Lecture 6: Introduction to Static Analysis

Eelco Visser

CS4200 Compiler Construction
TU Delft
October 2018

This Lecture

Check that names are used correctly and that expressions are well-typed

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.

Type checkers are algorithms that check names and types in programs.

This paper introduces the NaBL Name Binding Language, which supports the declarative definition of the name binding and scope rules of programming languages through the definition of scopes, declarations, references, and imports associated.

Background

SLE 2012

http://dx.doi.org/10.1007/978-3-642-36089-3_18

Declarative Name Binding and Scope Rules

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser

Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl,
{l.c.l.kats,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. In textual software languages, names are used to reference elements like variables, methods, classes, etc. Name resolution analyses these names in order to establish references between definition and use sites of elements. In this paper, we identify recurring patterns for name bindings in programming languages and introduce a declarative meta-language for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes. Based on such declarative name binding specifications, we provide a language-parametric algorithm for static name resolution during compile-time. We discuss the integration of the algorithm into the Spoofax Language Workbench and show how its results can be employed in semantic editor services like reference resolution, constraint checking, and content completion.

1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity. One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*. Syntax definitions are a prominent example. With declarative formalisms such as EBNF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [15].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 311–331, 2013.

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 Neron[1], Andrew Tolmach[2], Eelco Visser[1], and Guido Wachsmuth[1]

[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.

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,

PEPM 2016

https://doi.org/10.1145/2847538.2847543

Documentation for NaBL2 at the metaborg.org website.

Spoofax

latest

Search docs

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

Note

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

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

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.

The future

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.

2475-1421/2018/11-ART114
https://doi.org/10.1145/3276484

Why Type Checking?

Why Type Checking? Some Discussion Points

Dynamically Typed vs Statically Typed

- Dynamic: type checking at run-time
- Static: type checking at compile-time (before run-time)

What does it mean to type check?

- Type safety: guarantee absence of run-time type errors

Why static type checking?

- Avoid overhead of run-time type checking
- Fail faster: find (type) errors at compile time
- Find all (type) errors: some errors may not be triggered by testing
- But: not all errors can be found statically (e.g. array bounds checking)

Context-Sensitive Properties

Homework Assignment: What is the Syntax of This Language?

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
 </book>
 <book>
  <title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
</catalogue>
```

```
<languages>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <language>
   <name>Stratego</name>
    <purpose>Transformation</purpose>
    <implementedin>SDF3</implementedin>
    <implementedin>Stratego</implementedin>
    <target>Java</target>
 </language>
</languages>
```

Syntax of Book Catalogues

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
 </book>
 <book>
  <title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
</catalogue>
```

```
context-free syntax

  Document.Catalogue = [
    <catalogue>
      [Book*]
    </catalogue>
  ]

  Book.Book = [
    <book>
      [Title]
      [Author]
      [Publisher]
    </book>
  ]

  …
```

Schema-specific syntax definition

A Generic Syntax of XML Documents

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
 </book>
 <book>
  <title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
</catalogue>
```

```
Doc(
  Node(
    "catalogue"
  , [ Node(
        "book"
      , [ Node("title", [Text(["Modern Compiler Implementation in ML"])], "title")
        , Node("author", [Text(["Andrew Appel"])], "author")
        , Node("publisher", [Text(["Cambridge"])], "publisher")
        ]
      , "book"
      )
    , Node(
        "book"
      , [ Node("title", [Text(["Parsing Schemata"])], "title")
        , Node("author", [Text(["Klaas Sikkel"])], "author")
        , Node("publisher", [Text(["Springer"])], "publisher")
        ]
      , "book"
      )
    ]
  , "catalogue"
  )
)
```

```
context-free start-symbols Document

sorts Tag Word
lexical syntax
  Tag  = [a-zA-Z][a-zA-Z0-9]*
  Word = ~[\<\>]+

lexical restrictions
  Word -/- ~[\<\>]

sorts Document Elem
context-free syntax

  Document.Doc = Elem

  Elem.Node = [
    <[Tag]>
      [Elem*]
    </[Tag]>
  ]

  Elem.Text = Word+ {longest-match}
```

A Generic Syntax of XML Documents

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
 </book>
 <book>
  <title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
</catalogue>
```

```
context-free start-symbols Document

sorts Tag Word
lexical syntax
  Tag  = [a-zA-Z][a-zA-Z0-9]*
  Word = ~[\<\>]+

lexical restrictions
  Word -/- ~[\<\>]

sorts Document Elem
context-free syntax

  Document.Doc = Elem

  Elem.Node = [
    <[Tag]>
      [Elem*]
    </[Tag]>
  ]

  Elem.Text = Word+ {longest-match}
```

What is the problem with this approach?

Generic Syntax is Too Liberal!

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

Year is not a valid element of book

Only books in catalogue

Book should have a title

Closing tag is not consistent with starting tag

```
context-free start-symbols Document

sorts Tag Word
lexical syntax
  Tag  = [a-zA-Z][a-zA-Z0-9]*
  Word = ~[\<\>]+

lexical restrictions
  Word -/- ~[\<\>]

sorts Document Elem
context-free syntax

  Document.Doc = Elem

  Elem.Node = [
    <[Tag]>
      [Elem*]
    </[Tag]>
  ]

  Elem.Text = Word+ {longest-match}
```

Context-Sensitive Properties

Context-free grammar is … context-free!

- Cannot express alignment

Languages have context-sensitive properties

How can we have our cake and eat it too?

- Generic (liberal) syntax
- Forbid programs/documents that are not well-formed

Checking Context-Sensitive Properties

Approach: Checking Context-Sensitive Properties

Generic (liberal) syntax

- Allow more programs/documents

Check properties on AST

- Reject programs/documents that are not well-formed

Checking Context-Context-Sensitive Properties in Spoofax

errors.xml

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

parse

```
Doc(
  Node(
    "catalogue"
  , [ Node(
        "book"
      , [ Node("title", [Text(["Modern Compiler Implementation in ML"])], "title")
        , Node("author", [Text(["Andrew Appel"])], "author")
        , Node("publisher", [Text(["Cambridge"])], "publisher")
        , Node("year", [], "year")
        ]
      , "book"
      )
    , Node(
        "language"
      , [ Node("name", [Text(["SDF3"])], "name")
        , Node("purpose", [Text(["Syntax Definition"])], "purpose")
        , Node("implementedin", [Text(["SDF3"])], "implementedin")
        ]
      , "language"
      )
    , Node(
        "book"
      , [ Node("author", [Text(["Klaas Sikkel"])], "author")
        , Node("publisher", [Text(["Springer"])], "publisher")
        ]
      , "book"
      )
    , Node("book", [], "koob")
    ]
```

```
rules // Analysis

  editor-analyze:
    (ast, path, project-path) -> (ast', error*, warning*, info*)
    with
      ast'       := <id> ast
    ; error*     := <collect-all(constraint-error)> ast'
    ; warning*   := <collect-all(constraint-warning)> ast'
    ; info*      := <collect-all(constraint-info)> ast'
```

errors

Check Violations using Constraint-Error Rules

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

Find all sub-terms that are not consistent with the context-sensitive rules

collect-all: type-unifying generic traversal

```
rules // Analysis

  editor-analyze:
    (ast, path, project-path) -> (ast', error*, warning*, info*)
    with
      ast'     := <id> ast
    ; error*   := <collect-all(constraint-error)> ast'
    ; warning* := <collect-all(constraint-warning)> ast'
    ; info*    := <collect-all(constraint-info)> ast'
```

Check Violations using Constraint-Error Rules

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

```
Closing tag does not match starting tag
```

Origin

Error message

```
rules

  constraint-error :
    Node(tag1, elems, tag2) -> (tag2, $[Closing tag does not match starting tag])
    where <not(eq)>(tag1, tag2)
```

Check Violations using Constraint-Error Rules

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

Book should have a title

Containment checks

```
rules

  constraint-error :
    n@Node(tag@"book", elems, _) -> (tag, $[Book should have title])
    where <not(has(|"title"))> elems

  has(|tag) = fetch(?Node(tag, _, _))
```

Check Violations using Constraint-Error Rules

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

Containment checks

```
Catalogue can only have books
```

```
rules

  constraint-error :
    Node("catalogue", elems, _) -> error
    where <filter(not-a-book)> elems => [error | _]

  not-a-book :
    Node(tag, _, _) -> (tag, $[Catalogue can only have books])
    where <not(eq)> (tag, "book")
```

Check Violations using Constraint-Error Rules

```
<catalogue>
 <book>
  <title>Modern Compiler Implementation in ML</title>
  <author>Andrew Appel</author>
  <publisher>Cambridge</publisher>
  <year></year>
 </book>
 <language>
  <name>SDF3</name>
    <purpose>Syntax Definition</purpose>
    <implementedin>SDF3</implementedin>
 </language>
 <book>
  //<title>Parsing Schemata</title>
  <author>Klaas Sikkel</author>
  <publisher>Springer</publisher>
 </book>
 <book>
 </koob>
</catalogue>
```

```
Book can only have title, author, publisher
```

Containment checks

```
rules

  constraint-error :
    Node("book", elems, _) -> error
    where <filter(not-a-book-elem)> elems => [error | _]

  not-a-book-elem :
    Node(tag, _, _) -> (tag, $[Book can only have title, author, publisher])
    where <not(elem())> (tag, ["title", "author", "publisher"])
```

Approach: Checking Context-Sensitive Properties

Generic (liberal) syntax

- Allow more programs/documents
- `permissive syntax’

Check properties on AST

- Reject programs/documents that are not well-formed

Advantage

- Smaller syntax definition
- Parser does not fail (so often)
- Better error messages than parser can give

How are programming languages different from XML?

Programming Languages vs XML

XML checking

- Tag consistency
- Schema consistency
- These are structural properties

Programming languages

- Type consistency (similar to schema?)
- Name consistency: declarations and references should correspond
- Name dependent type consistency: names carry types

Static Analysis for Tiger

Scope

```
let
  var x : int := 0 + z
  var y : int := x + 1
  var z : int := x + y + 1
 in
   x + y + z
end
```

Scope: Definition before Use

```
let
  var x : int := 0 + z  // z not in scope
  var y : int := x + 1
  var z : int := x + y + 1
 in
   x + y + z
end
```

Mutual Recursion

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

Mutually Recursive Functions should be Adjacent

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

```
let
  function odd(x : int) : int =
    if x > 0 then even(x - 1) else false
  var x : int
  function even(x : int) : int =
    if x > 0 then odd(x - 1) else true
 in
   even(34)
end
```

Name Spaces

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
 in foo(56) + foo
end
```

Functions and Variables in Same Name Space

```
let
  type foo = int
  function foo(x : foo) : foo = 3
  var foo : foo := foo(4)
 in foo(56) + foo // both refer to the variable foo
end
```

Functions and variables are in the same namespace

Type Dependent Name Resolution

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

Type Dependent Name Resolution

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

```
Resolving origin.x requires the type of origin
```

Name Correspondence

```
let
  type point = {x : int, y : int}
  type errpoint = {x : int, x : int}
  var p : point
  var e : errpoint
 in
   p := point{ x = 3, y = 3, z = "a" }
   p := point{ x = 3 }
end
```

Name Set Correspondence

Duplicate Declaration of Field “x”

```
let
  type point = {x : int, y : int}
  type errpoint = {x : int, x : int}
  var p : point
  var e : errpoint
 in
   p := point{ x = 3, y = 3, z = "a" }
   p := point{ x = 3 }
end
```

Field “y” not initialized

Reference “z” not resolved

Recursive Types

```
let
  type intlist = {hd : int, tl : intlist}
  type tree = {key : int, children : treelist}
  type treelist = {hd : tree, tl : treelist}
  var l : intlist
  var t : tree
  var tl : treelist
 in
  l := intlist { hd = 3, tl = l };
  t := tree {
    key = 2,
    children = treelist {
      hd = tree{ key = 3, children = 3 },
      tl = treelist{ }
    }
  };
  t.children.hd.children := t.children
end
```

Recursive Types

```
let
  type intlist = {hd : int, tl : intlist}
  type tree = {key : int, children : treelist}
  type treelist = {hd : tree, tl : treelist}
  var l : intlist
  var t : tree
  var tl : treelist
 in
  l := intlist { hd = 3, tl = l };
  t := tree {
    key = 2,
    children = treelist {
      hd = tree{ key = 3, children = 3 },
      tl = treelist{ }
    }
  };
  t.children.hd.children := t.children
end
```

type mismatch

Field "tl" not initialized
Field "hd" not initialized

Intermezzo: Testing Static Analysis

Testing Name Resolution

```
test outer name [[
   let type t = u
       type [[u]] = int
       var x: [[u]] := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := 42
      end
end
]] resolve #2 to #1
```

```
test inner name [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type [[u]] = t
          var y: [[u]] := 0
      in
         y := 42
      end
end
]] resolve #2 to #1
```

Testing Type Checking

