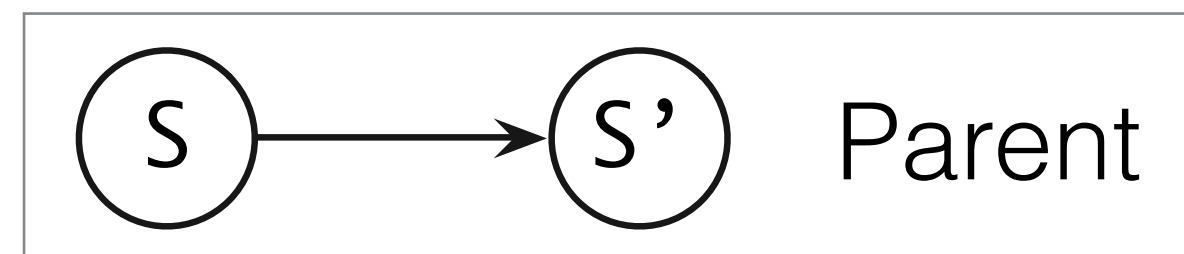
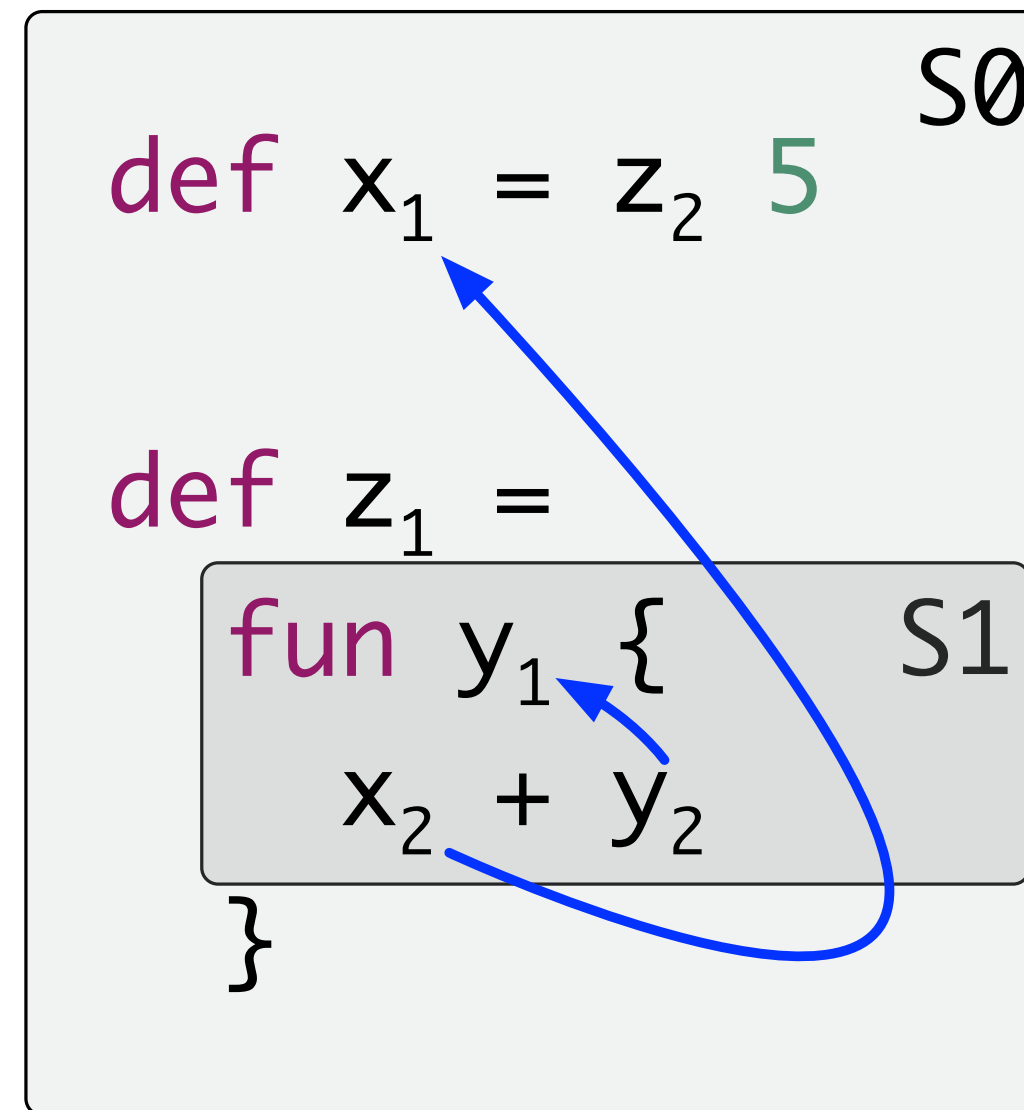
Neron, Tolmach, Visser, Wachsmuth
A Theory of Name Resolution
ESOP 2015

Simple Scopes

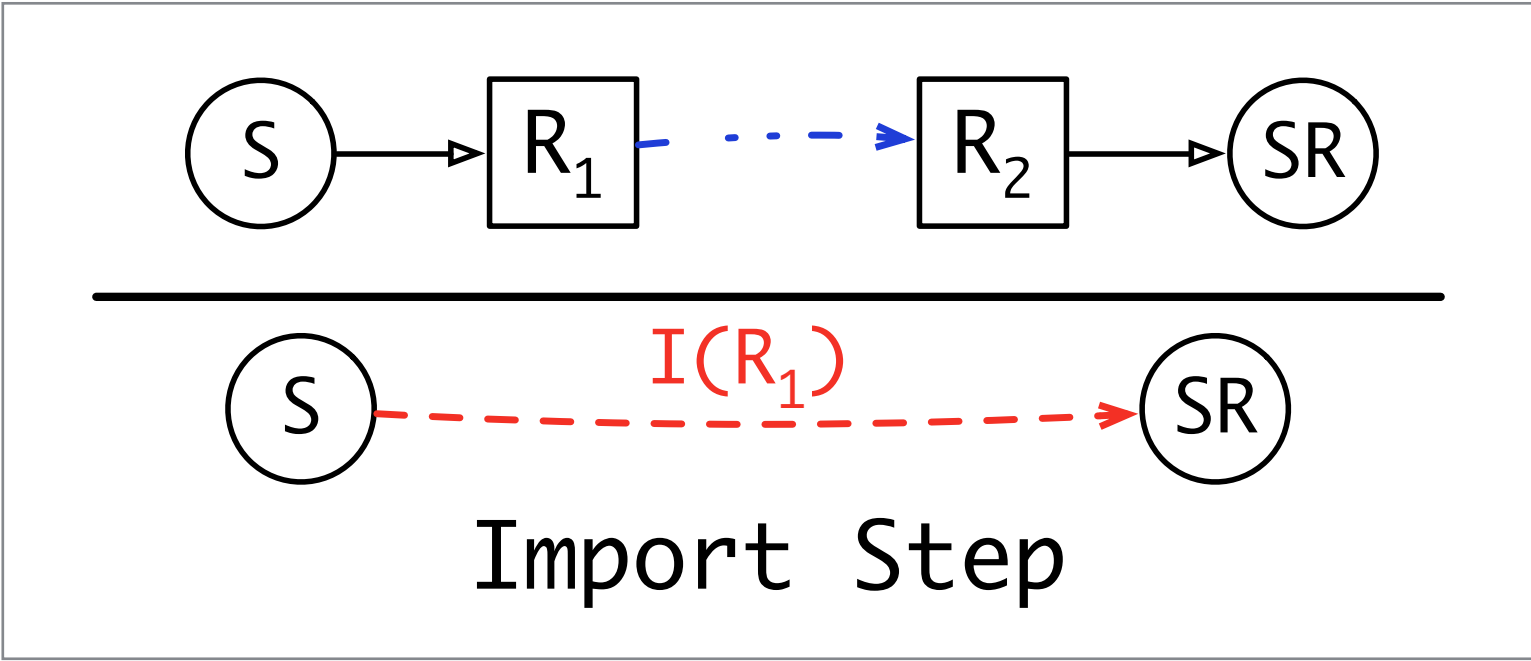
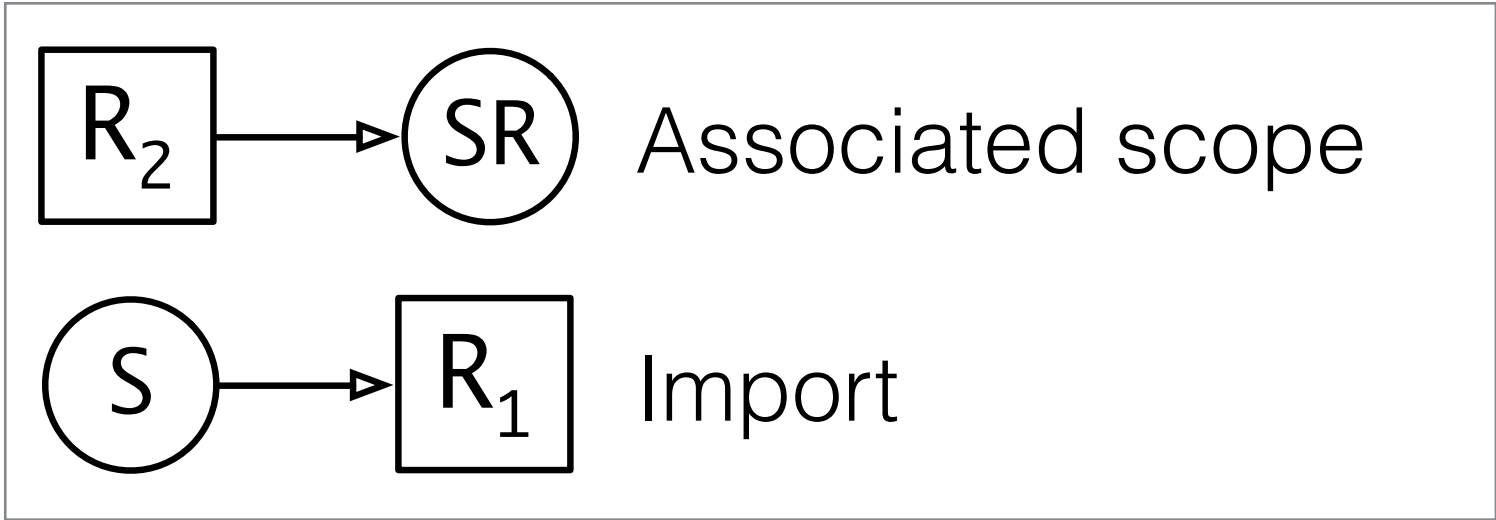
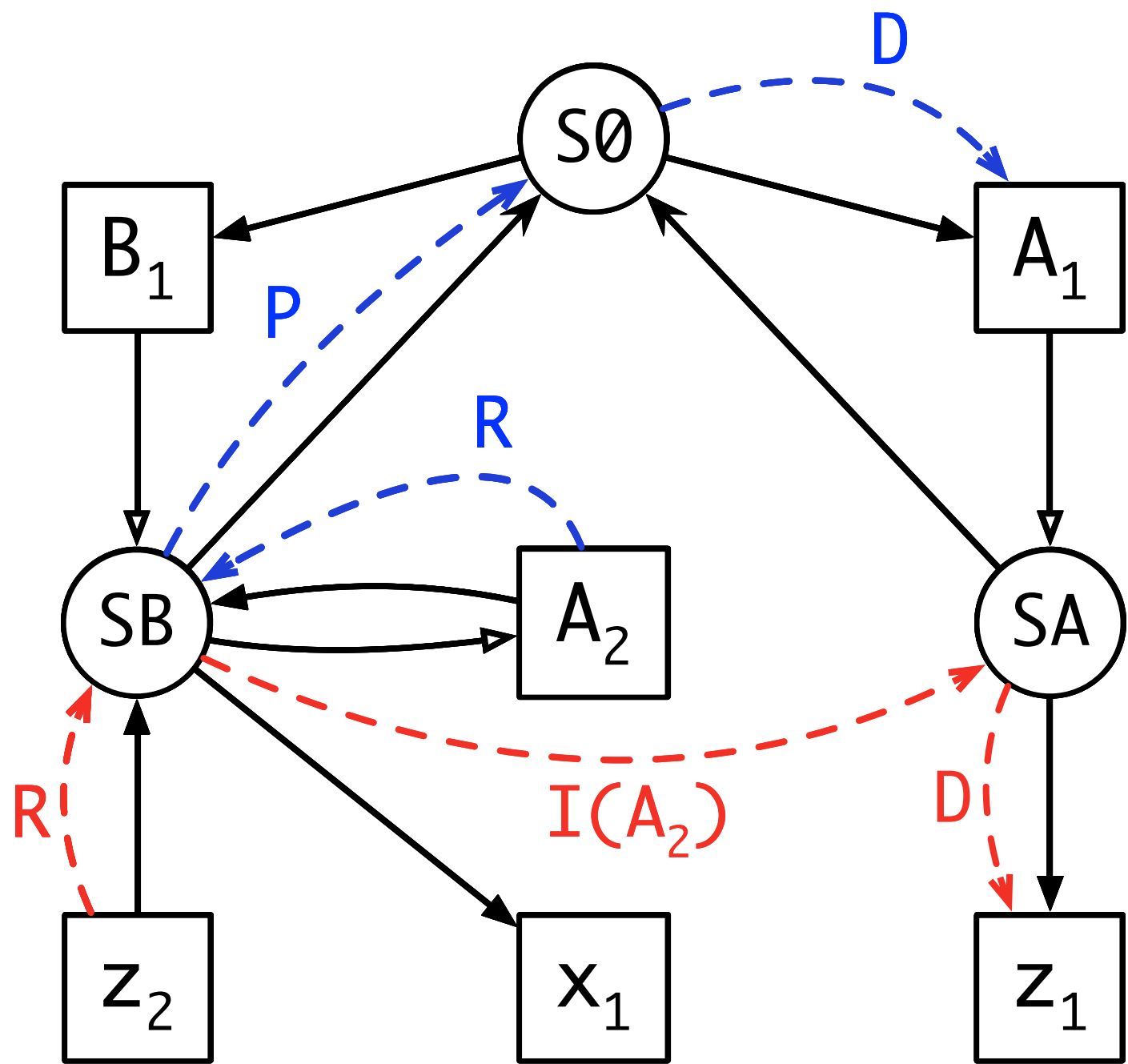
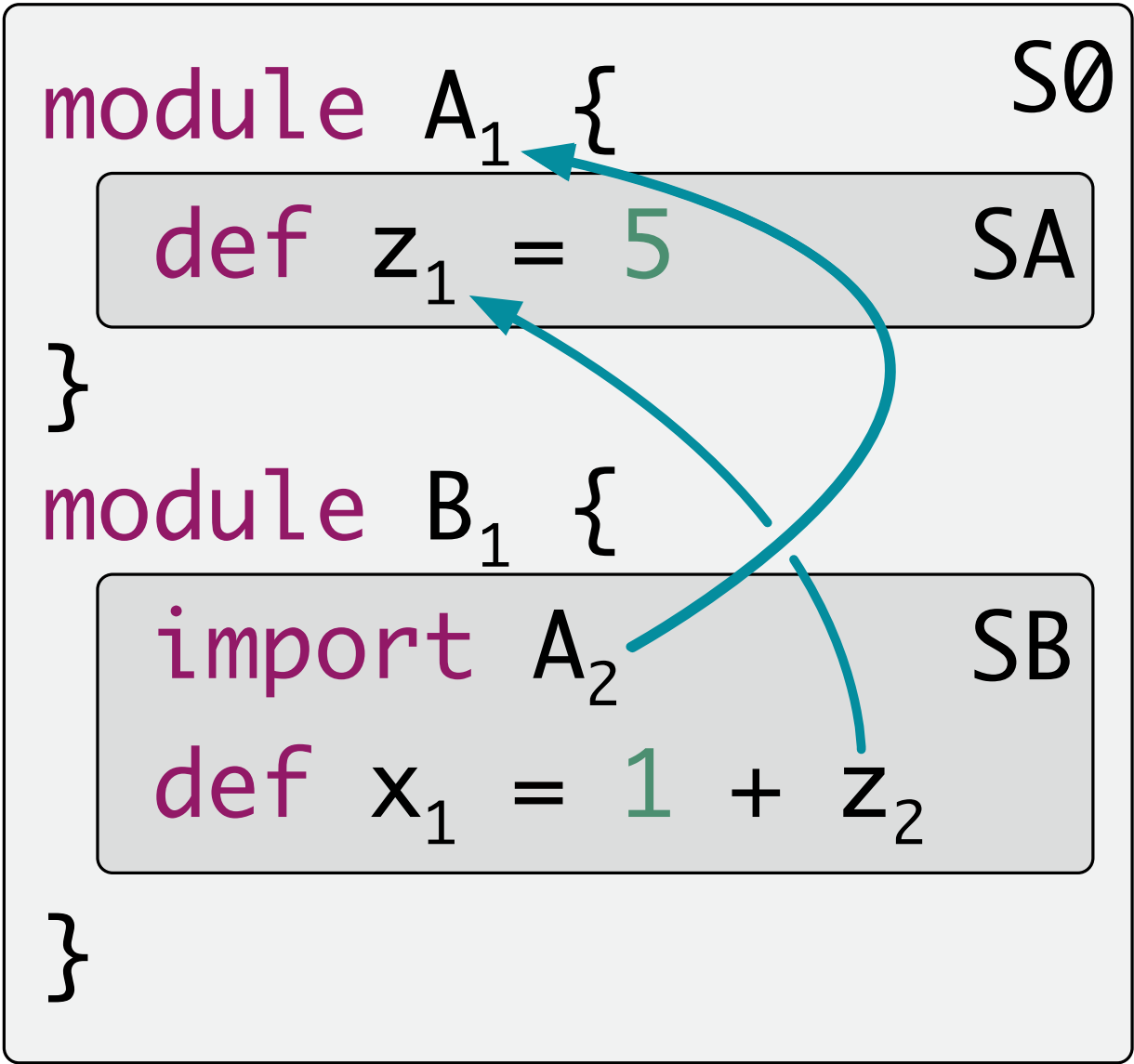
`def y1 = x2 + 1`
`def x1 = 5` S₀