```
test integer constant [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[42]]
      end
end
]] run get-type to INT()
```

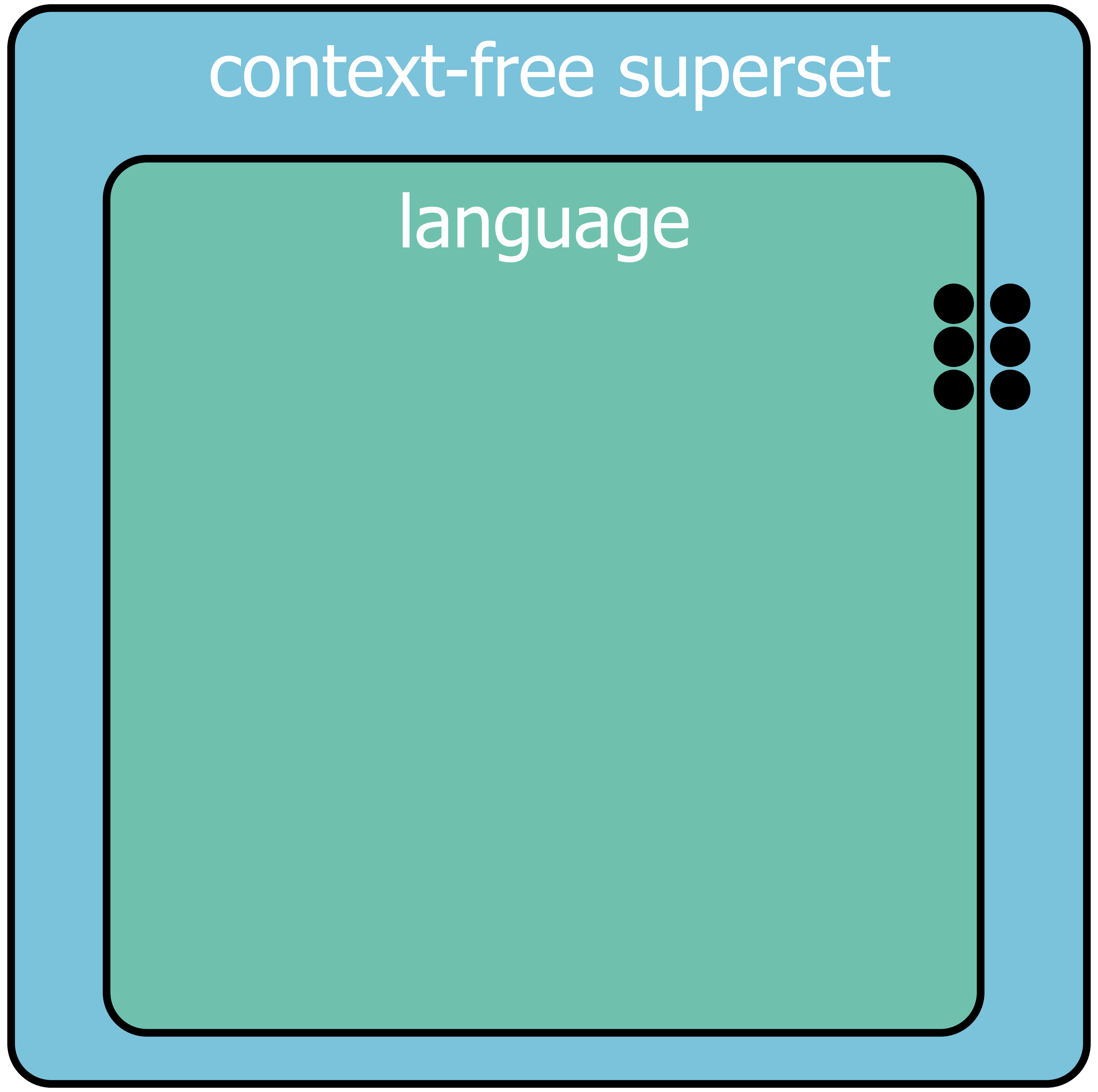
```
test variable reference [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[x]]
      end
end
]] run get-type to INT()
```

Testing Errors

```
test undefined variable [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[z]]
      end
end
]] 1 error
```

```
test type error [[
   let type t = u
       type u = string
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[x]]
      end
end
]] 1 error
```

Test Corner Cases



Implementing a Type Checker with Rewrite Rules

TU Delft

Compute Type of Expression

```
rules

  type-check :
    Mod(e) -> t
    where <type-check> e => t

  type-check :
    String(_) -> STRING()

  type-check :
    Int(_) -> INT()

  type-check :
    Plus(e1, e2) -> INT()
    where
      <type-check> e1 => INT();
      <type-check> e2 => INT()

  type-check :
    Times(e1, e2) -> INT()
    where
      <type-check> e1 => INT();
      <type-check> e2 => INT()
```

```
1 + 2 * 3
```

```
Mod(
  Plus(Int("1"),
       Times(Int("2"),
             Int("3")))
)
```

```
INT()
```

Compute Type of Variable?

```
rules

  type-check :
    Let([VarDec(x, e)], e_body) -> t
    where
      <type-check> e => t_e;
      <type-check> e_body => t

  type-check :
    Var(x) -> t // ???
```

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

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

?

Type Checking Variable Bindings

```
rules

  type-check(|env) :
    Let([VarDec(x, e)], e_body) -> t
    where
      <type-check(|env)> e => t_e;
      <type-check(|[(x, t_e) | env])> e_body => t

  type-check(|env) :
    Var(x) -> t
    where
      <fetch(?(x, t))> env
```

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

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

```
INT()
```

Store association between variable and type in type environment

Pass Environment to Sub-Expressions

```
rules

  type-check :
    Mod(e) -> t
    where <type-check(|[])> e => t

  type-check(|env) :
    String(_) -> STRING()

  type-check(|env) :
    Int(_) -> INT()

  type-check(|env) :
    Plus(e1, e2) -> INT()
    where
      <type-check(|env)> e1 => INT();
      <type-check(|env)> e2 => INT()

  type-check(|env) :
    Times(e1, e2) -> INT()
    where
      <type-check(|env)> e1 => INT();
      <type-check(|env)> e2 => INT()
```

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

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