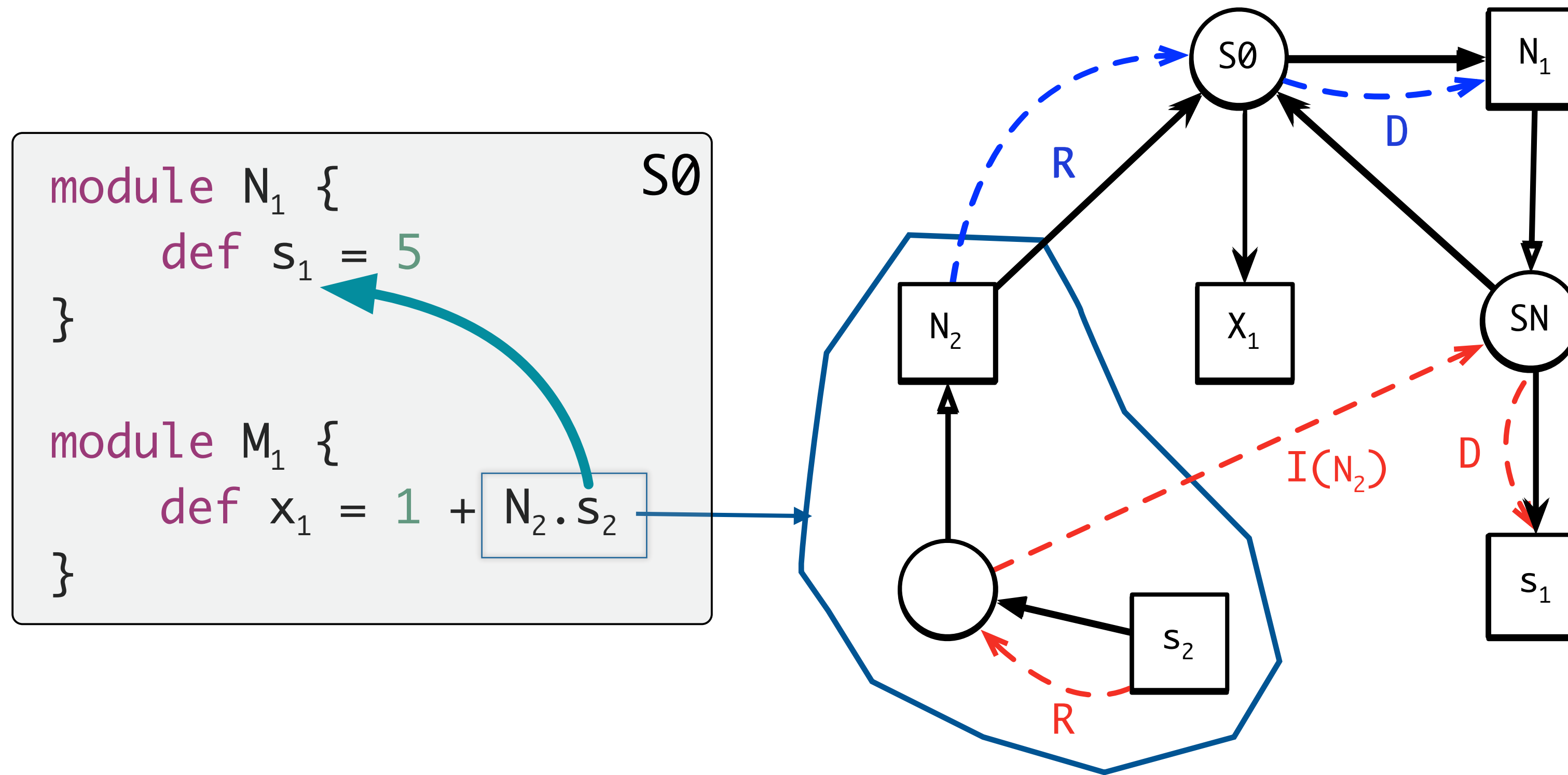
Lexical Scoping



Imports

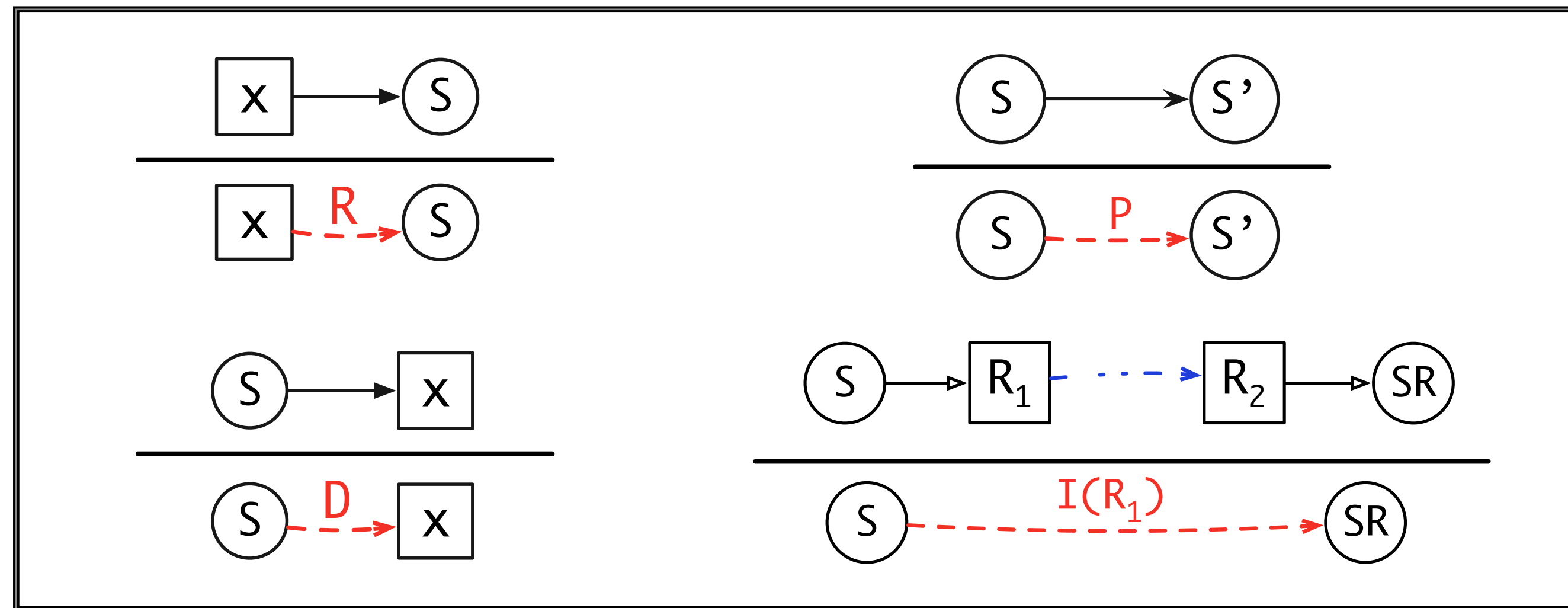


Qualified Names



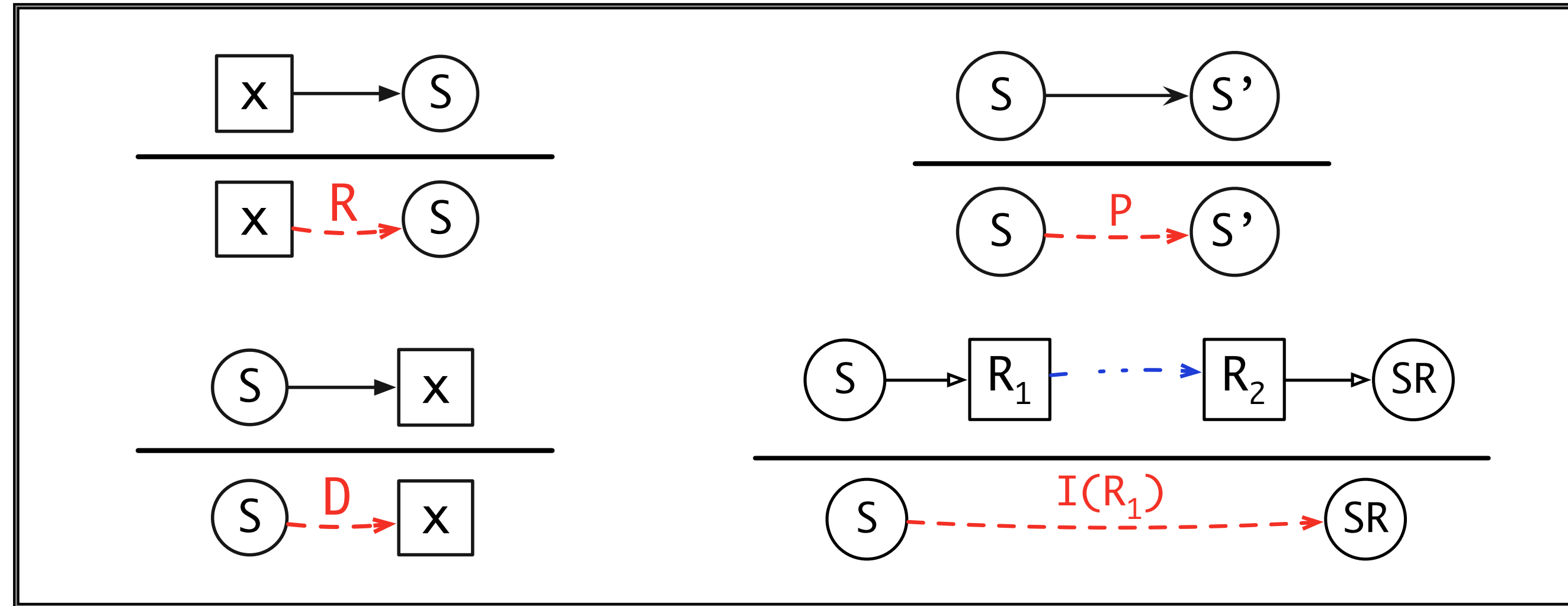
A Calculus for Name Resolution

Reachability of declarations from
references through scope graph edges



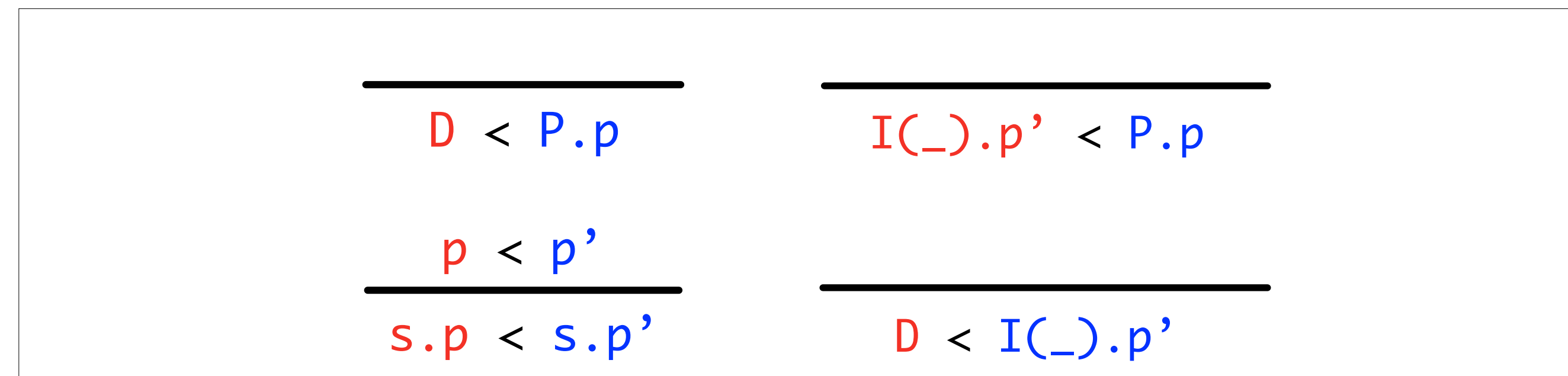
How about ambiguities?
References with multiple paths

A Calculus for Name Resolution



Reachability

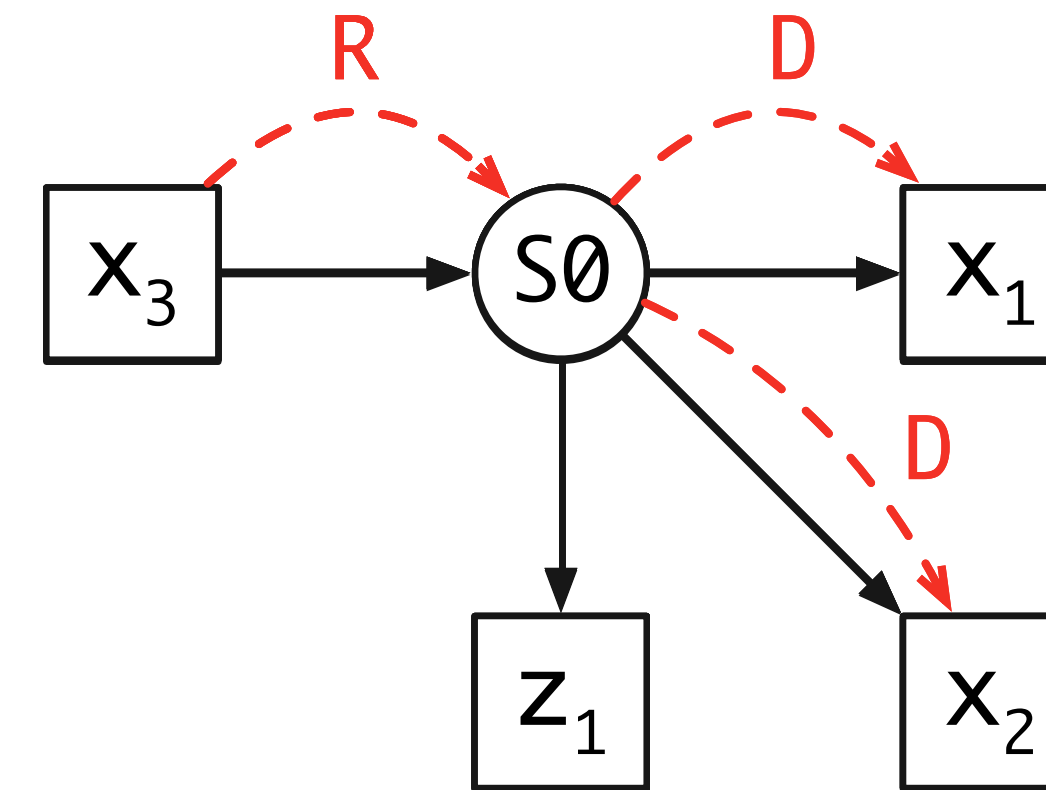
Well formed path: $R.P^*.I(_)*.D$



Visibility

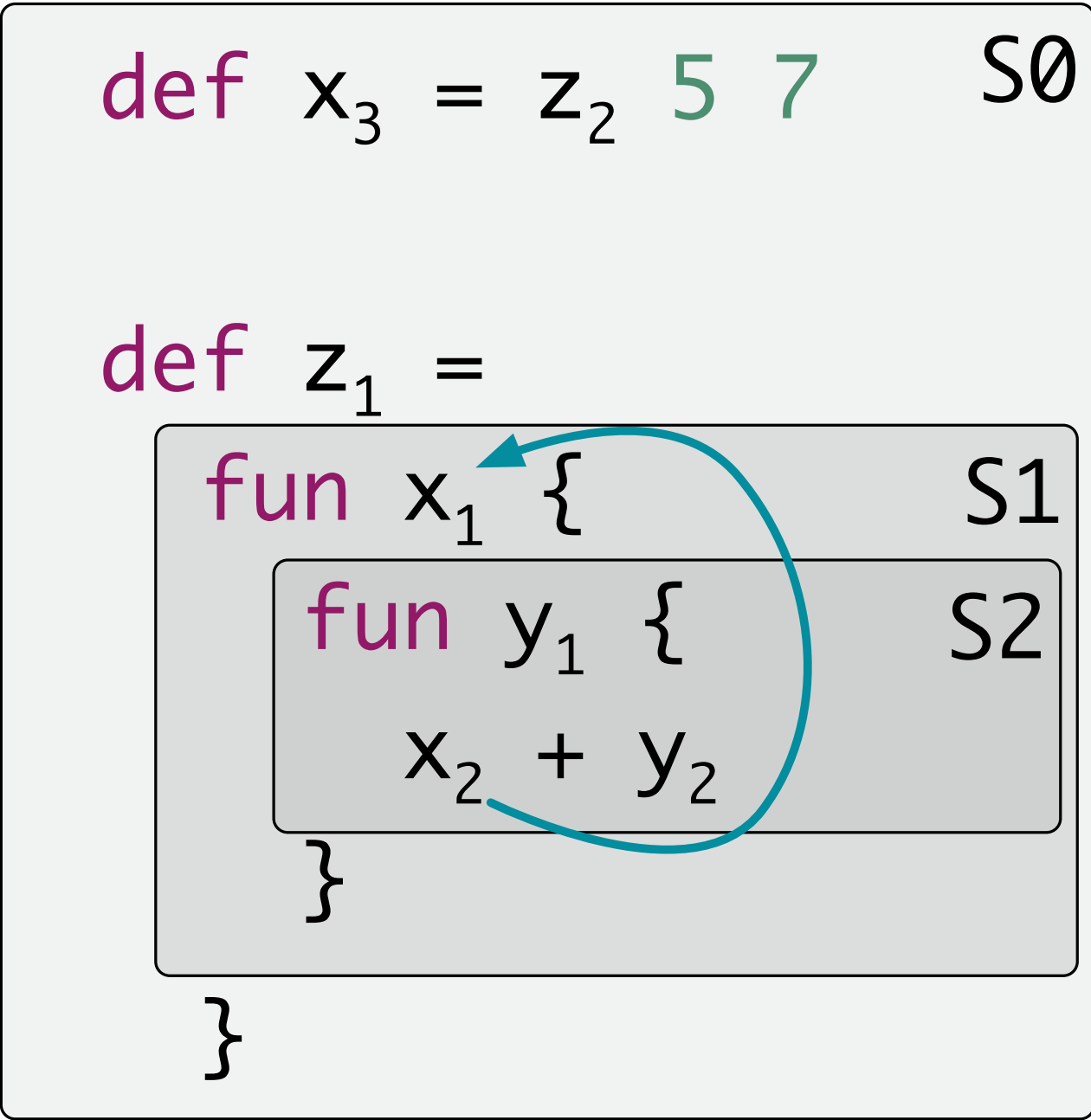
Ambiguous Resolutions

```
def x1 = 5      S0  
def x2 = 3  
def z1 = x3 + 1
```

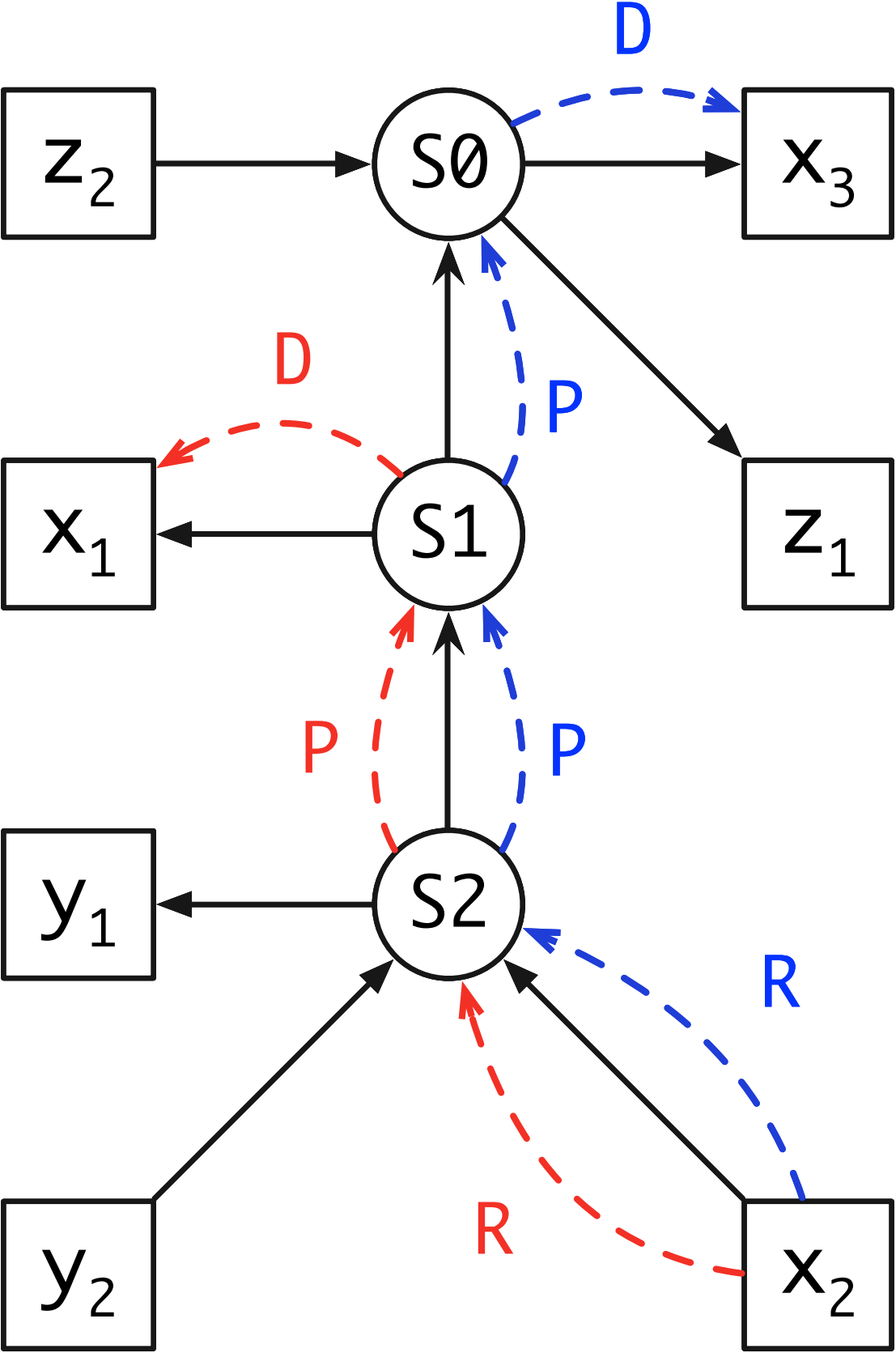


```
match t with  
| A x | B x => ...
```


Shadowing

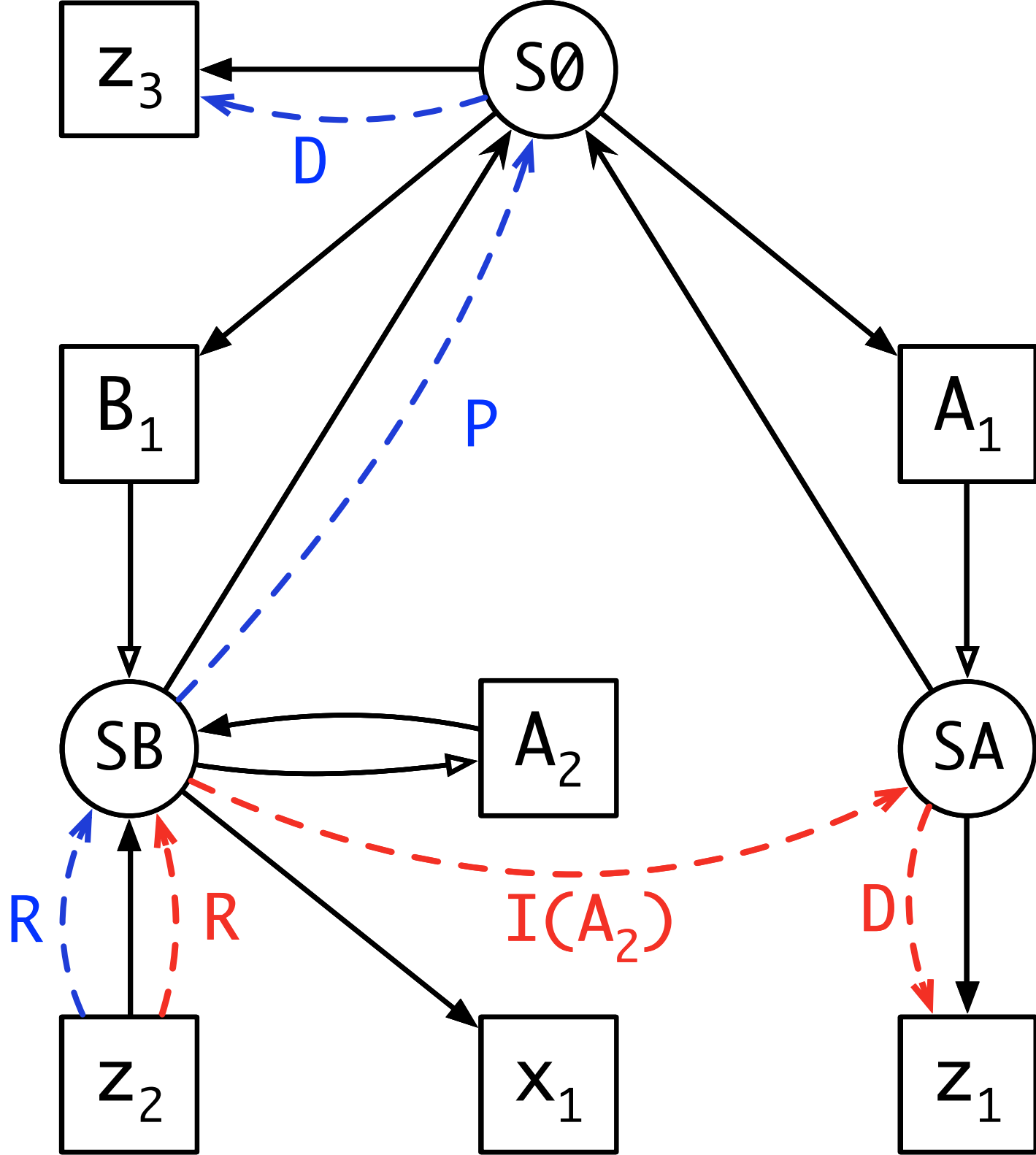
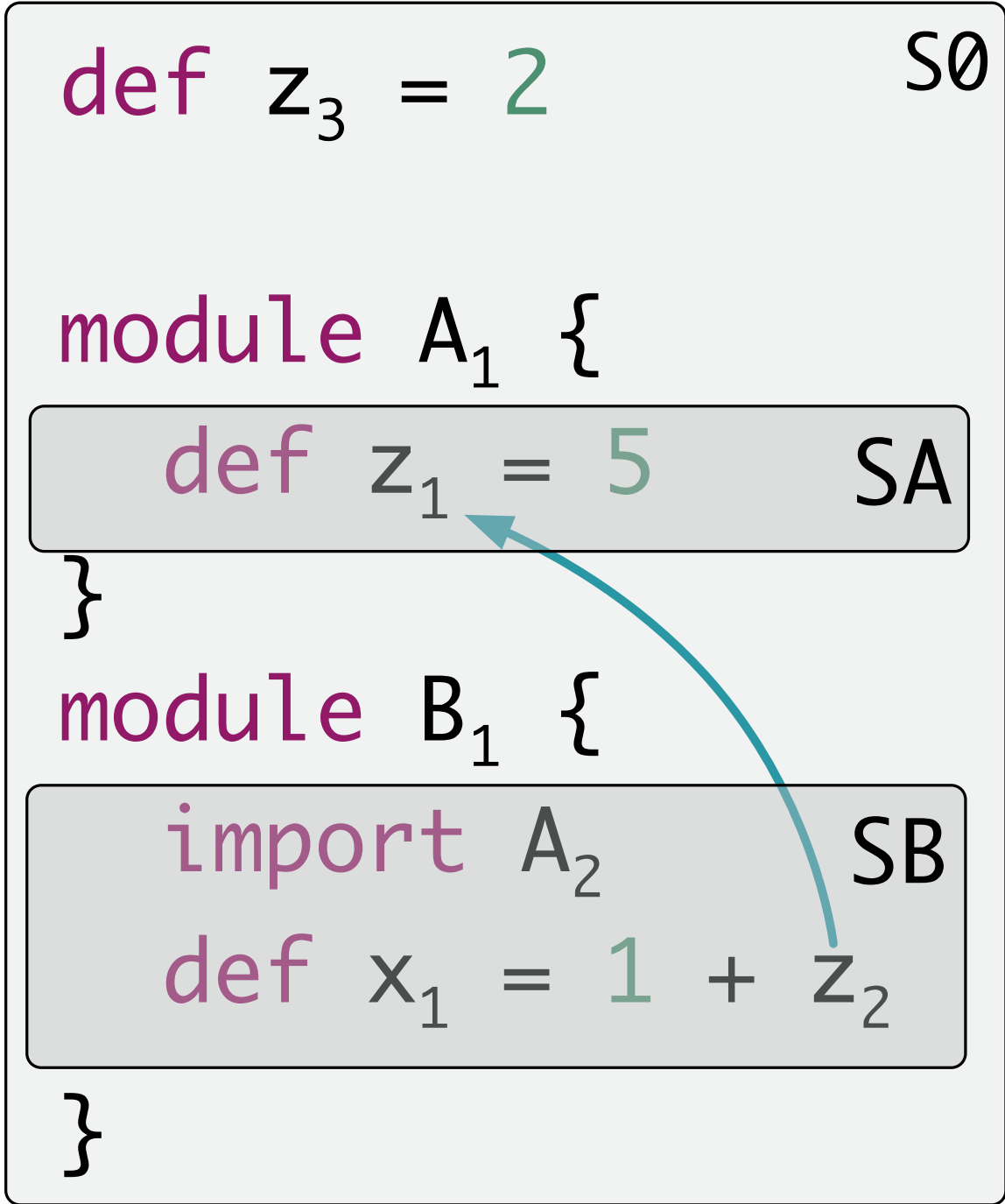