```
INT()
```

But what about?

Type checking ill-typed/named programs?

- add rules for ‘bad’ cases

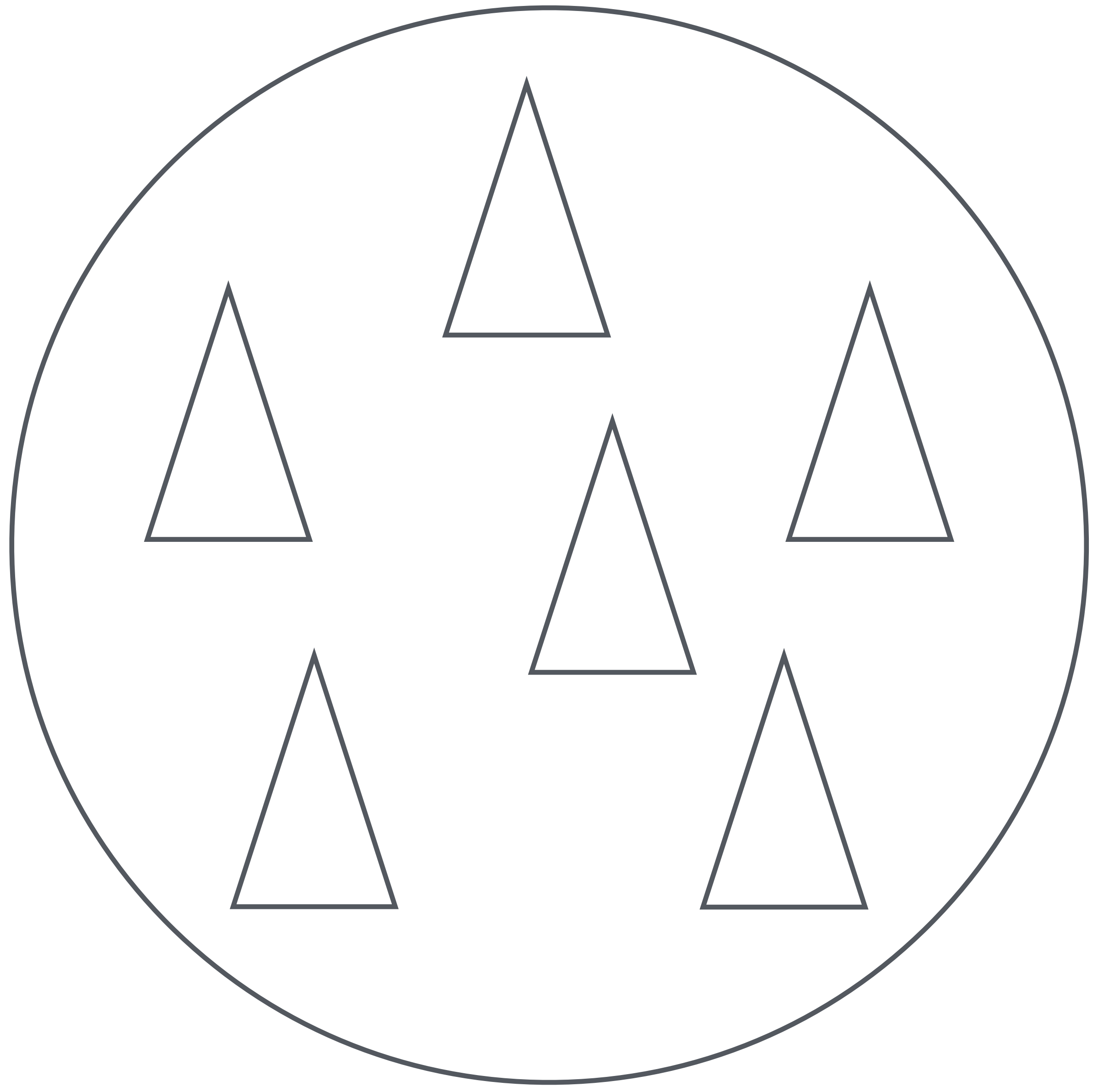
More complicated name binding patterns?

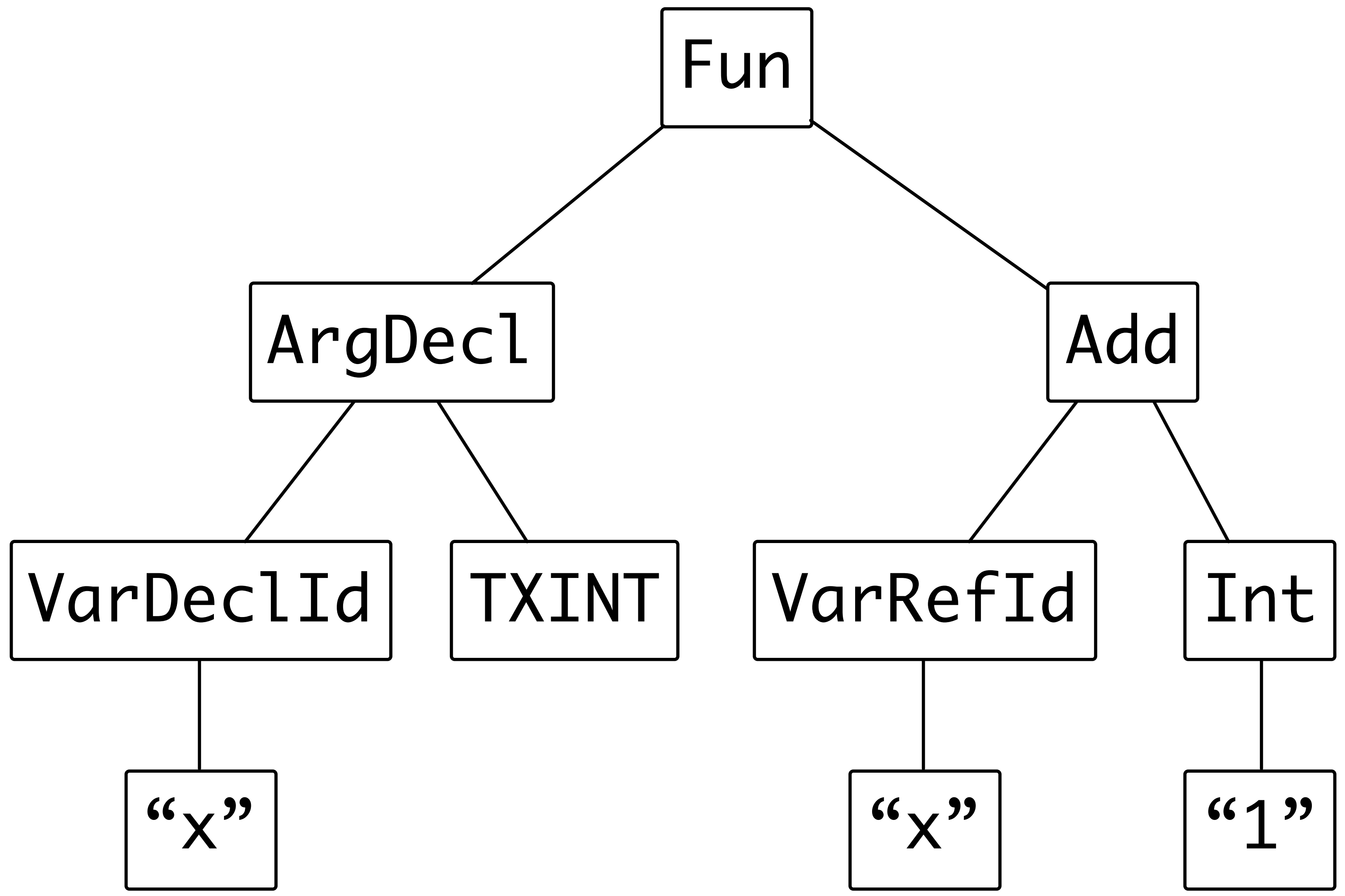
- Hoisting of variables in JavaScript functions
- Mutually recursive bindings
- Possible approaches
  - Multiple traversals over program
  - Defer checking until entire scope is processed
  - …

Name Binding Complicates Type Checking

Name Binding

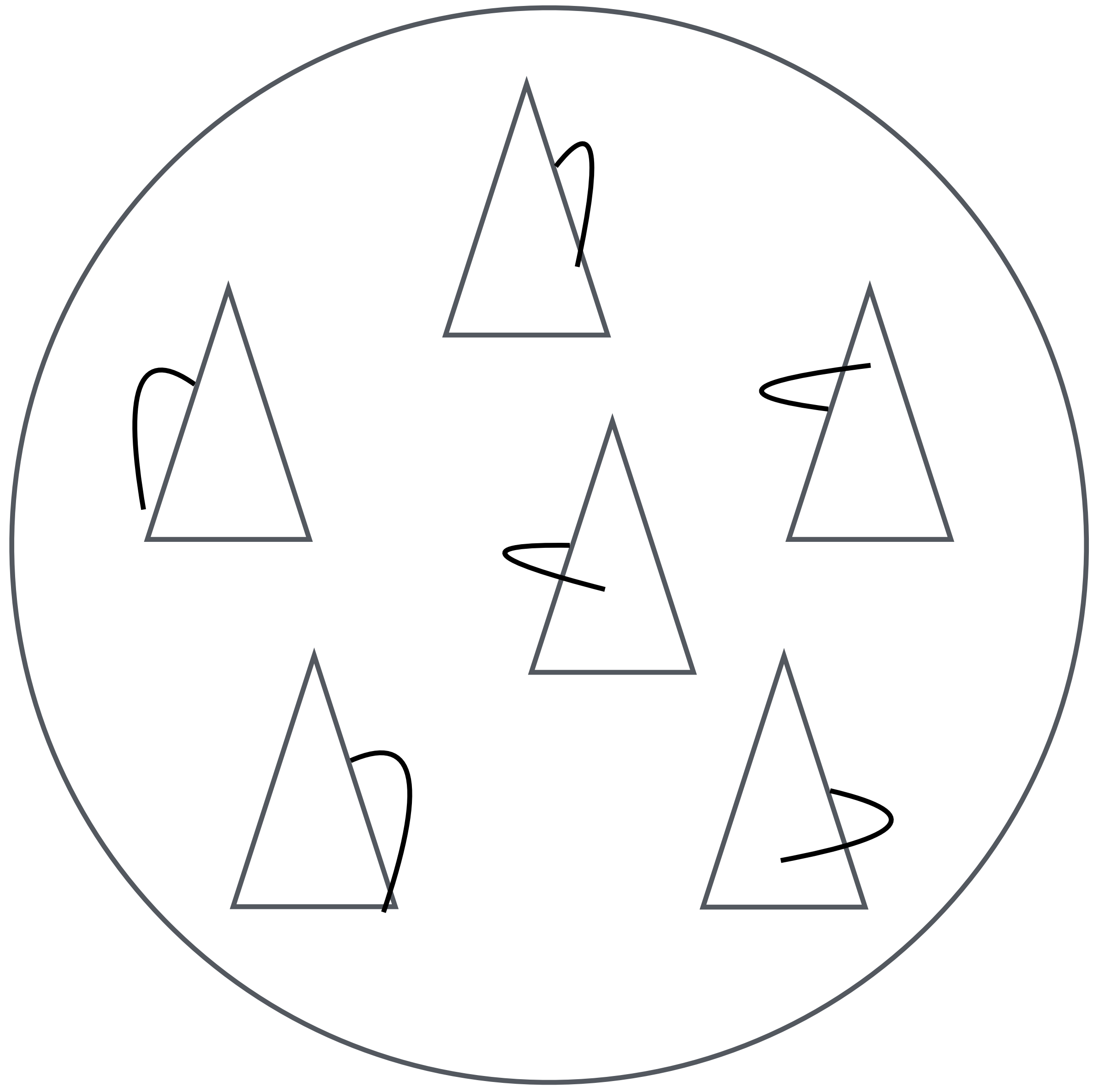
Language = Set of Trees

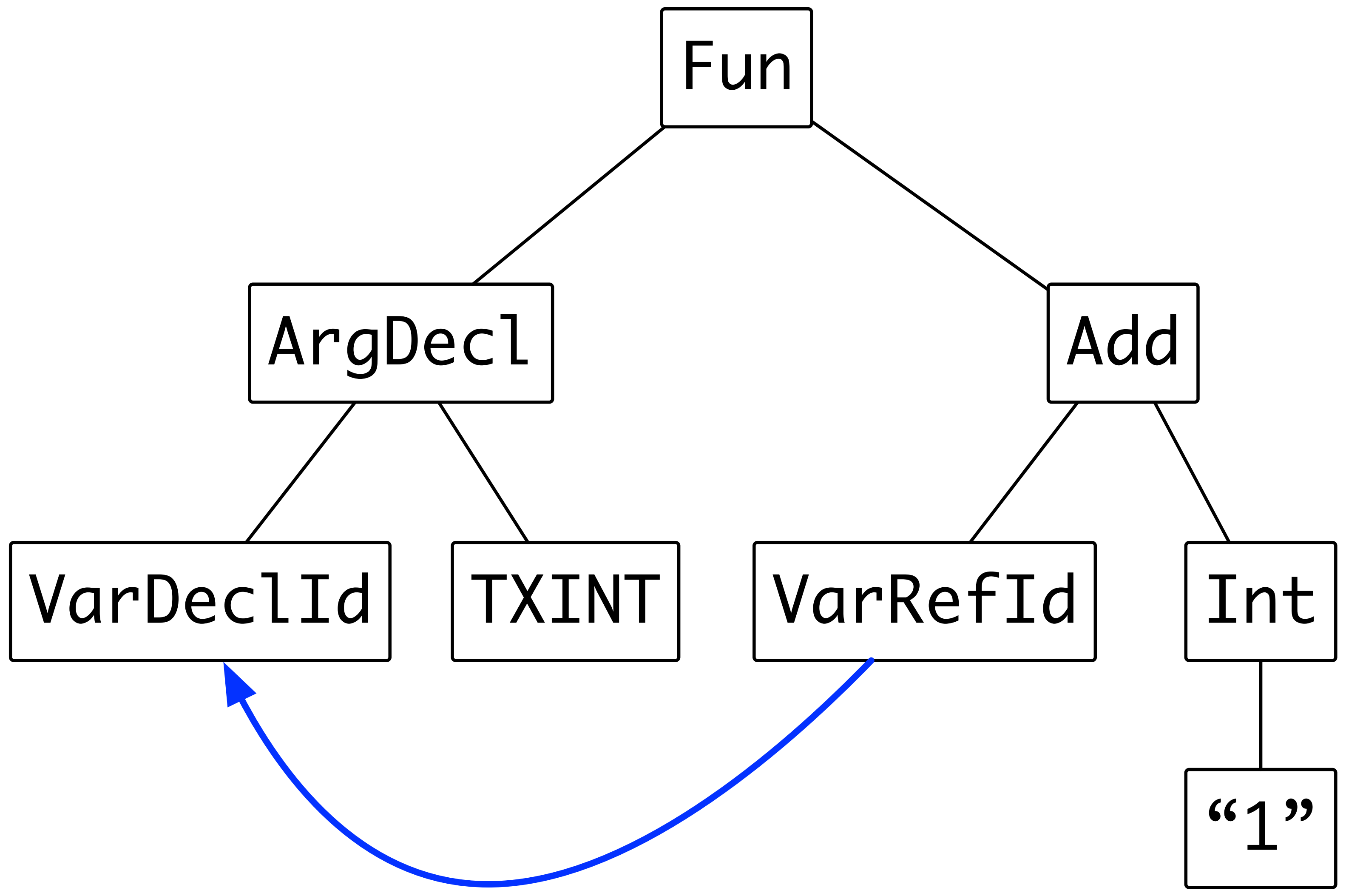




Tree is a convenient interface for transforming programs

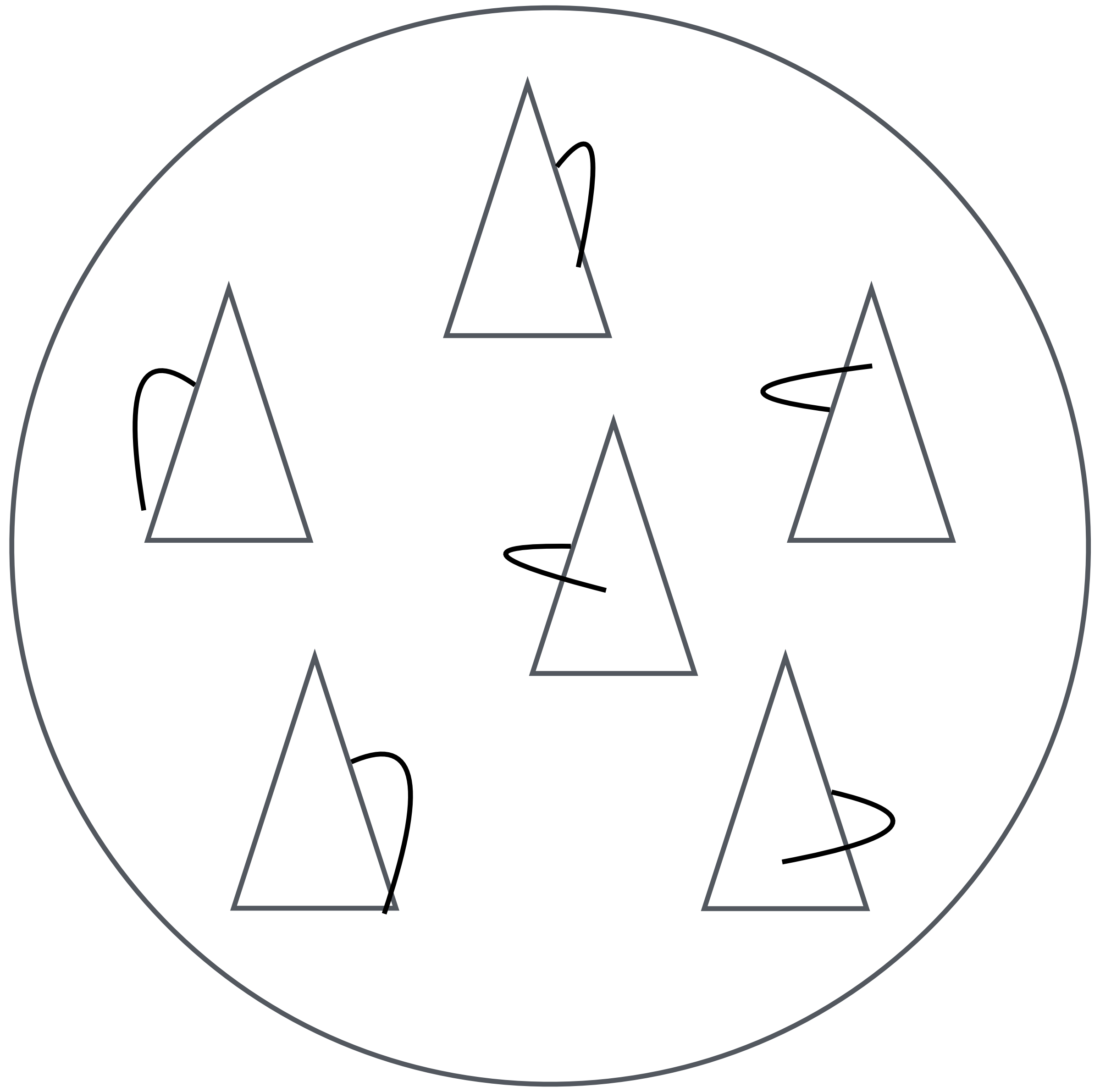
Language = Set of *Graphs*

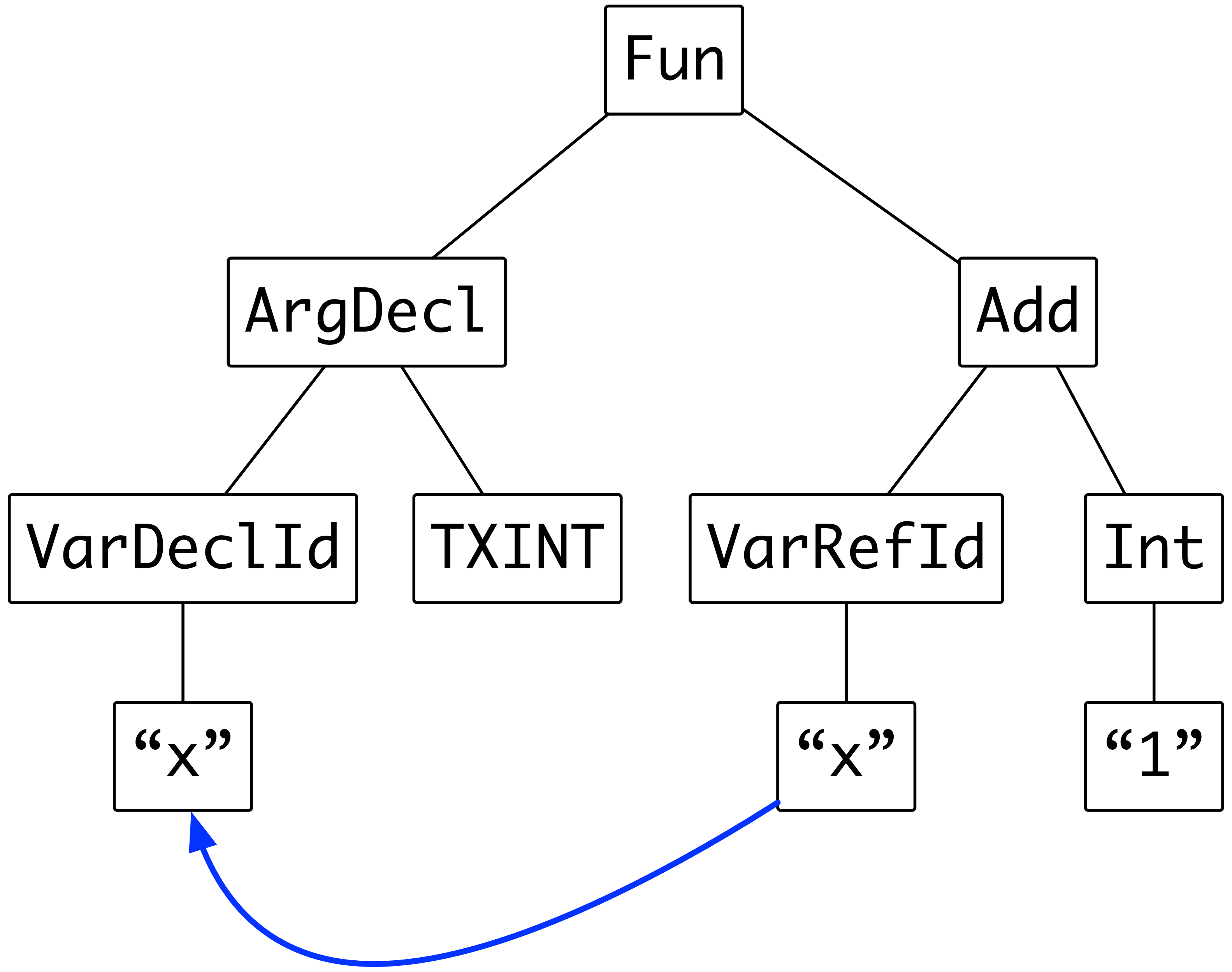




Edges from references to declarations

Language = Set of *Graphs*





Names are placeholders for edges in linear / tree representation

What are the commonalities of name binding rules in programming languages?

Name Binding Patterns

Variables

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

Function Calls

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

Nested Scopes

(Shadowing)

```
function prettyprint(tree: tree) : string =
let
  var output := ""

  function write(s: string) =
    output := concat(output, s)

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

 in show(0, tree);
    output
end
```

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

Type References

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

Record Fields

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

Type Dependent Name Resolution

Name Binding is Pervasive

Used in many different language artifacts

– compiler, interpreter, semantics, IDE, refactoring

Binding rules encoded in many different and ad-hoc ways

– symbol tables, environments, substitutions

No standard approach to formalization

– there is no BNF for name binding

No reuse of binding rules between artifacts

– how do we know substitution respects binding rules?

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?

Declarative Specification of Name Binding Rules

Separation of Concerns in Definition of Programming Languages

Representation

- Standardized representation for <aspect> of programs
- Independent of specific object language

Specification Formalism

- Language-specific declarative rules
- Abstract from implementation concerns

Language-Independent Interpretation

- Formalism interpreted by language-independent algorithm
- Multiple interpretations for different purposes
- Reuse between implementations of different languages

Separation of Concerns in Syntax Definition

Representation: (Abstract Syntax) Trees

- Standardized representation for structure of programs
- Basis for syntactic and semantic operations

Formalism: Syntax Definition

- Productions + Constructors + Templates + Disambiguation
- Language-specific rules: structure of each language construct

Language-Independent Interpretation

- Well-formedness of abstract syntax trees
  - provides declarative correctness criterion for parsing
- Parsing algorithm

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
- …

Name Binding Languages

DSLs for specifying binding structure of a (target) language

- Ott [Sewell+10]
- Romeo [StansiferWand14]
- Unbound [Weirich+11]
- Caml [Pottier06]
- NaBL [Konat+12]

Generate code to do resolution and record results

What is the semantics of such a name binding language?

Attempt: NaBL Name Binding Language

```
binding rules // variables

  Param(t, x) :
    defines Variable x of type t

  Let(bs, e) :
    scopes Variable

  Bind(t, x, e) :
    defines Variable x of type t

  Var(x) :
    refers to Variable x
```

Declarative specification

Abstracts from implementation

Incremental name resolution

But:

How to explain it to Coq?

What is the semantics of NaBL?

Declarative Name Binding and Scope Rules

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser

SLE 2012

Attempt: NaBL Name Binding Language

```
binding rules // classes

  Class(c, _, _, _) :
   defines Class c of type ClassT(c)
   scopes Field, Method, Variable

  Extends(c) :
   imports Field, Method from Class c

  ClassT(c) :
   refers to Class c

  New(c) :
   refers to Class c
```

Especially:

What is the semantics of imports?

Declarative Name Binding and Scope Rules

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, Eelco Visser

SLE 2012

Approach

What is semantics of binding language?

- 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)

That supports

- 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

Name Resolution in Scope Graphs

Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- To define name binding rules of a language

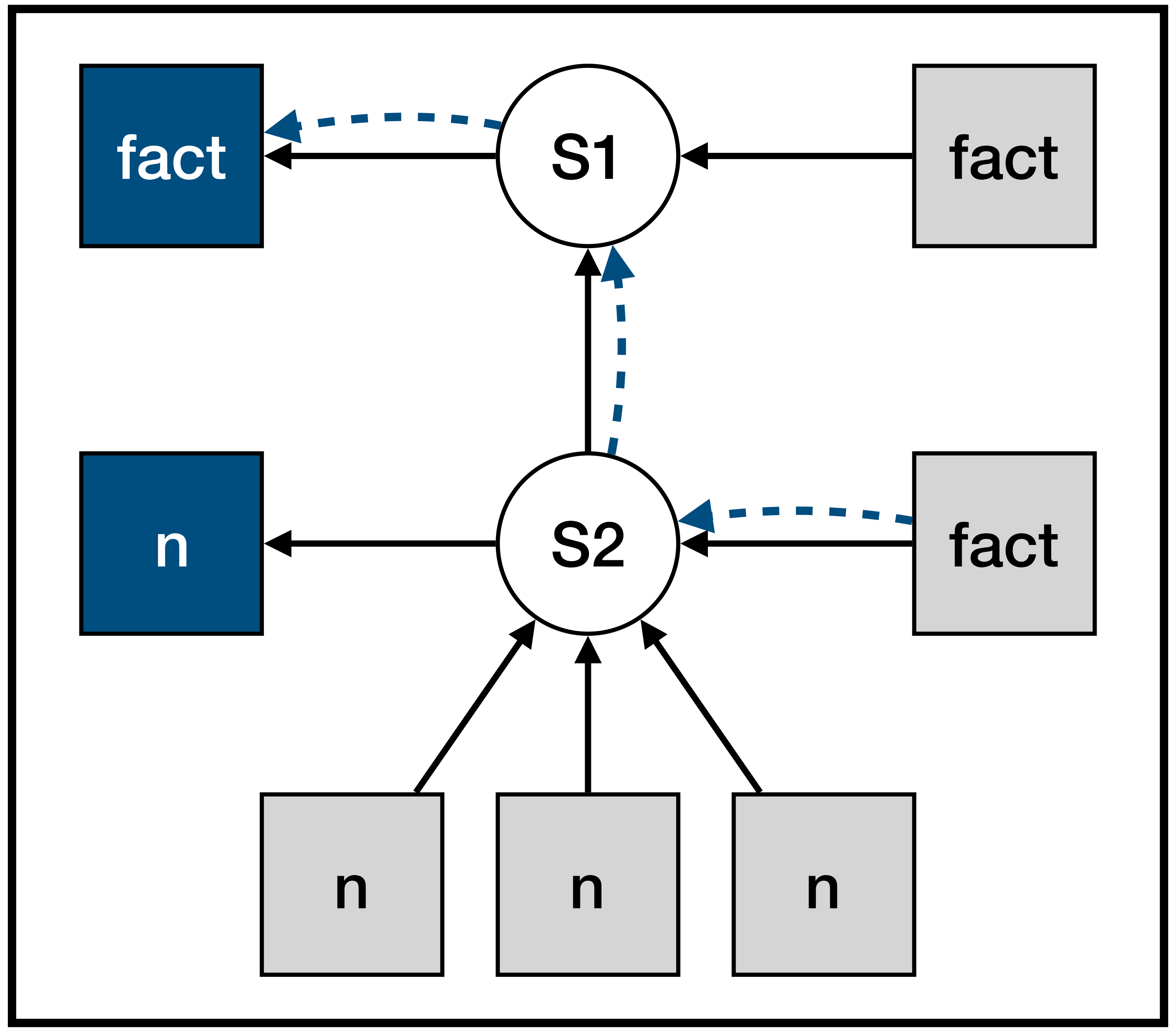
Language-Independent Tooling

- 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
```

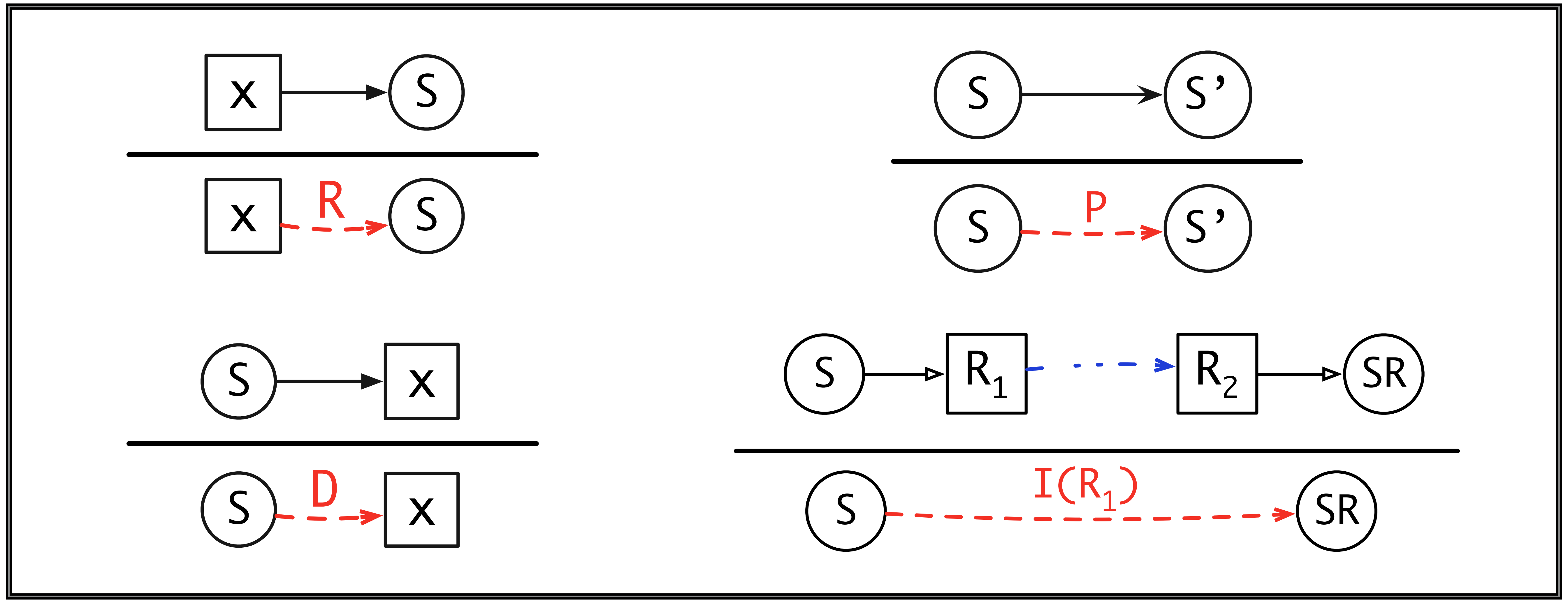
Scope Graph



Name Resolution

A Calculus for Name Resolution

Scopes, References, Declarations, Parents, Imports



Path in scope graph connects reference to declaration

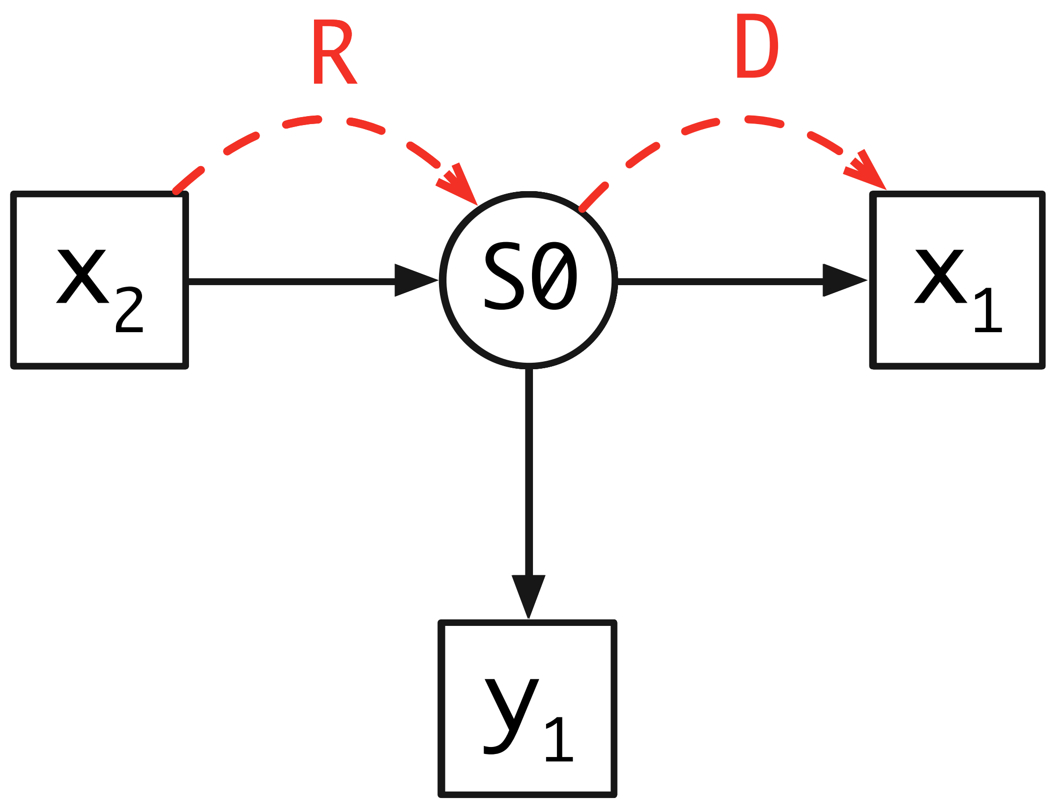
Neron, Tolmach, Visser, Wachsmuth
A Theory of Name Resolution
ESOP 2015