$$\frac{}{D < P.p}$$
$$\frac{p < p'}{s.p < s.p'}$$



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

Imports shadow Parents

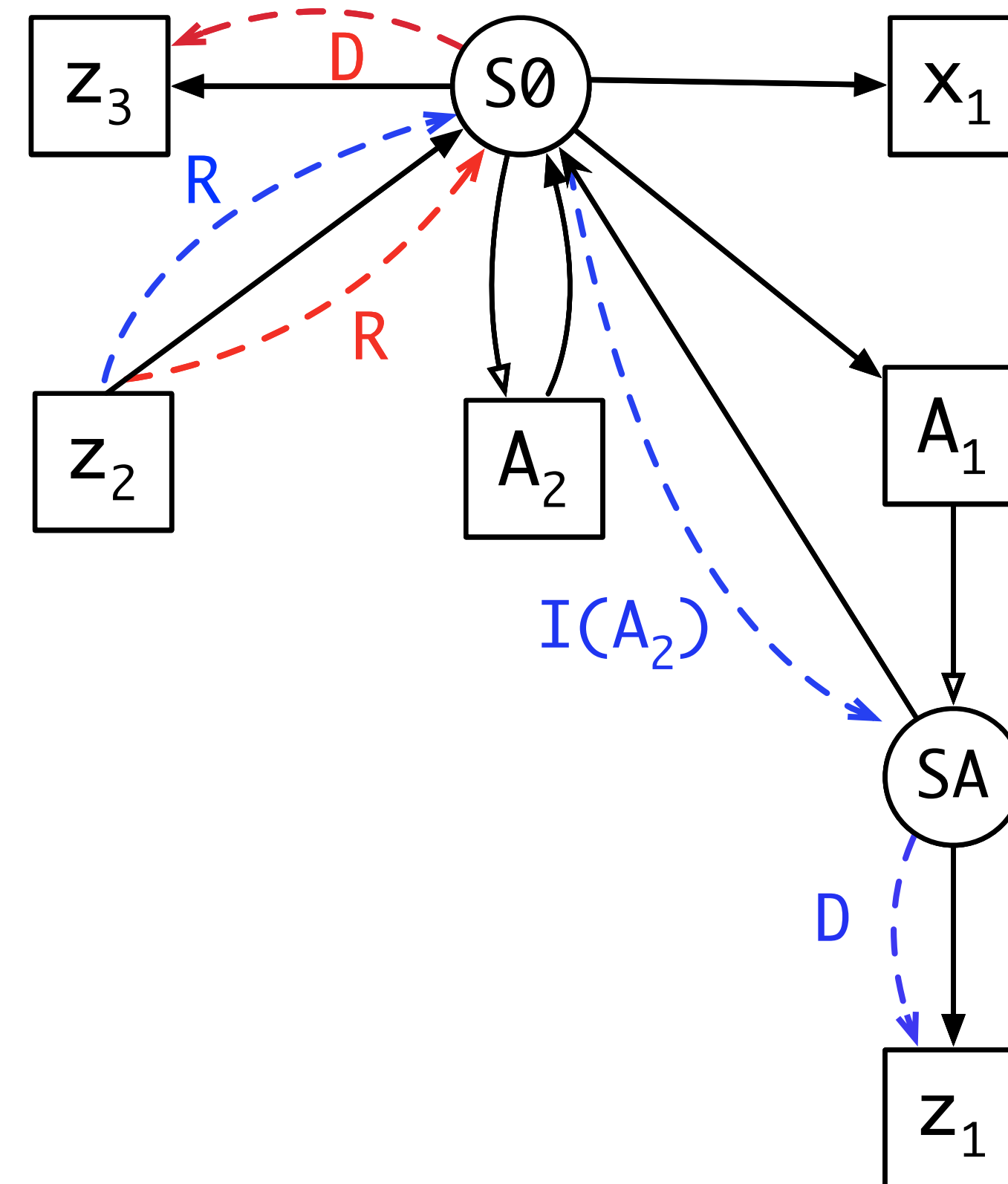
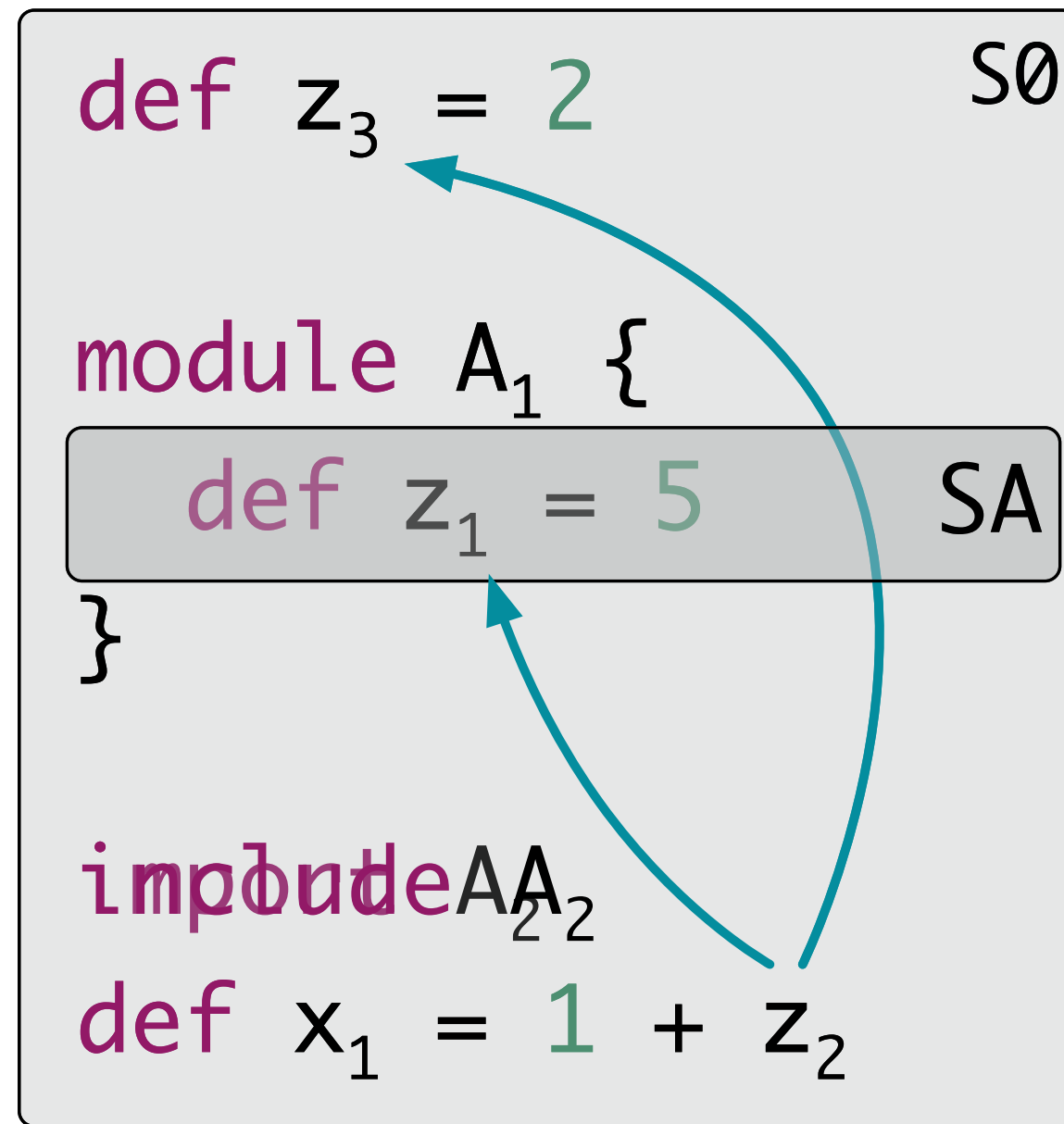


$$I(_).p' < P.p$$

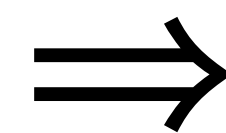
\Rightarrow

$$R.I(A_2).D < R.P.D$$

Imports vs. Includes

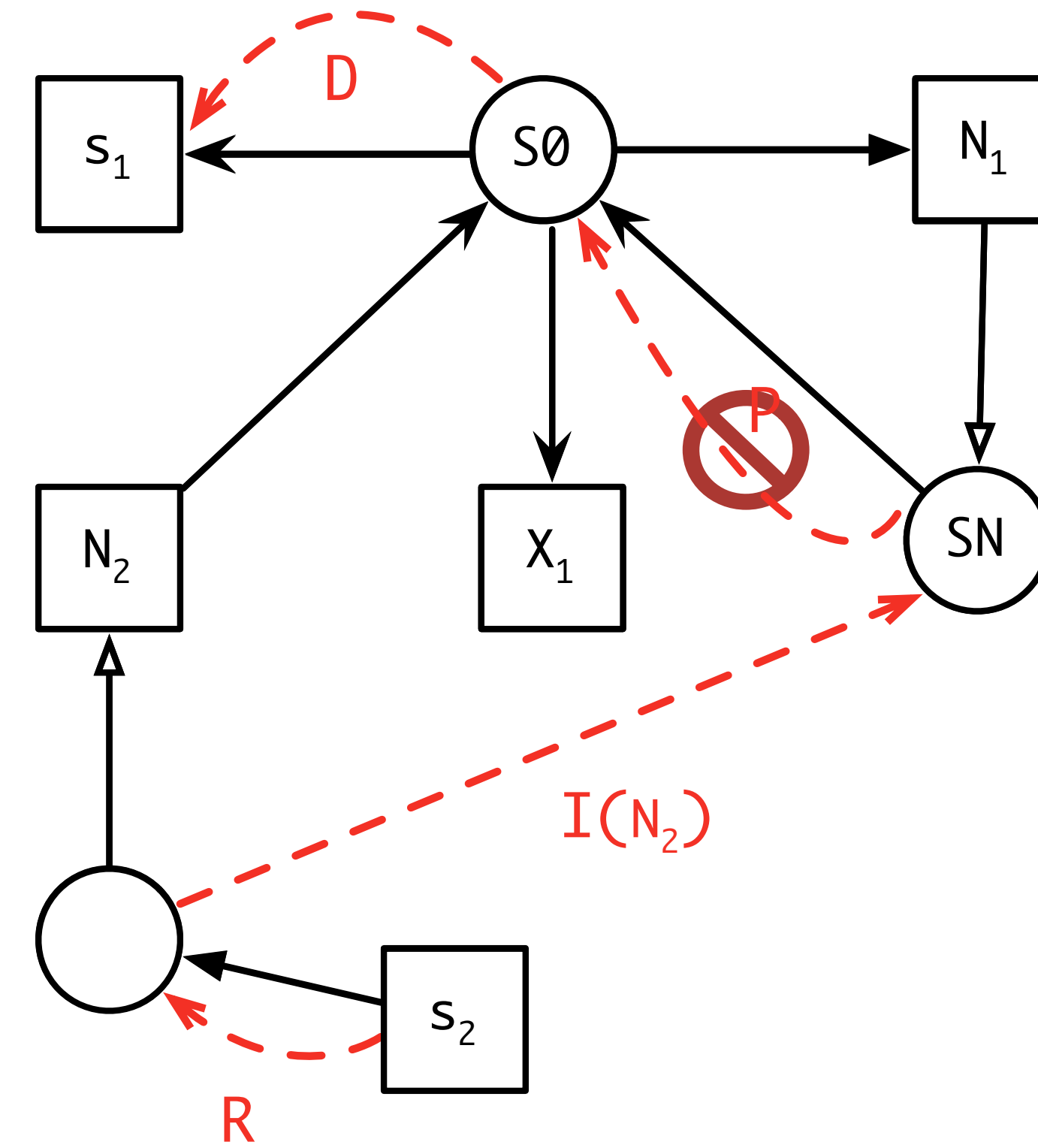
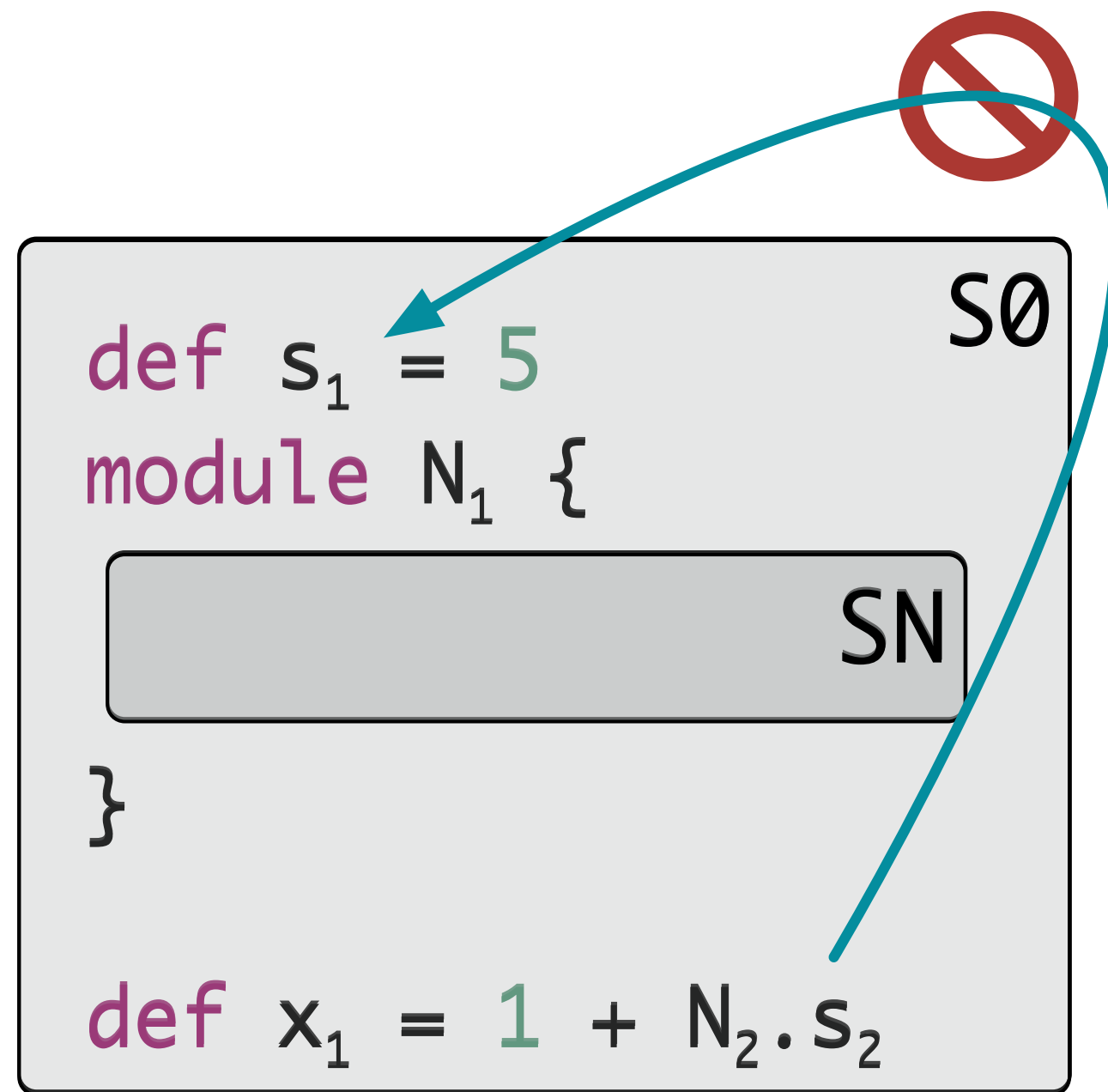