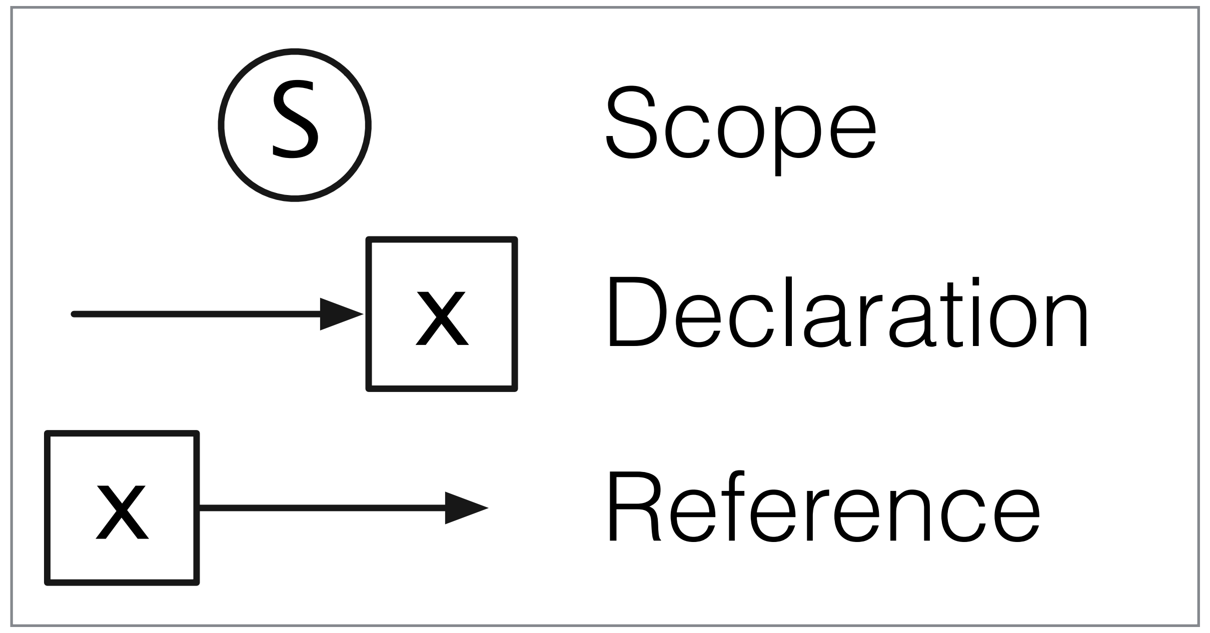
Simple Scopes

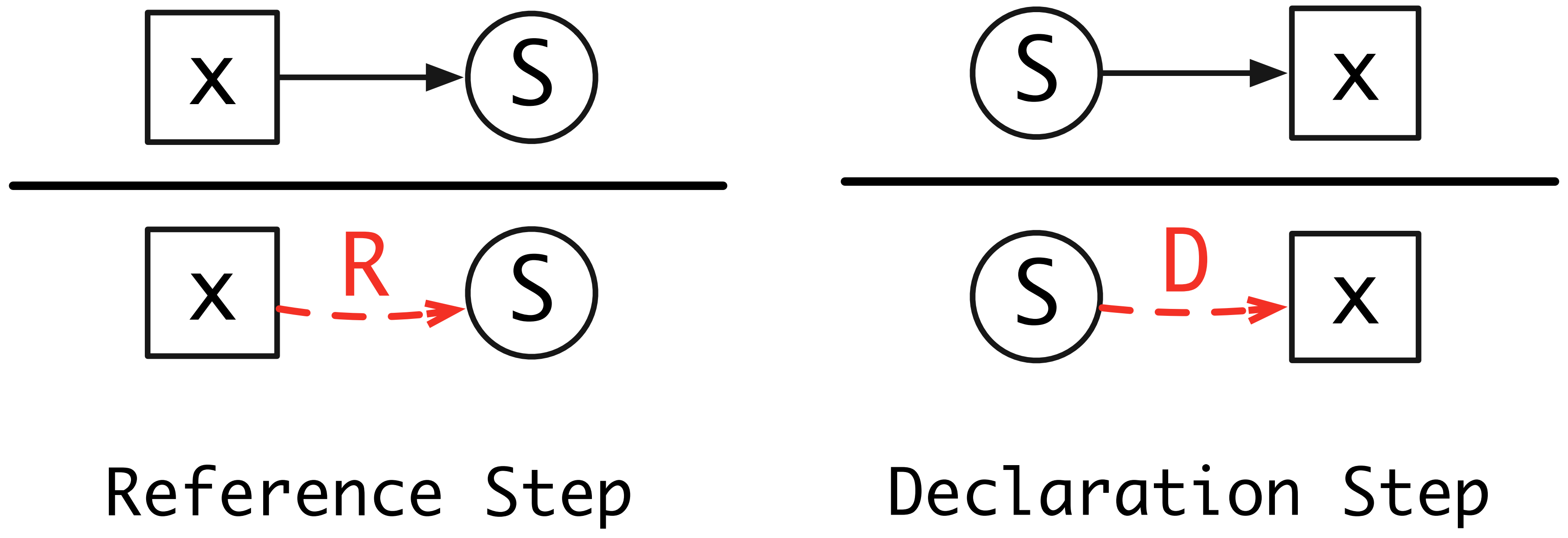
S0

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

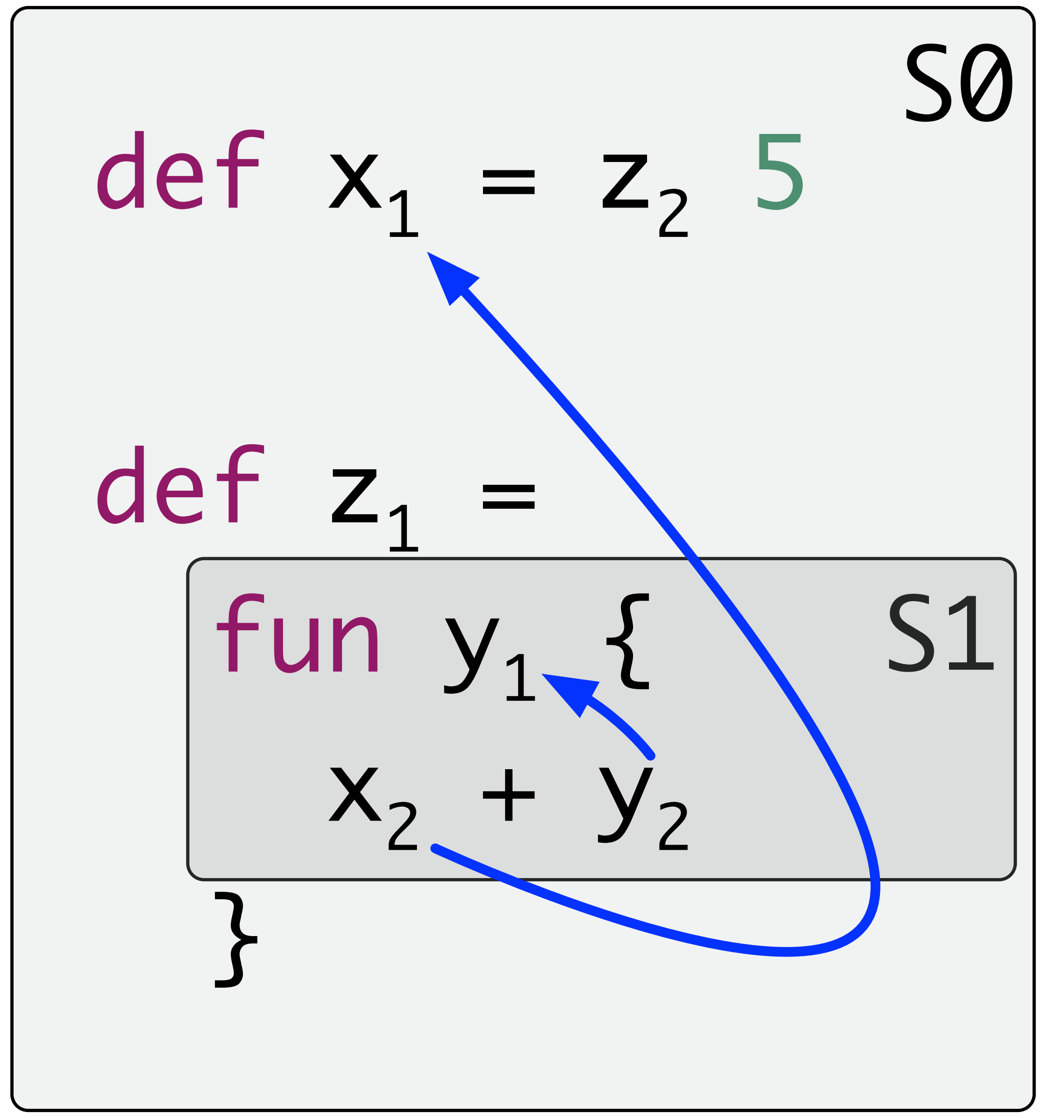
$\mathtt{def}\ x_1 = 5$

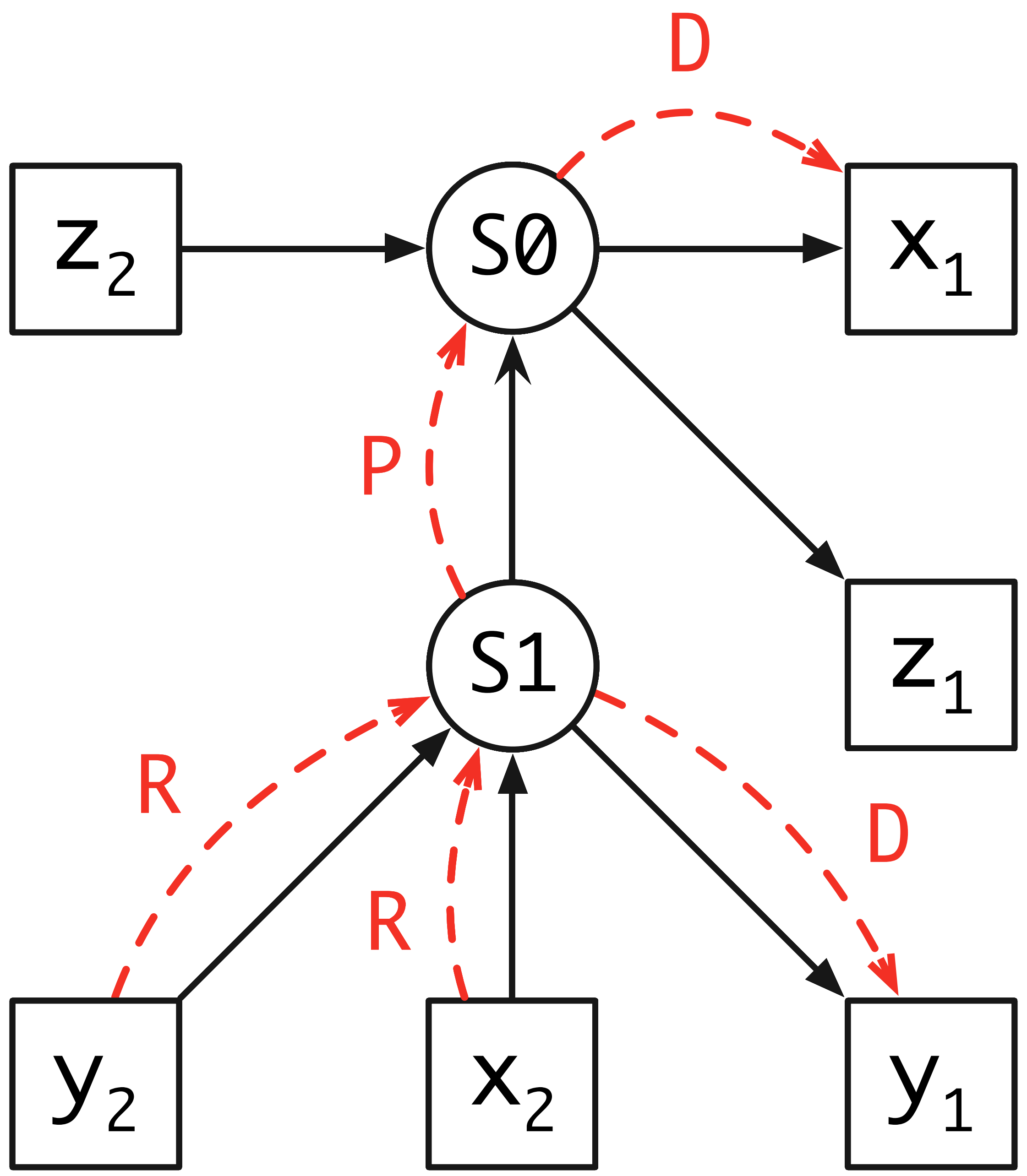


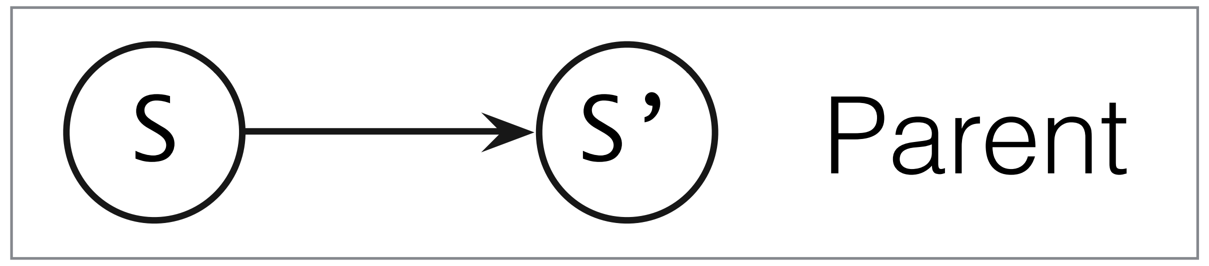


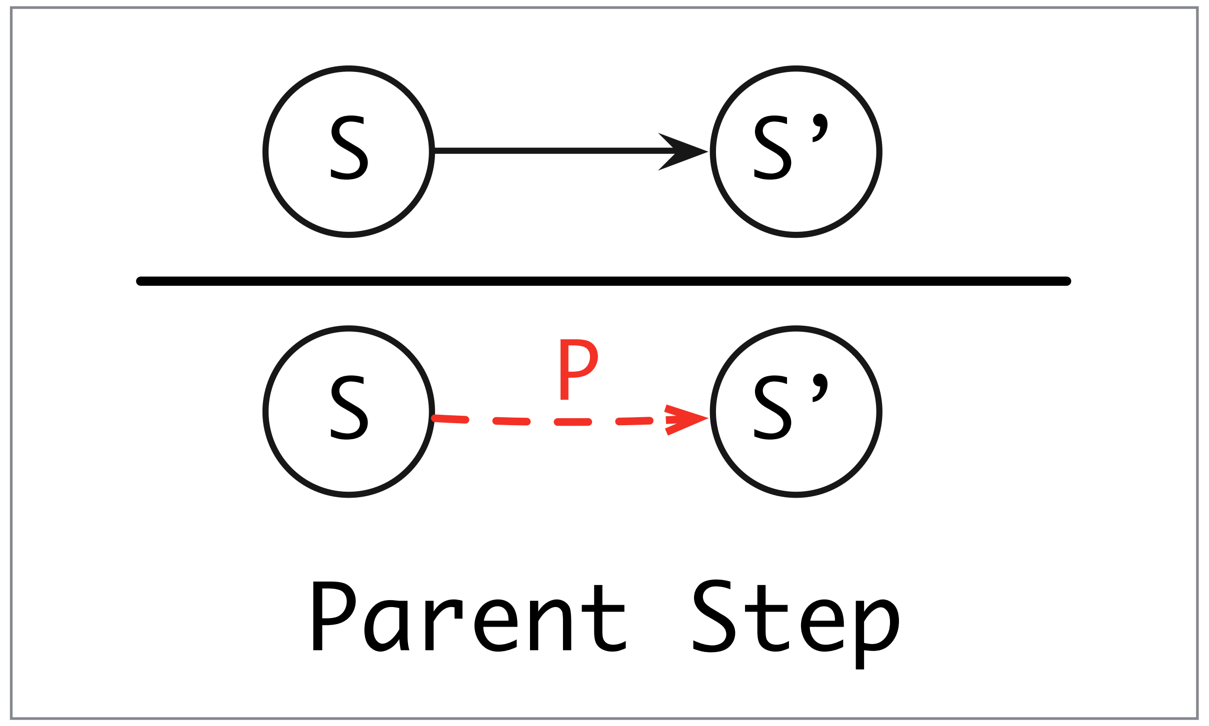


Lexical Scoping



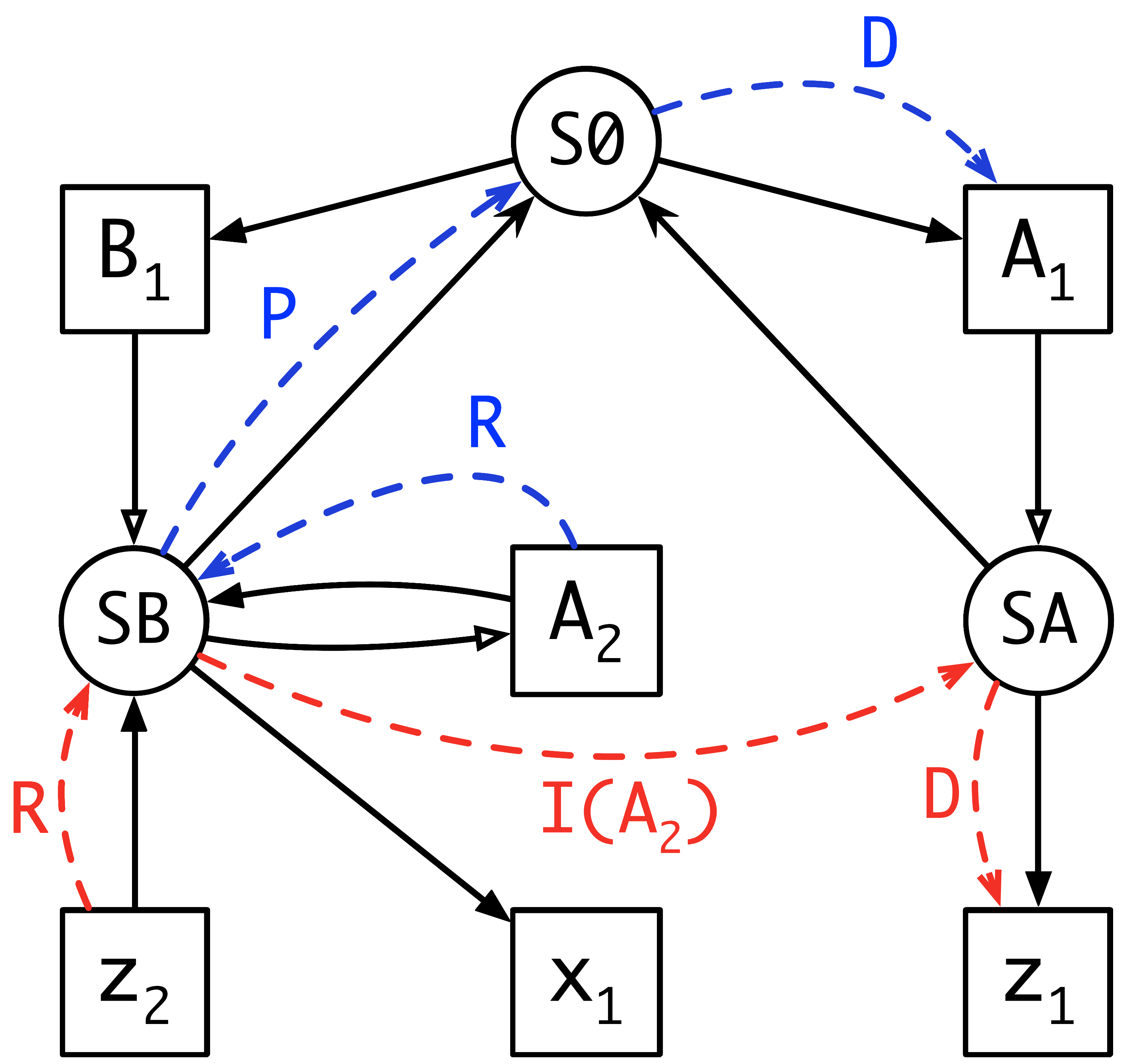


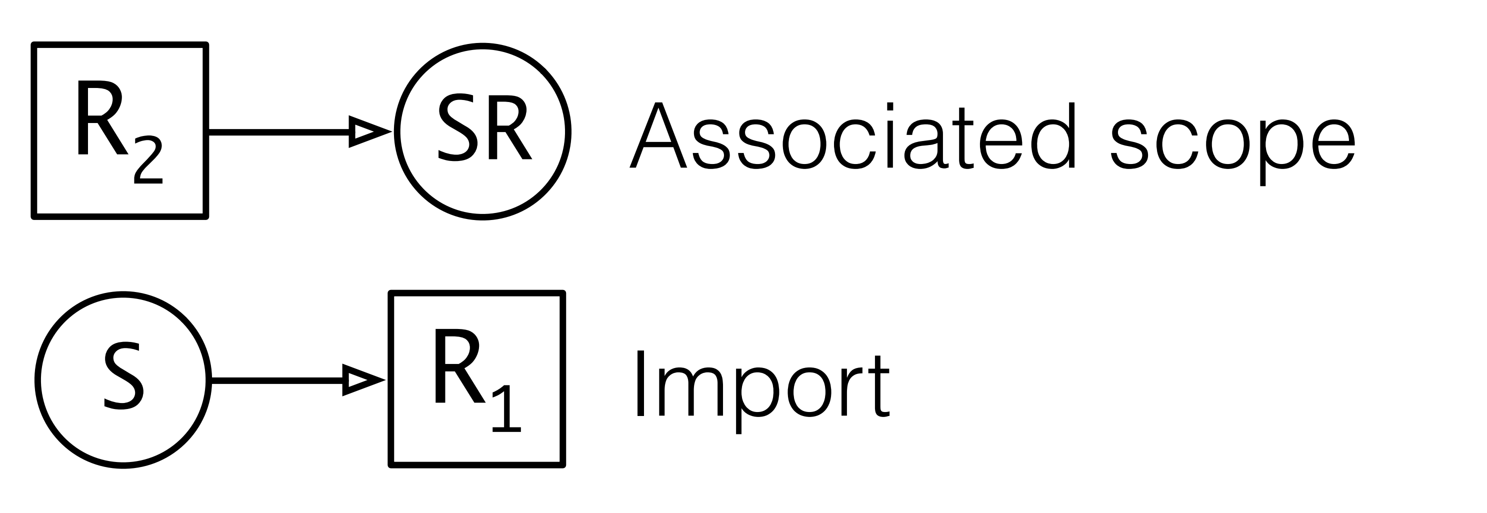


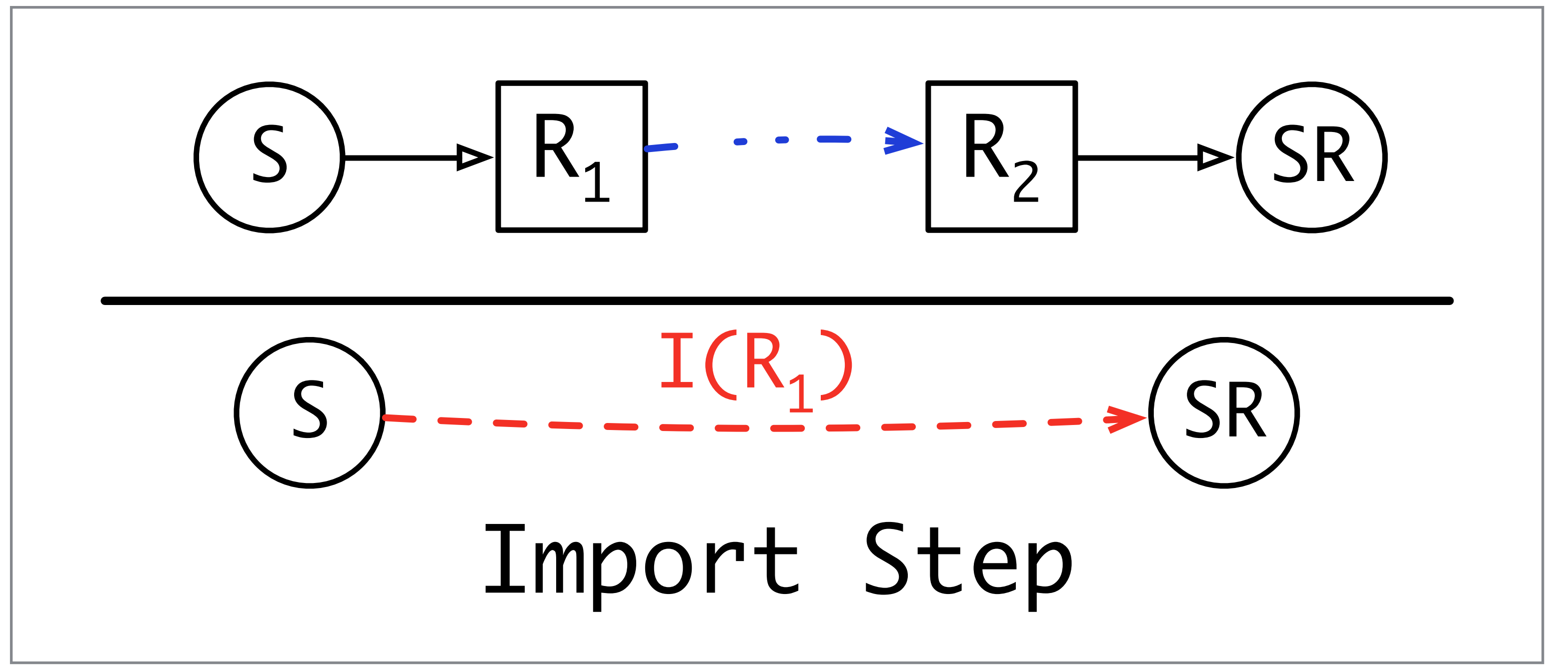


Imports

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



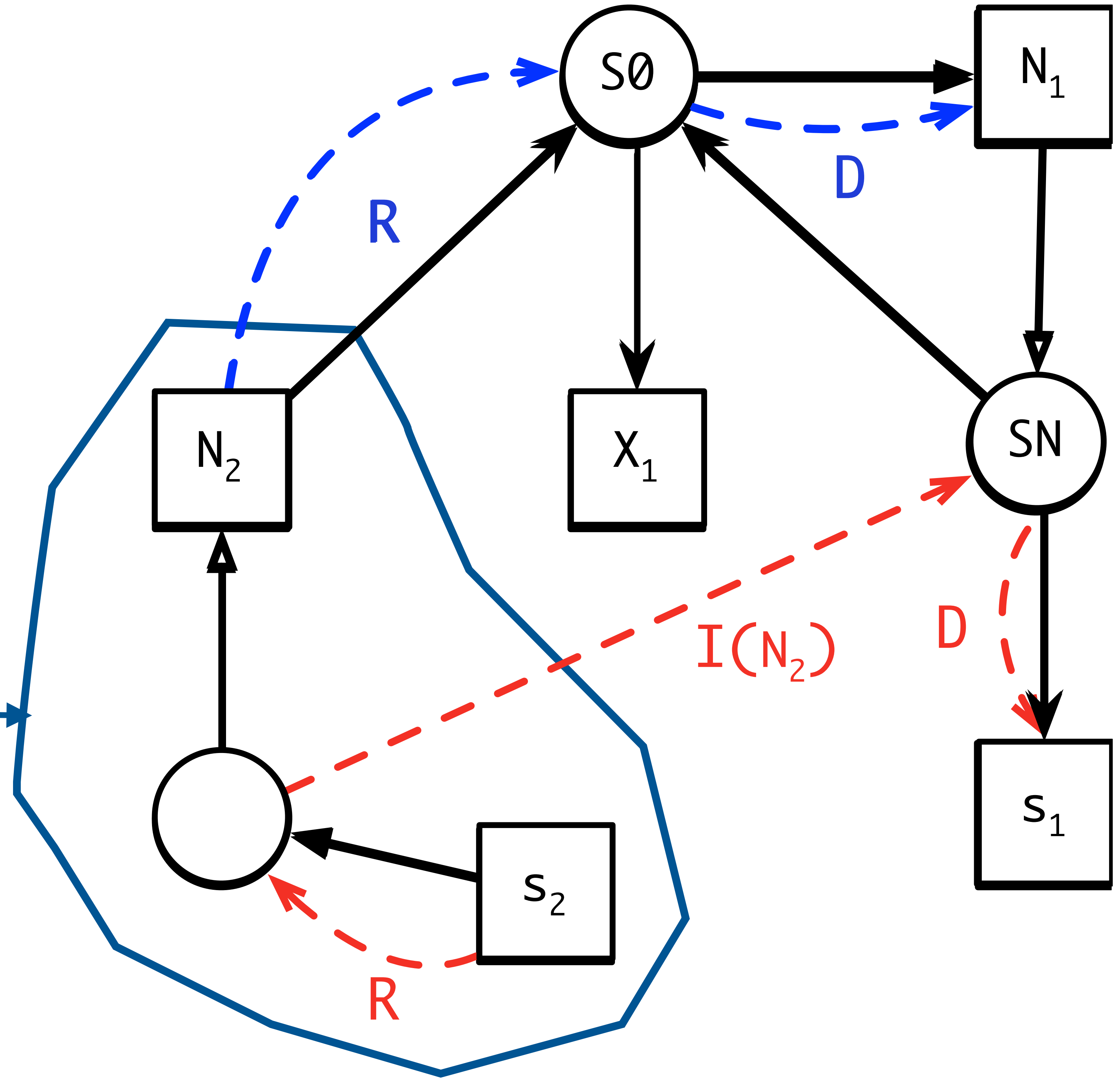




Qualified Names

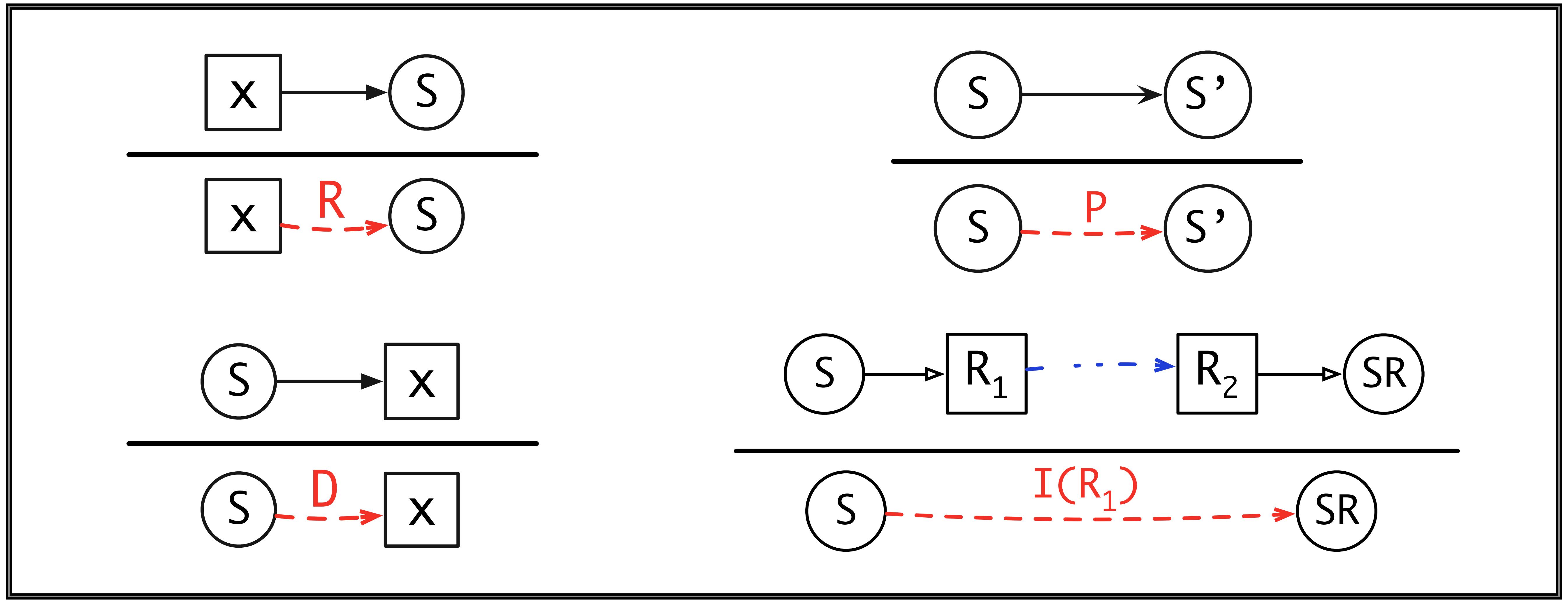
S0

```
module N1 {
    def s1 = 5
}

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



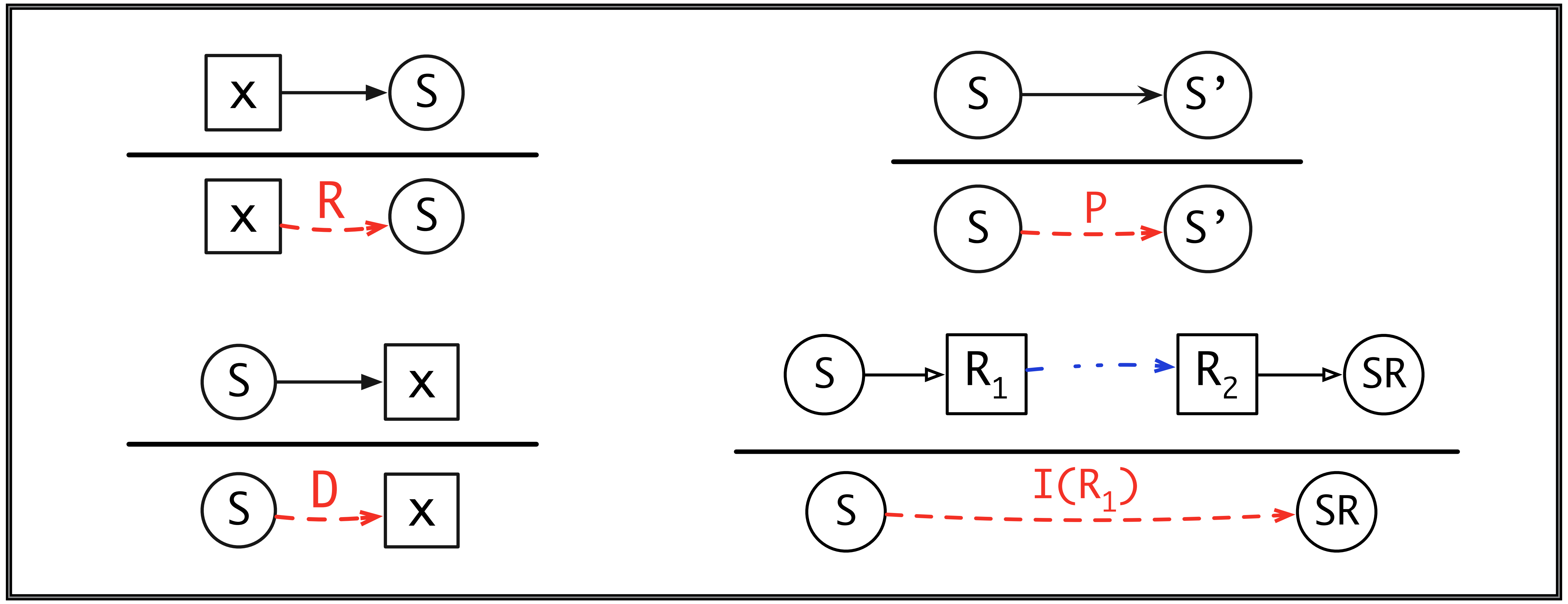
A Calculus for Name Resolution

Reachability of declarations from
references through scope graph edges



How about ambiguities?
References with multiple paths