~~$D < I(_).p'$~~



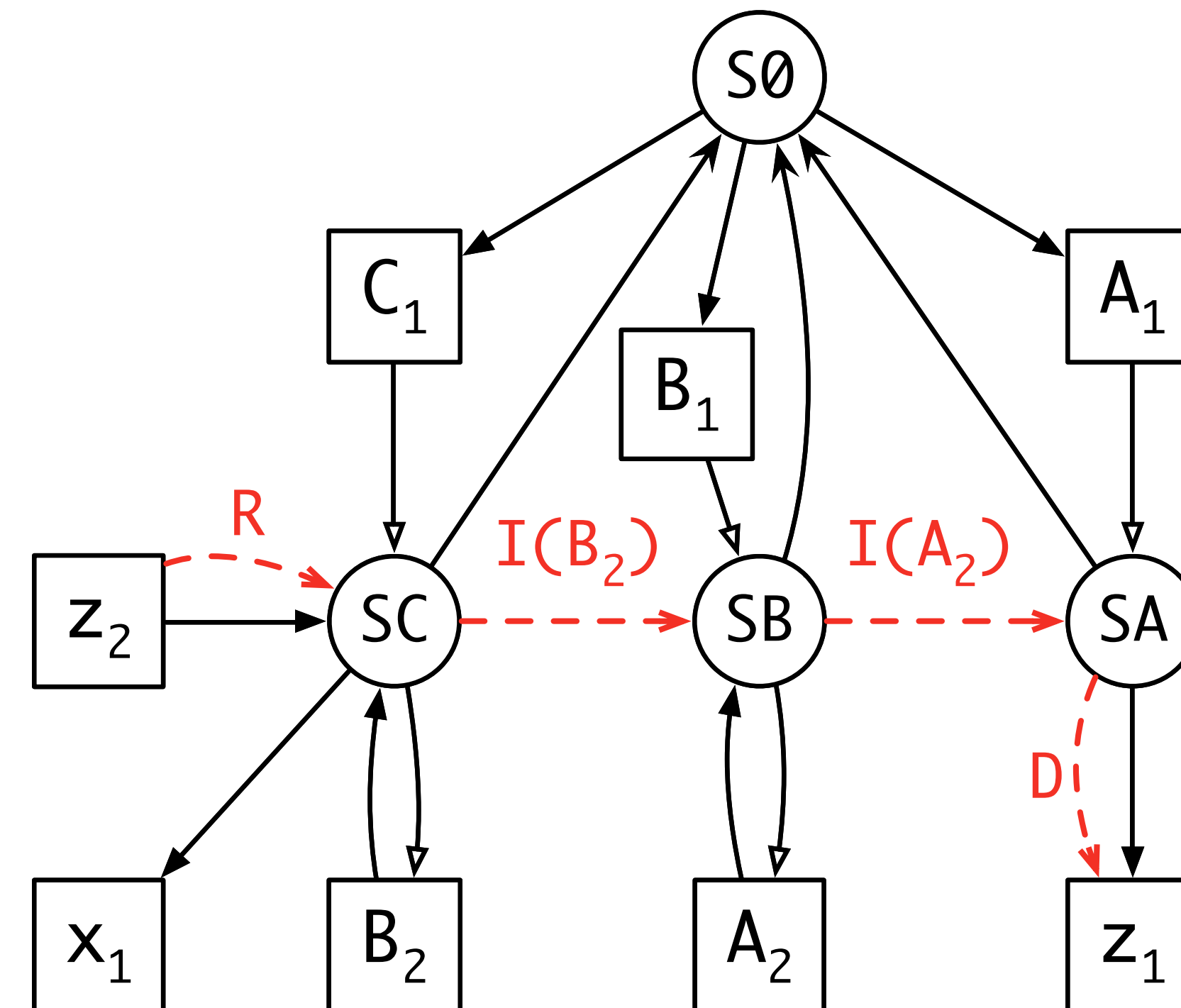
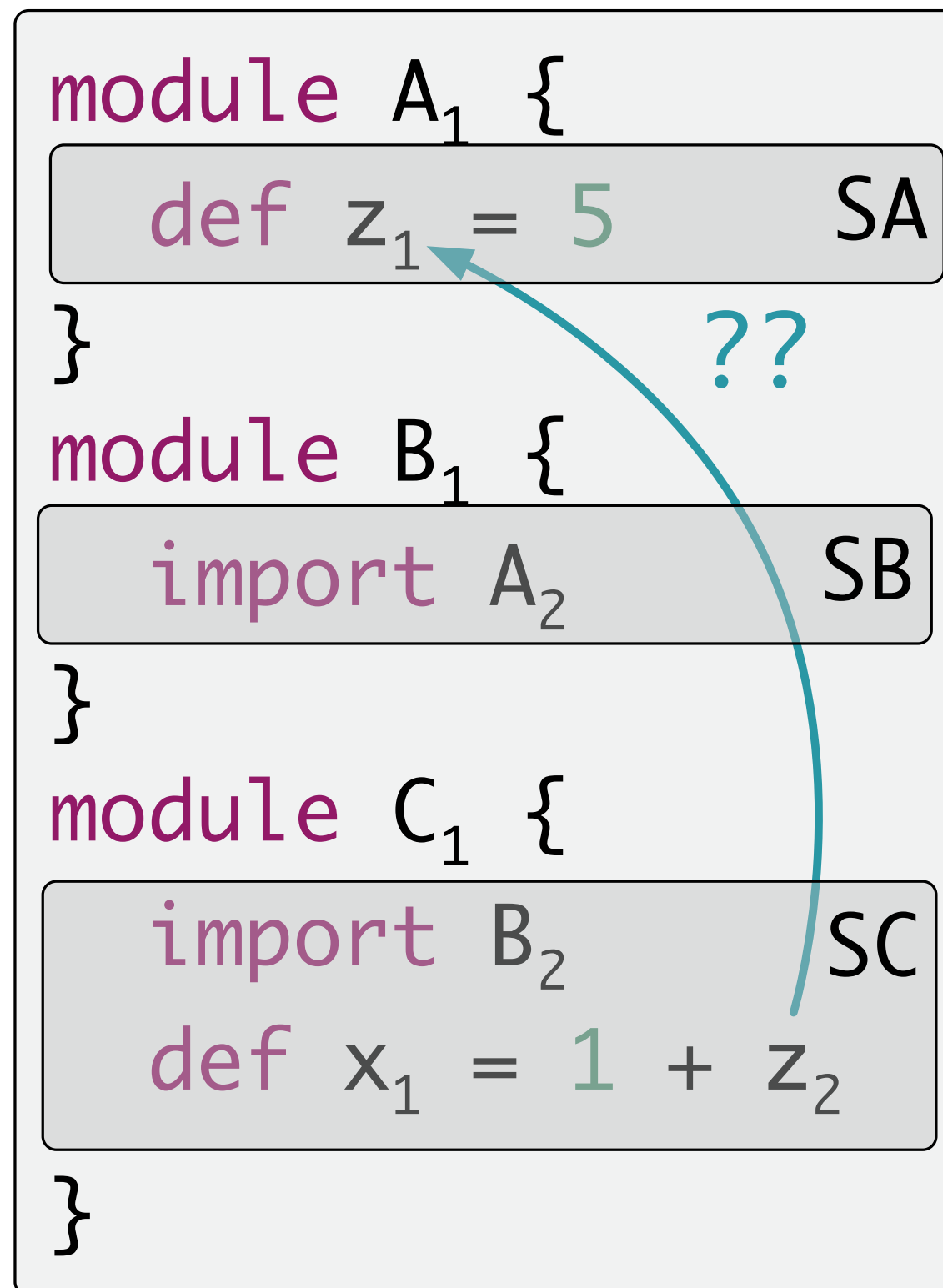
$R.D < R.I(A_2).D$

Import Parents



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

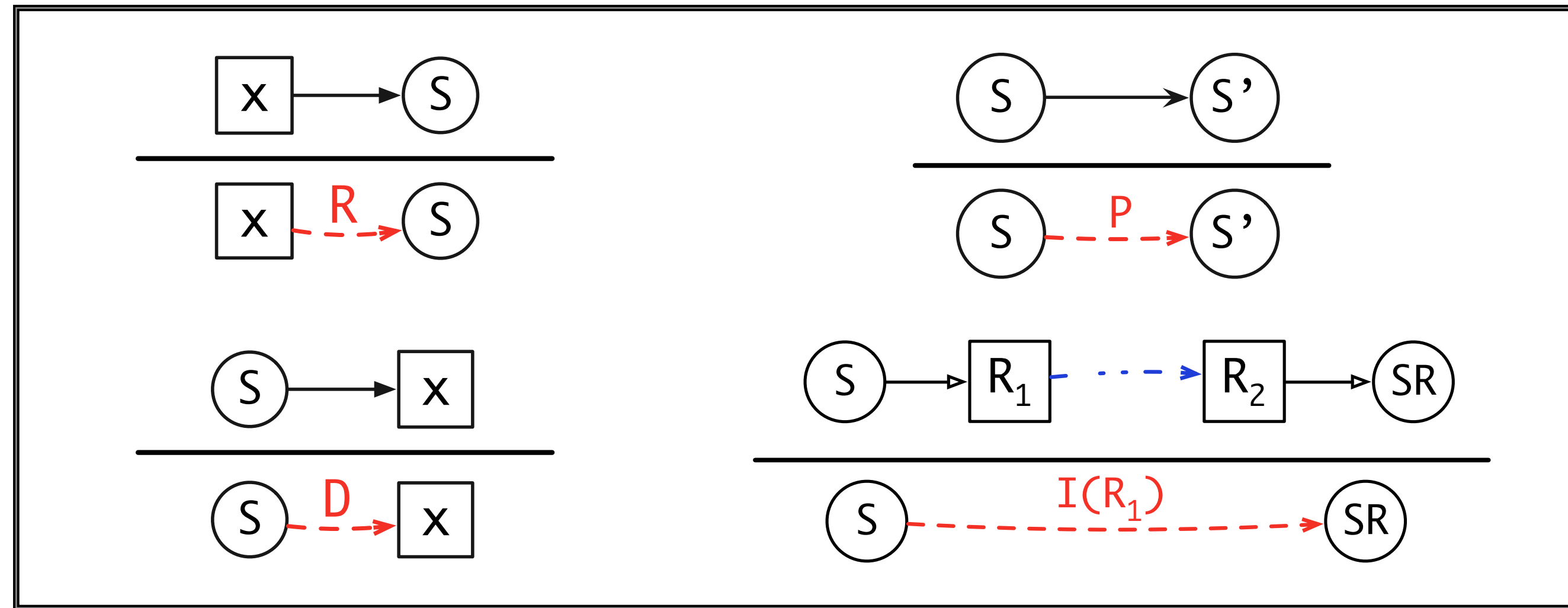
Transitive vs. Non-Transitive



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

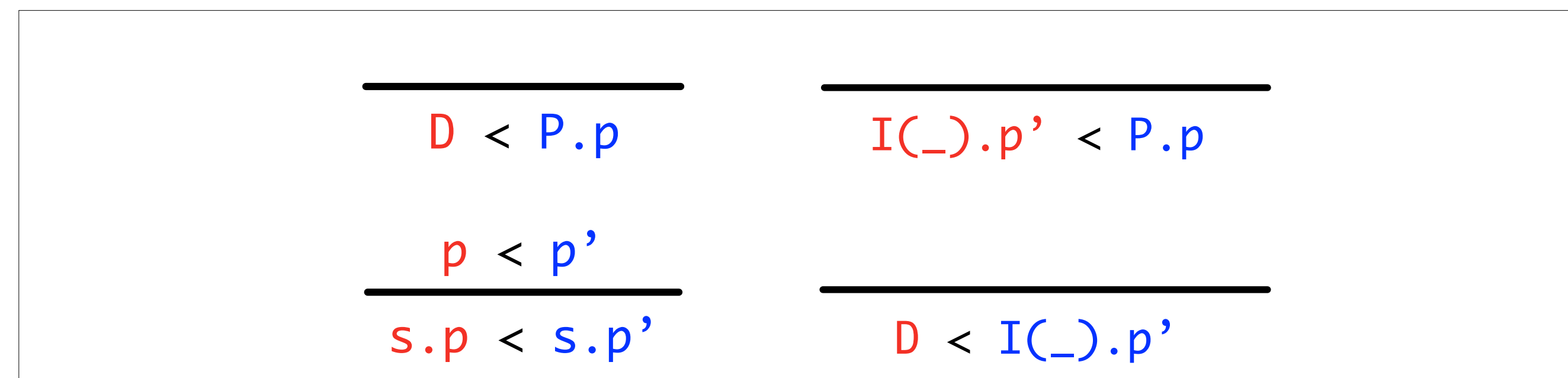
With non-transitive imports, a well formed path is $R.P^*.I(_)?.D$

A Calculus for Name Resolution



Reachability

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



Visibility

Visibility Policies

Lexical scope

$$\mathcal{L} := \{\mathbf{P}\} \quad \mathcal{E} := \mathbf{P}^* \quad \mathbf{D} < \mathbf{P}$$

Non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{I}^? \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P}$$

Transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{TI}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{TI}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}$$

Transitive Includes

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{Inc}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}$$