A Calculus for Name Resolution



Reachability

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

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

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

Visibility

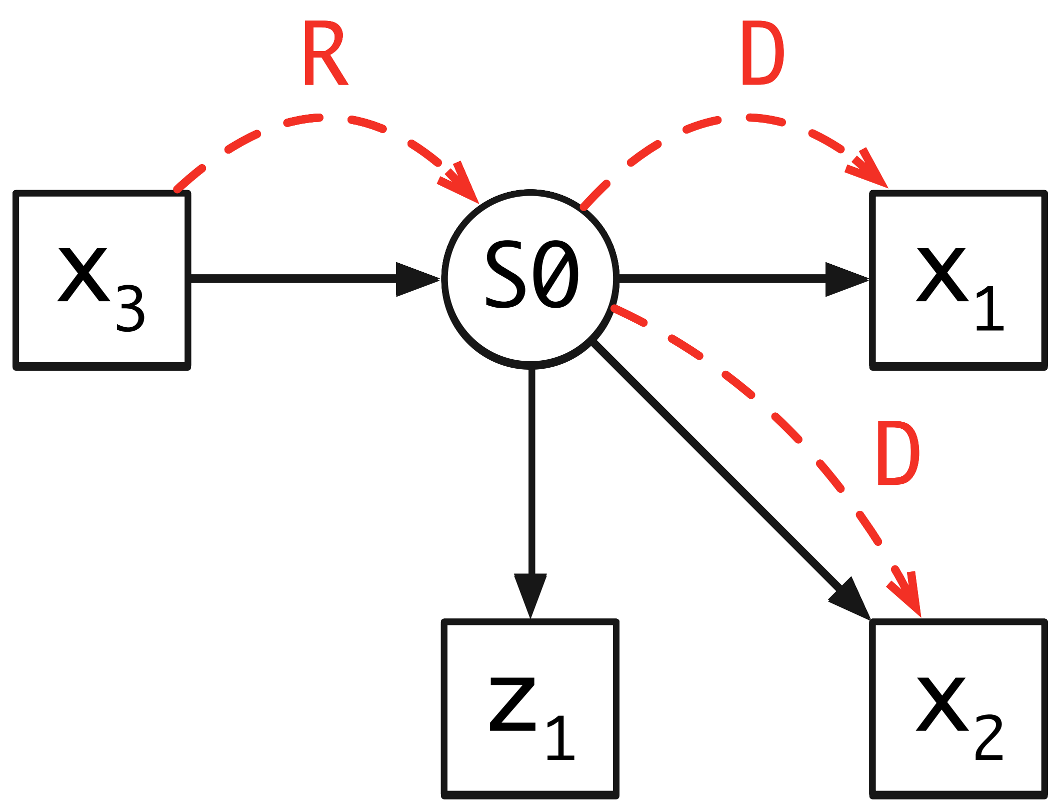
Ambiguous Resolutions

S0

def $\boxed{x_1}$ = 5

def $\boxed{x_2}$ = 3

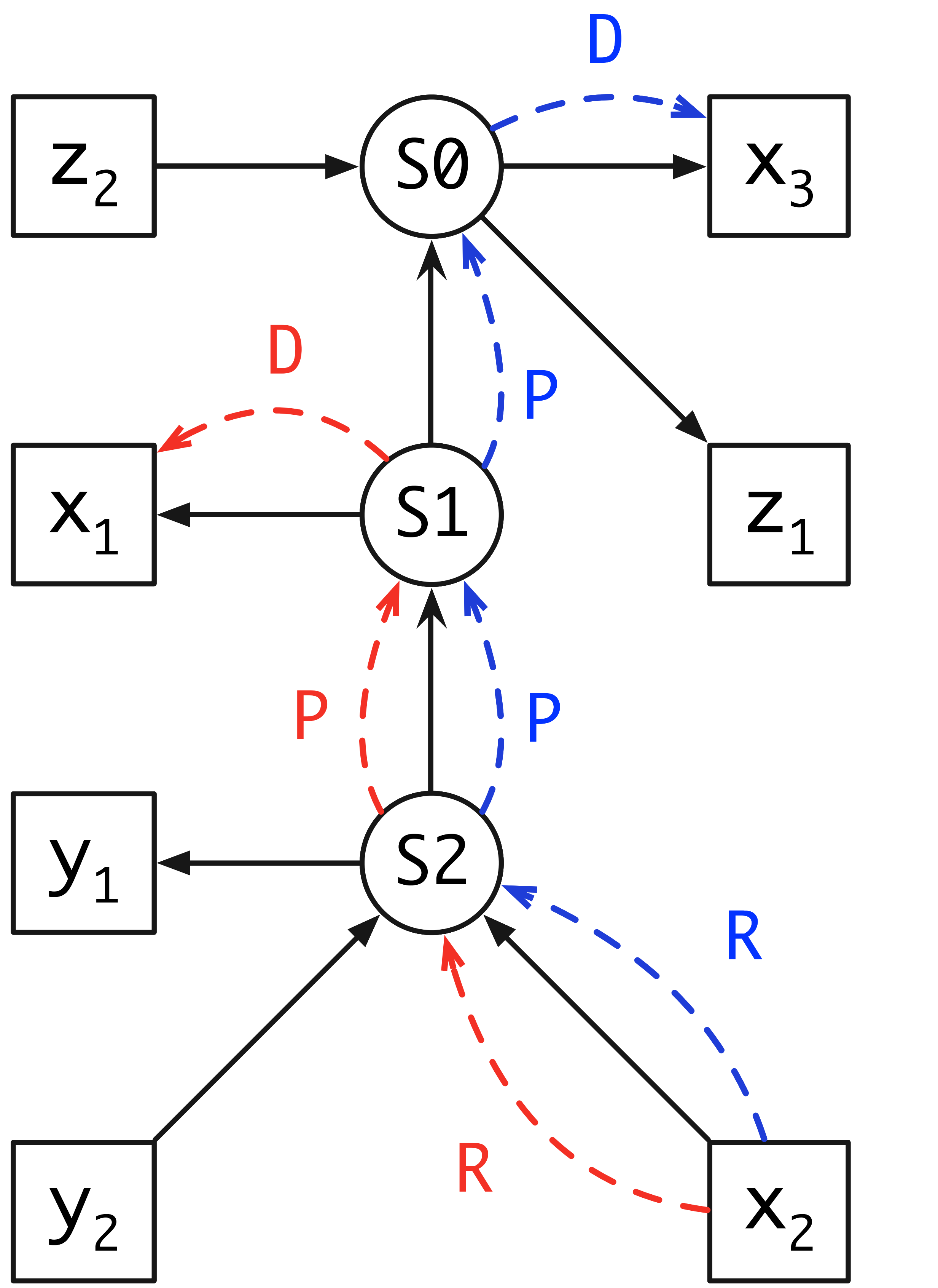
def $z_1 = x_3 + 1$



match t with
| A $\boxed{x}$ | B $\boxed{x}$ => …

Shadowing