Transitive includes and imports, and non-transitive imports

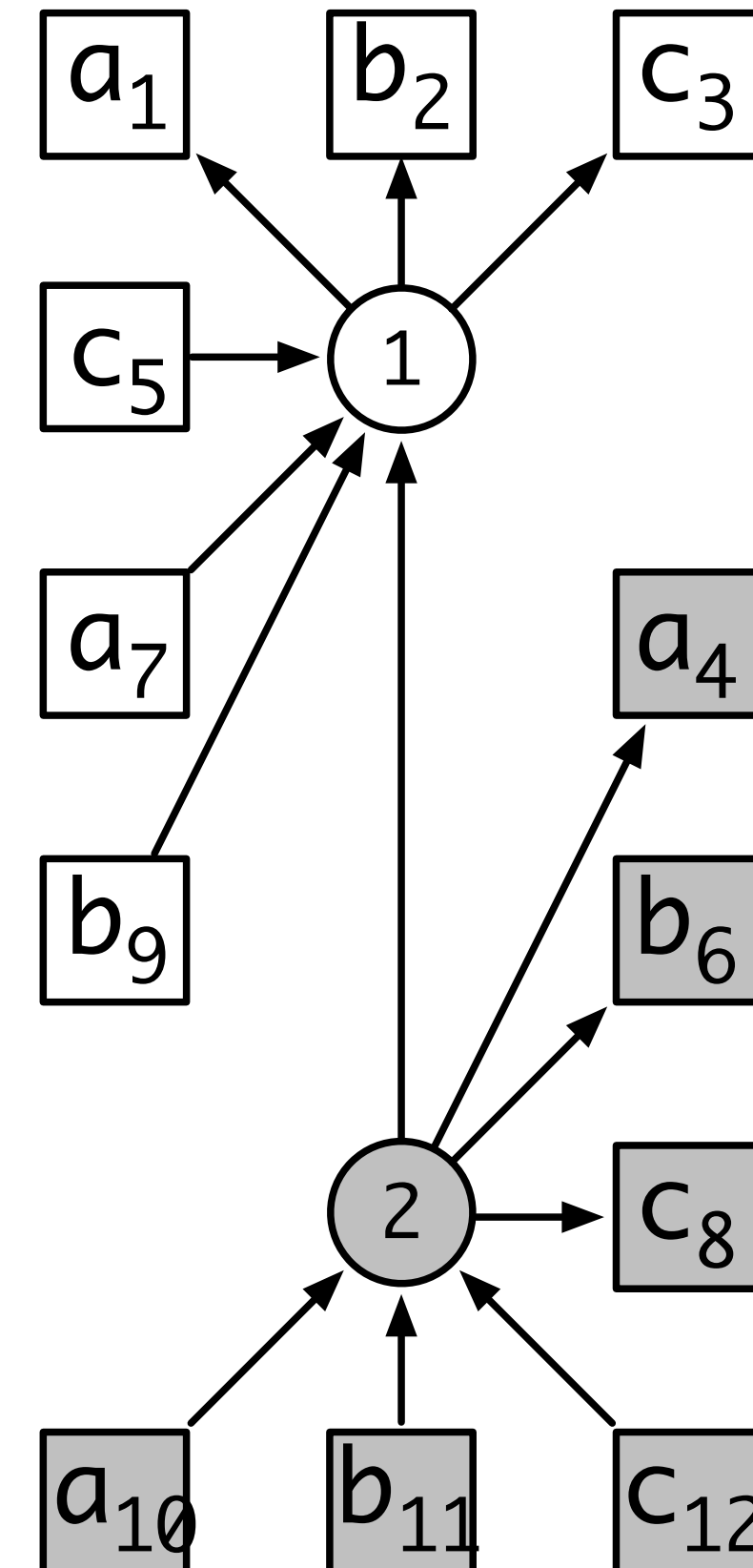
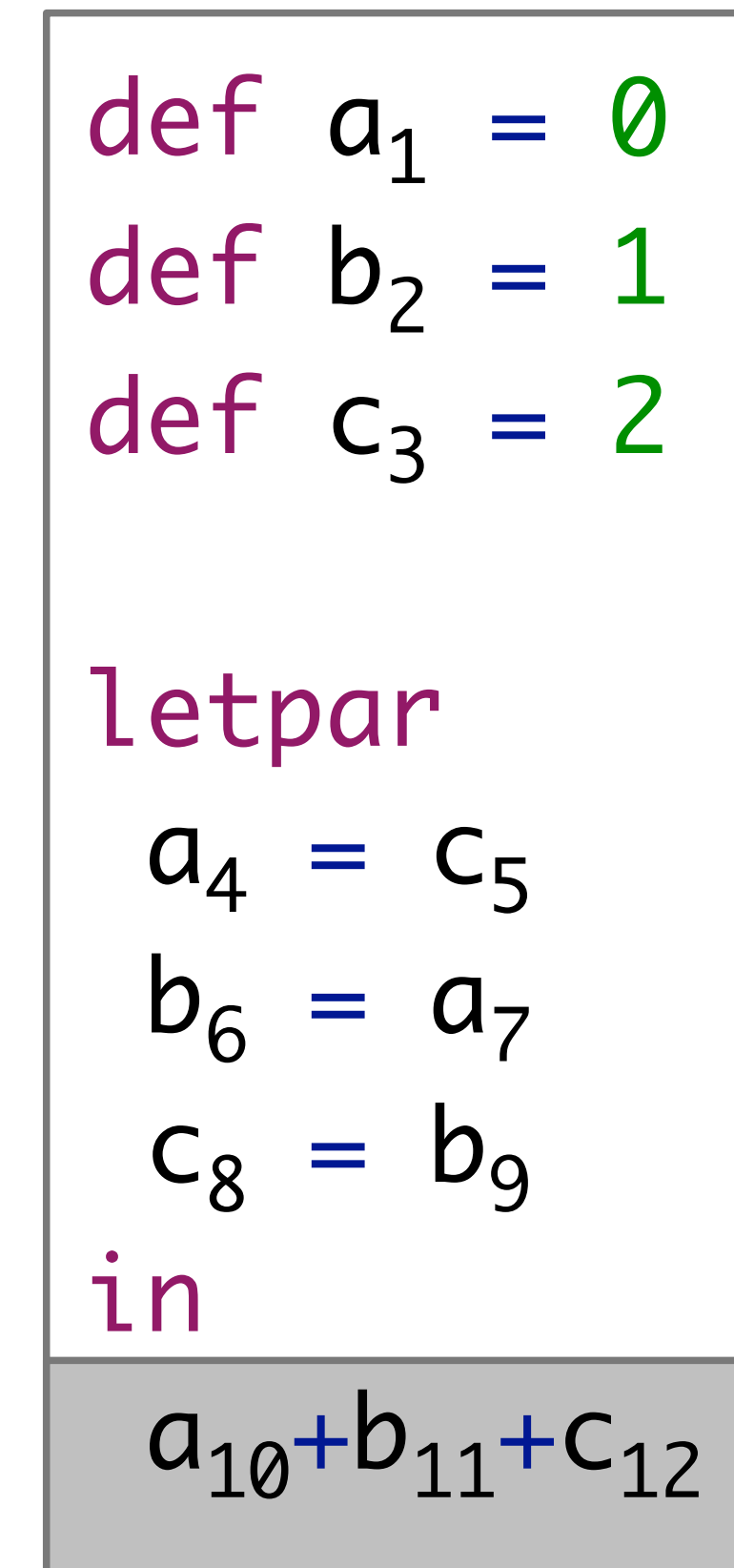
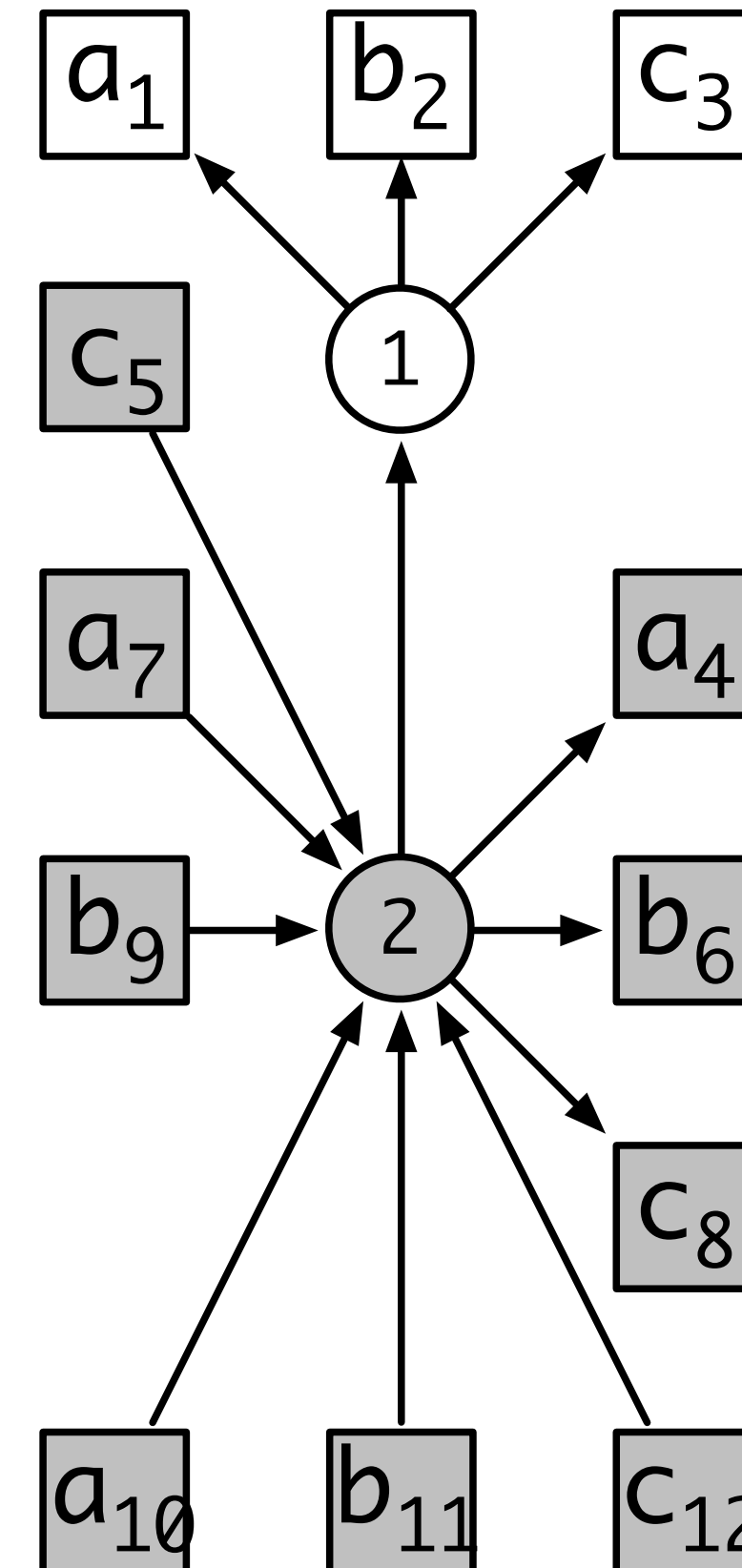
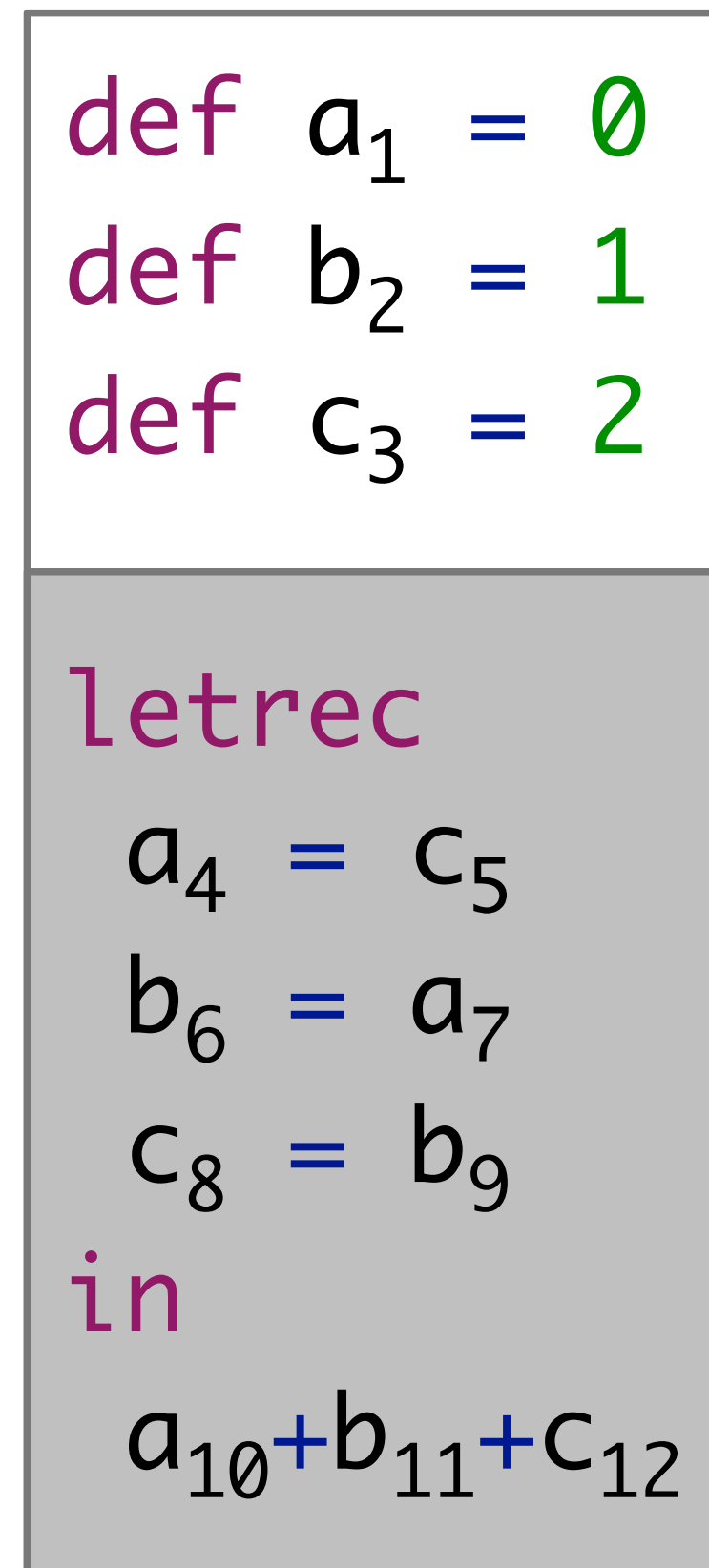
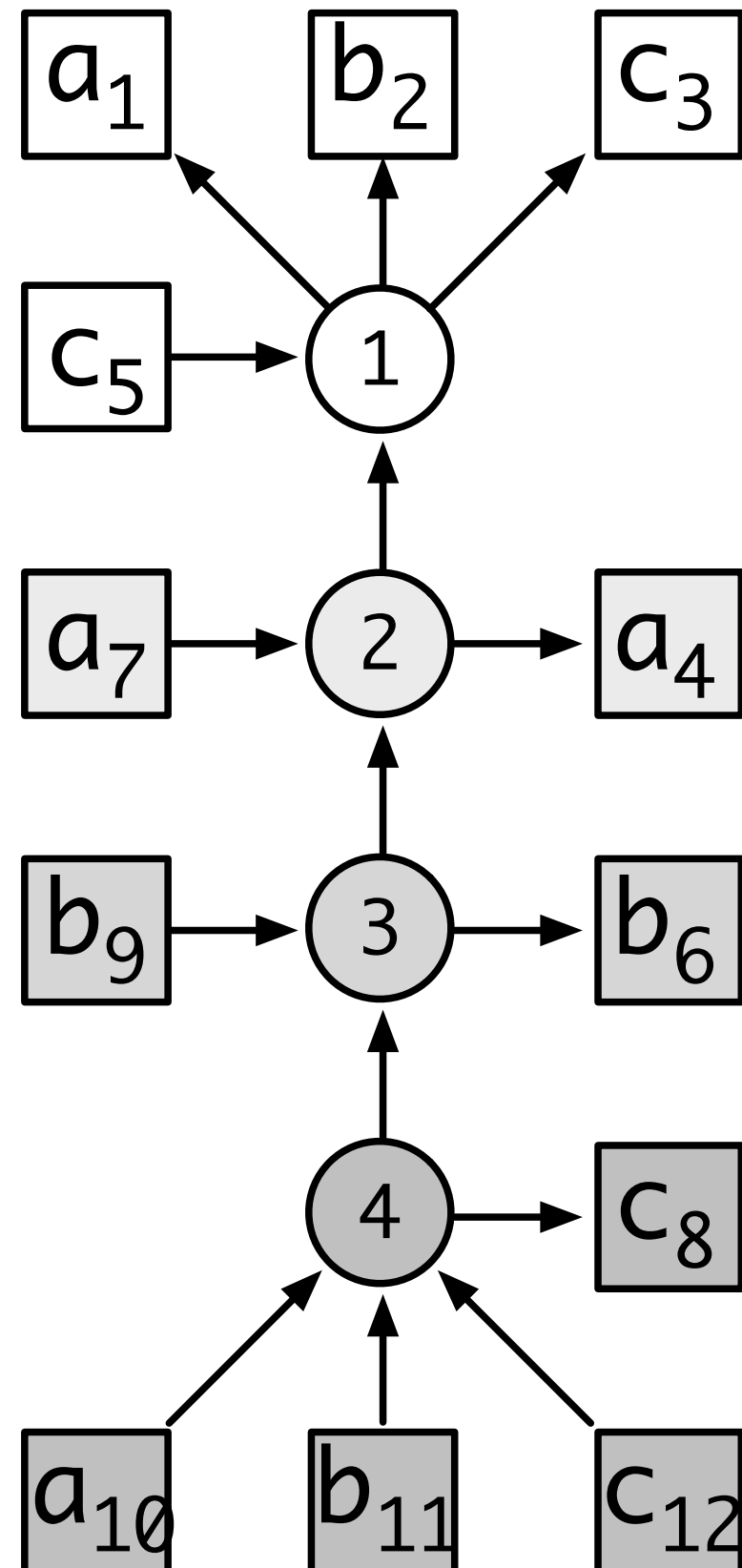
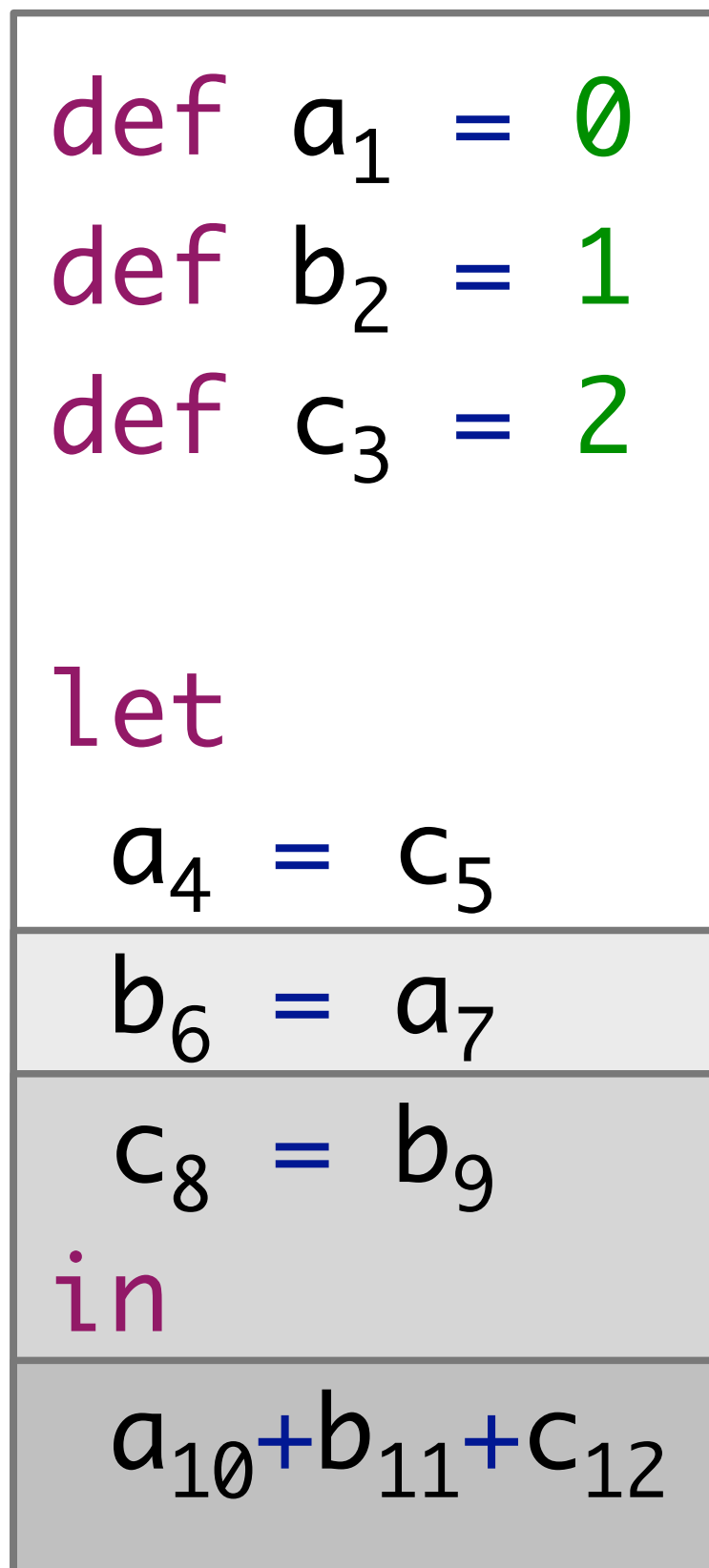
$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}, \mathbf{TI}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot (\mathbf{Inc} \mid \mathbf{TI})^* \cdot \mathbf{I}^?$$

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

More Examples

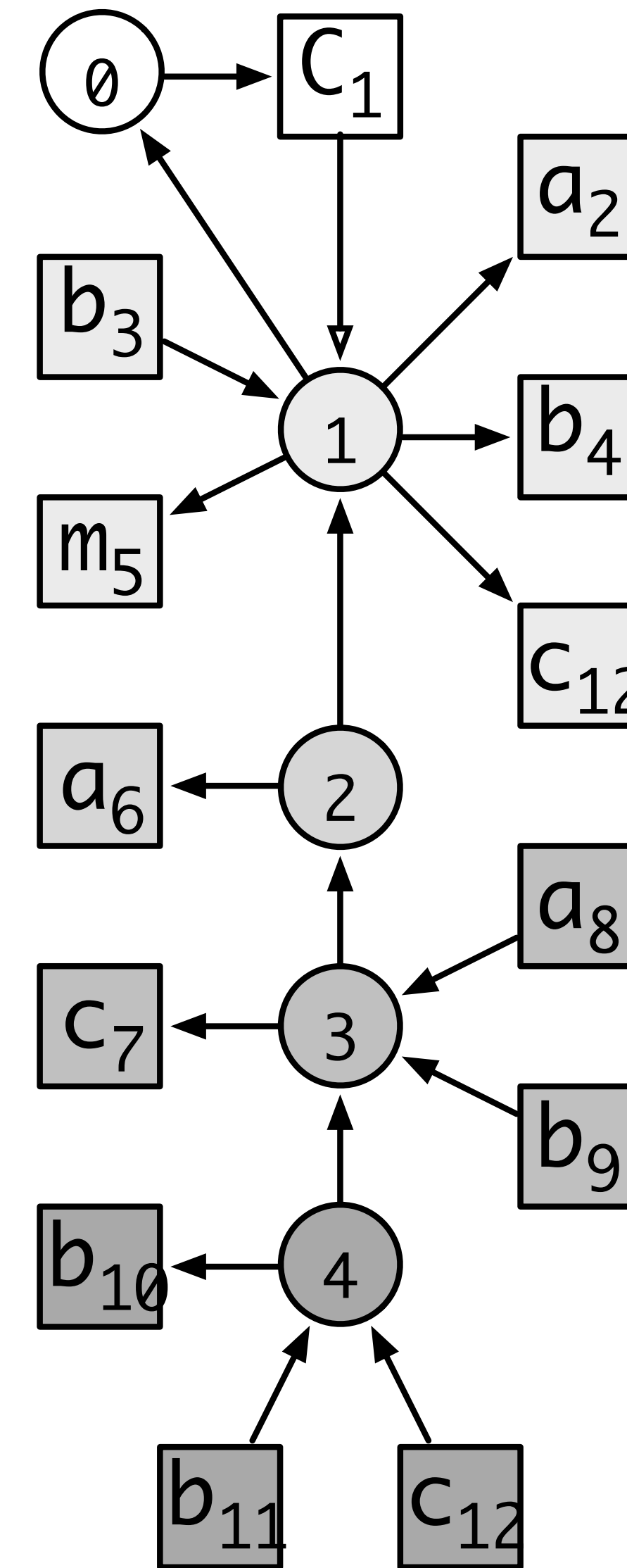
From TUD-SERG-2015-001

Let Bindings



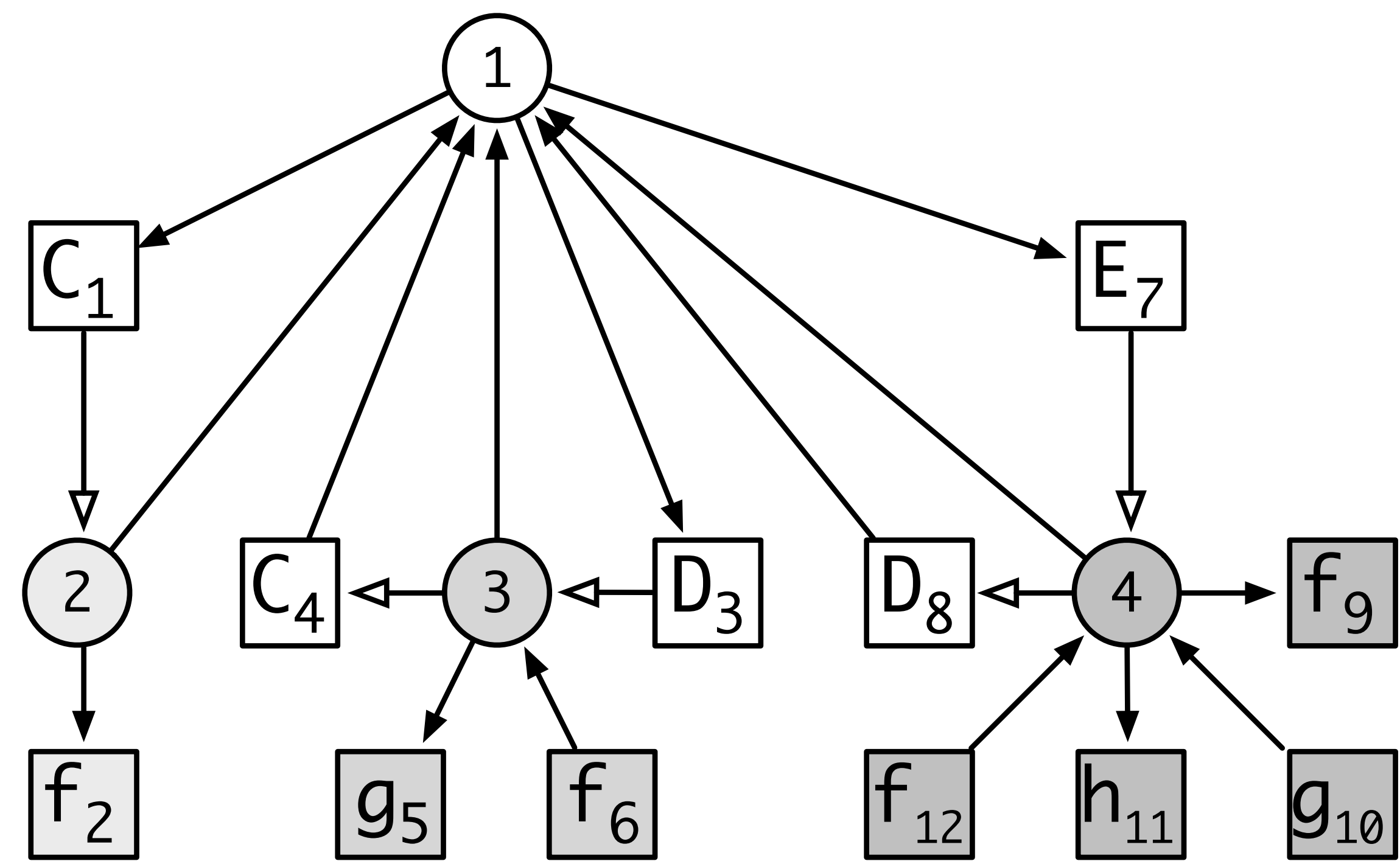
Definition before Use / Use before Definition

```
class C1 {  
  int a2 = b3;  
  int b4;  
  void m5 (int a6) {  
    int c7 = a8 + b9;  
    int b10 = b11 + c12;  
  }  
  int c12;  
}
```

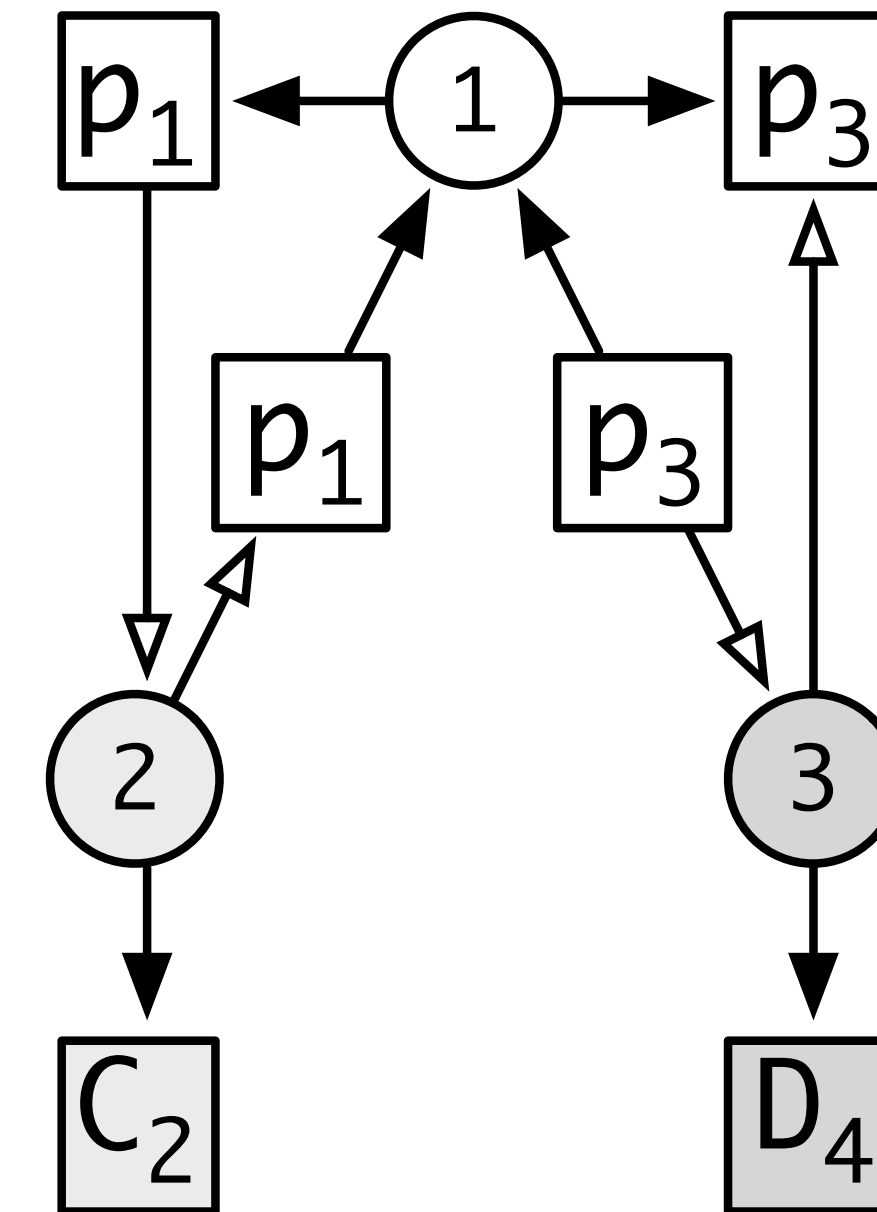
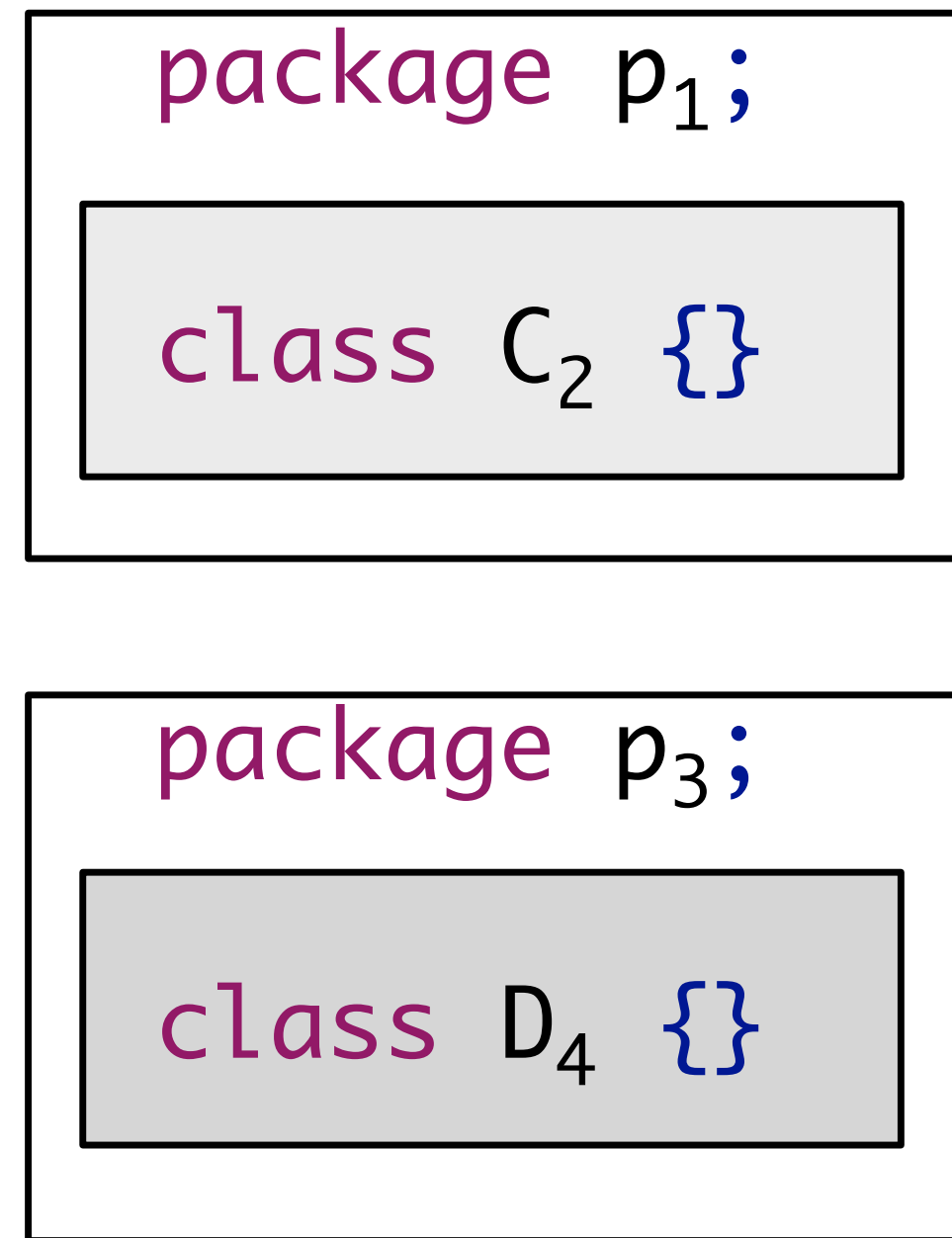


Inheritance

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

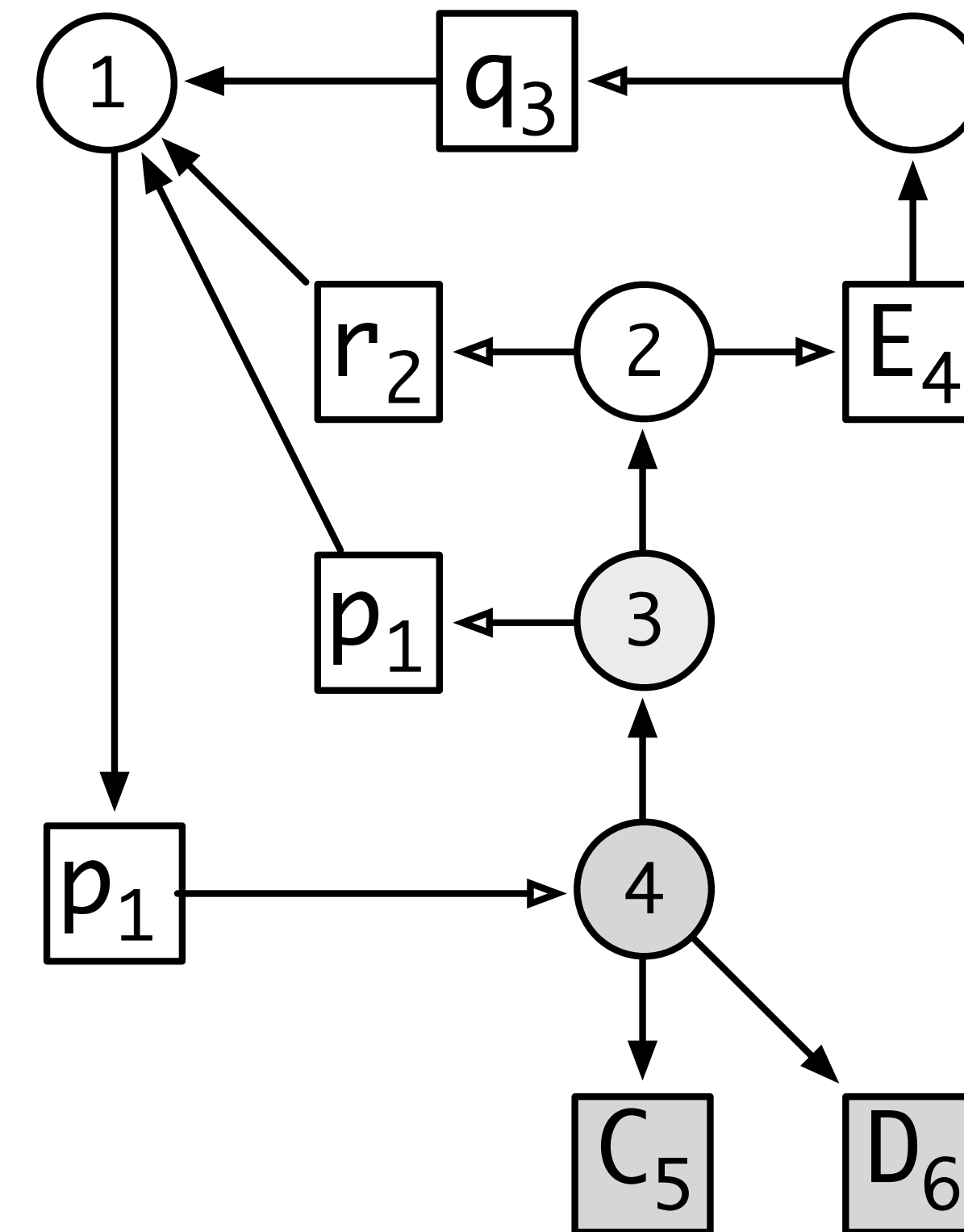


Java Packages



Java Import

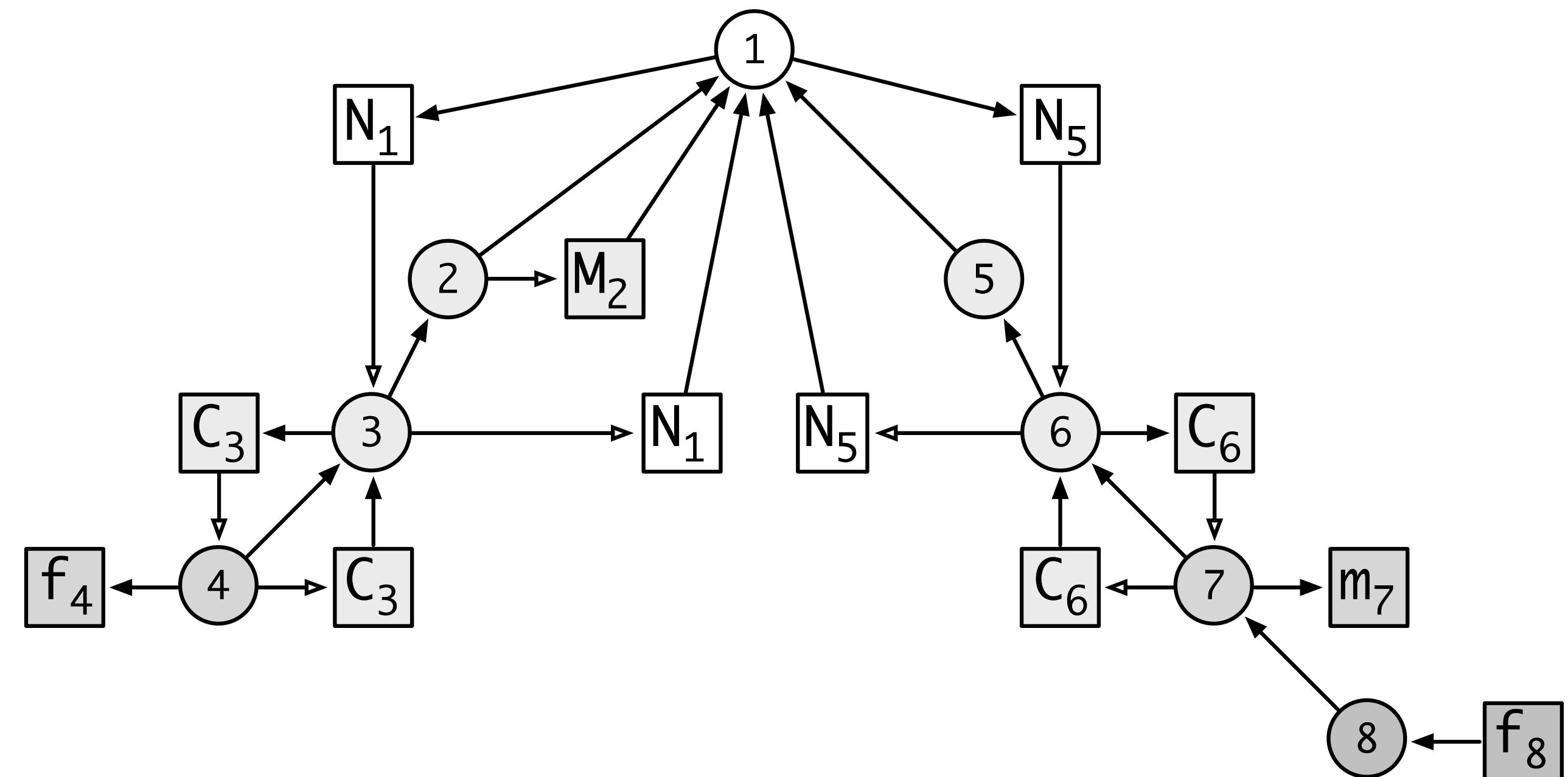
```
package p1;  
  
imports r2.*;  
imports q3.E4;  
  
public class C5 {}  
  
class D6 {}
```



C# Namespaces and Partial Classes

```
namespace N1 {  
    using M2;  
    partial class C3 {  
        int f4;  
    }  
}
```

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



Summary

Static Analysis

Static analysis

- Properties that can be checked of the ('static') program text
- Decide which properties to encode in syntax definition vs checker
- Strict vs liberal syntax

Context-sensitive properties

- Some properties are intrinsically context-sensitive
- In particular: names

Declarative specification of name binding rules

- Common (language agnostic) understanding of name binding

A Theory of Name Resolution

Representation: Scope Graphs

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

Formalism: Name Binding Constraints

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

Language-Independent Interpretation

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

Validation

We have modeled a large set of example binding patterns

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

Next goal: fully model some real languages

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

Ongoing/Future Work

Scope graph semantics for binding languages [OOPSLA18]

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

Dynamic analogs to static scope graphs [ECOOP16]

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

Supporting mechanized language meta-theory [POPL18]

- relating static and dynamic bindings

Resolution-sensitive program transformations

- renaming, refactoring, substitution, ...

Next

Separation of Concerns in Name Binding

Representation

- To conduct and represent the results of name resolution

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- ...

Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- ...

Separation of Concerns in Name Binding

Representation

- Scope Graphs

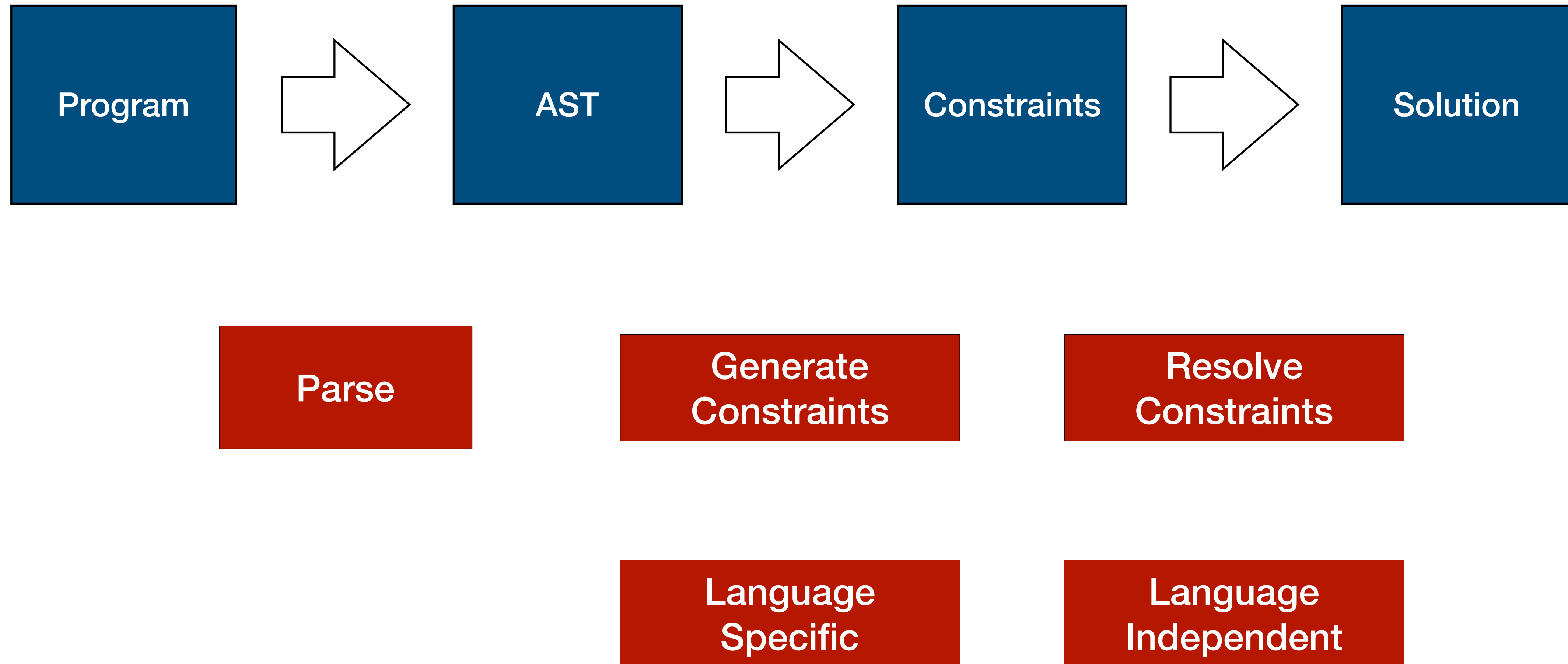
Declarative Rules

- Scope & Type Constraint Rules [PEPM16/OOPSLA18]

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- ...

Next: Constraint-Based Type Checkers



Except where otherwise noted, this work is licensed under