```
def x₃ = z₂ 5 7        S0

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



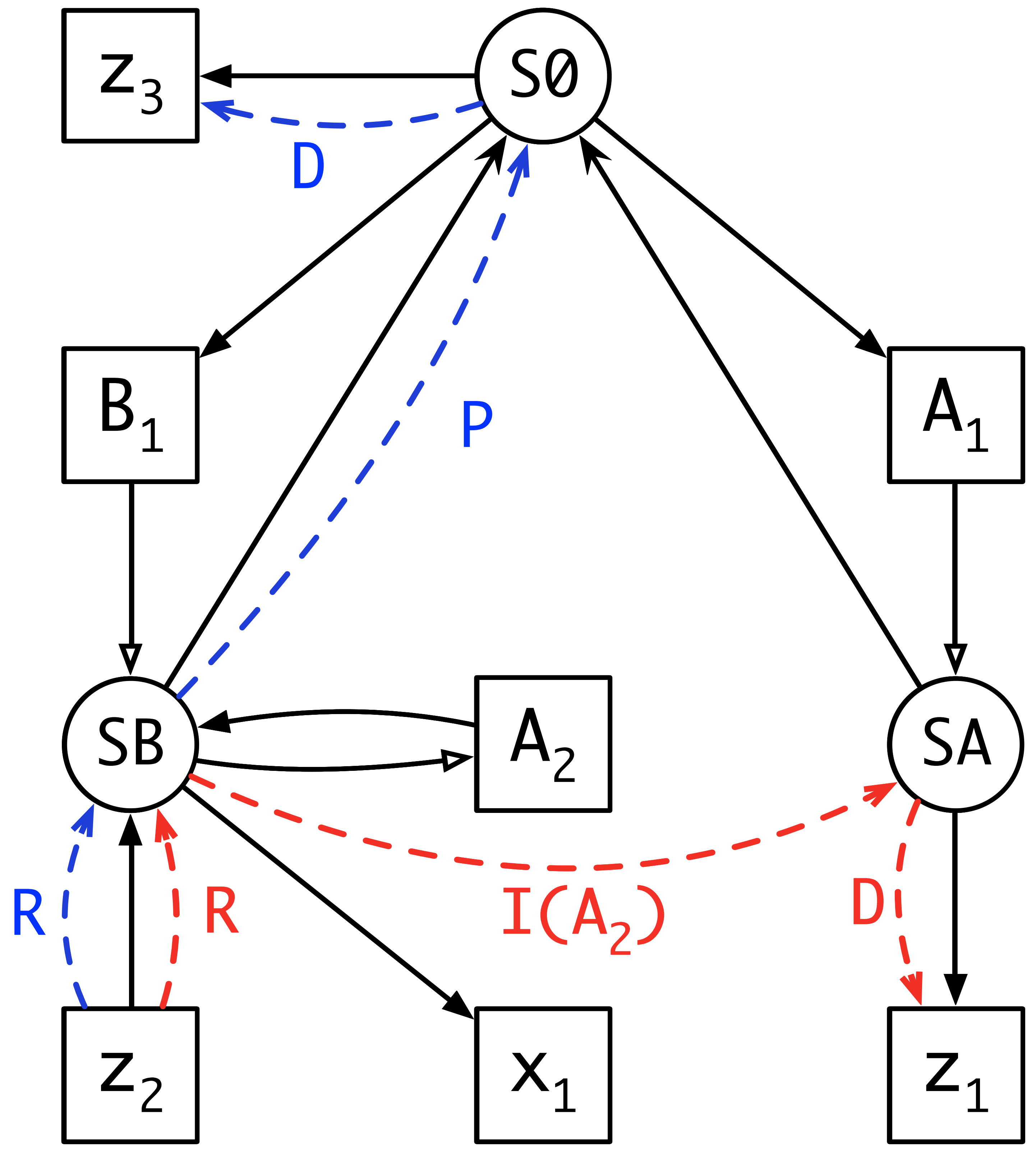
$$\frac{}{D < P.p}$$

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

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

Imports shadow Parents

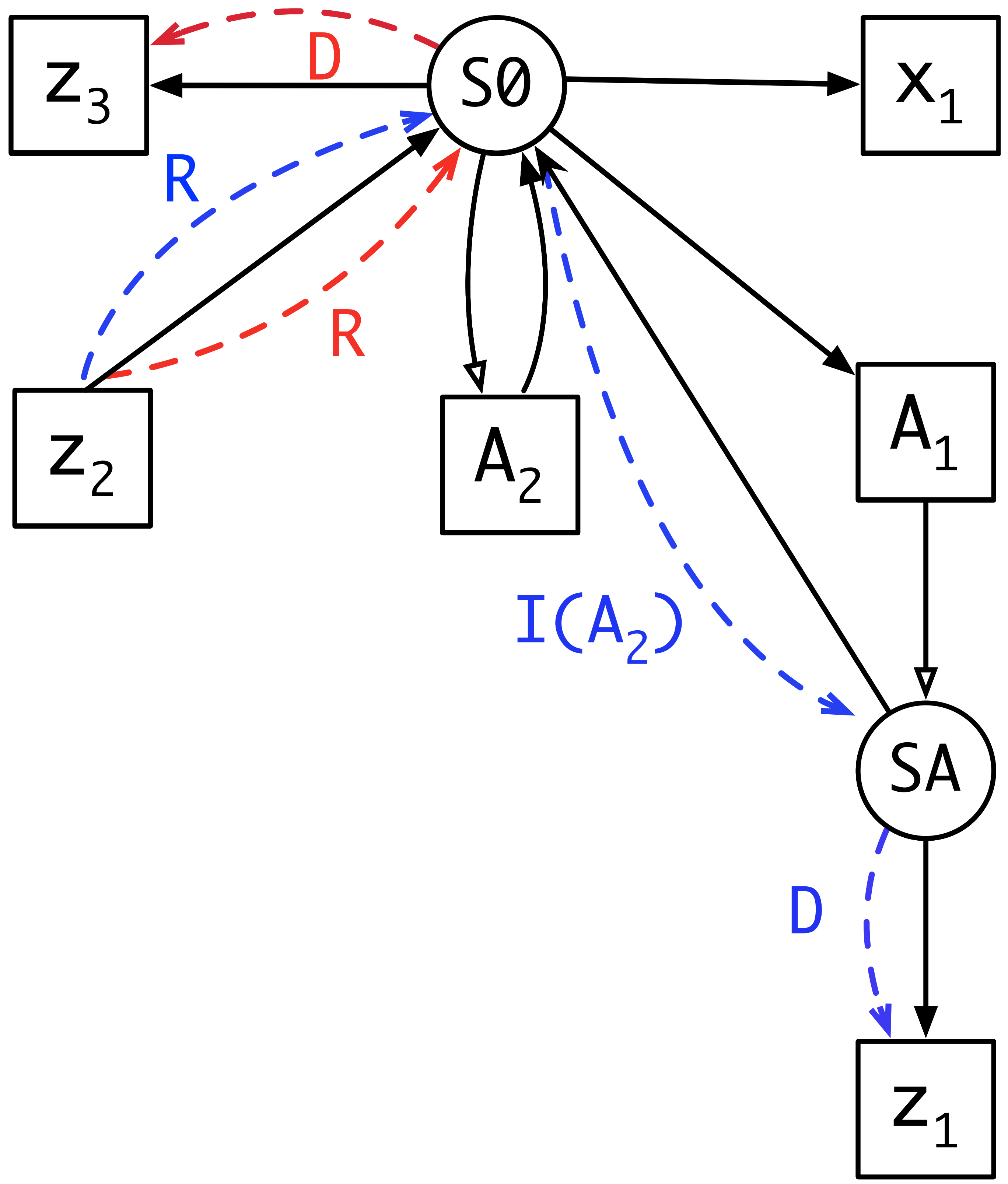
```
def z₃ = 2                    S0

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



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

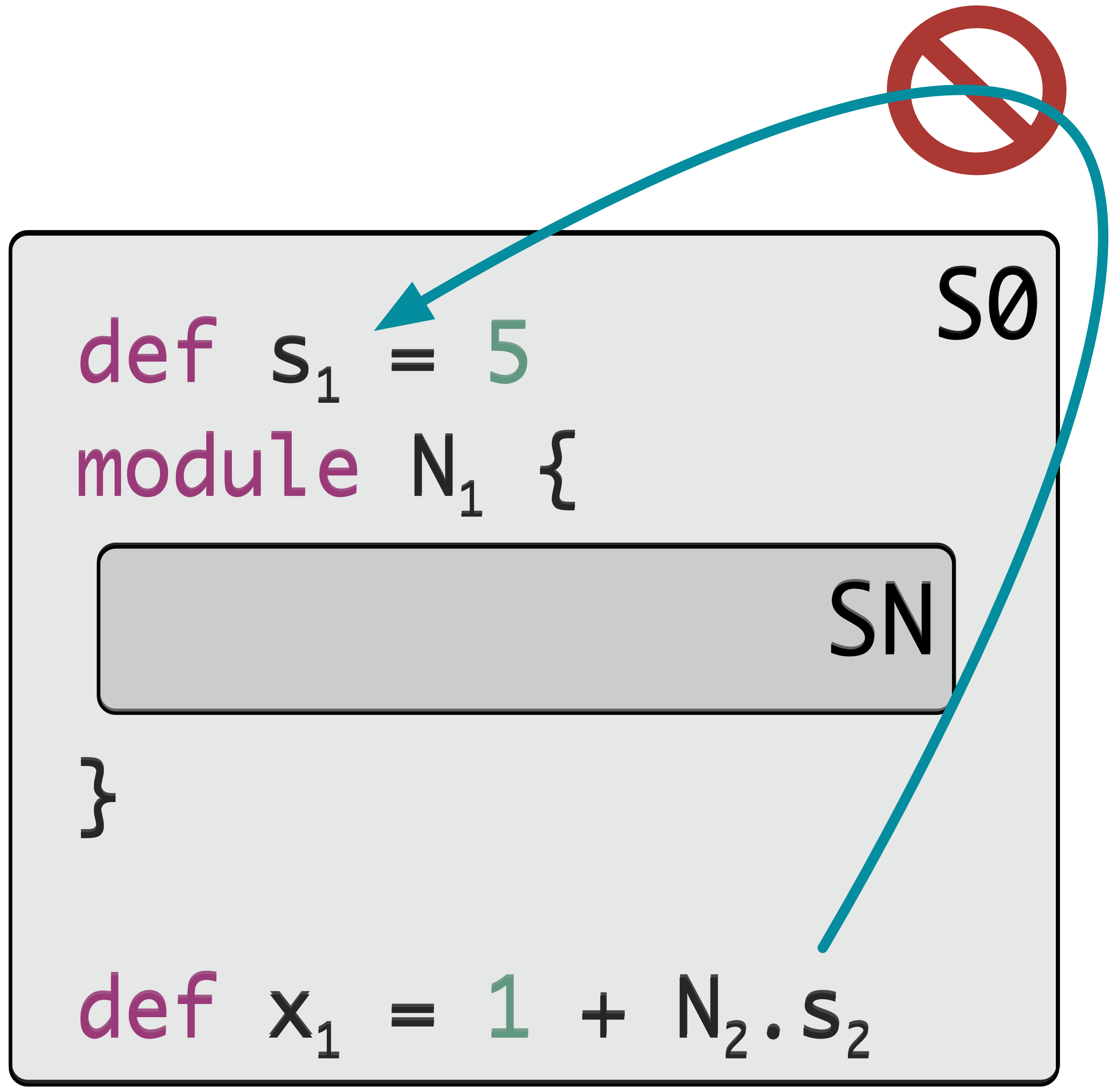
Imports vs. Includes

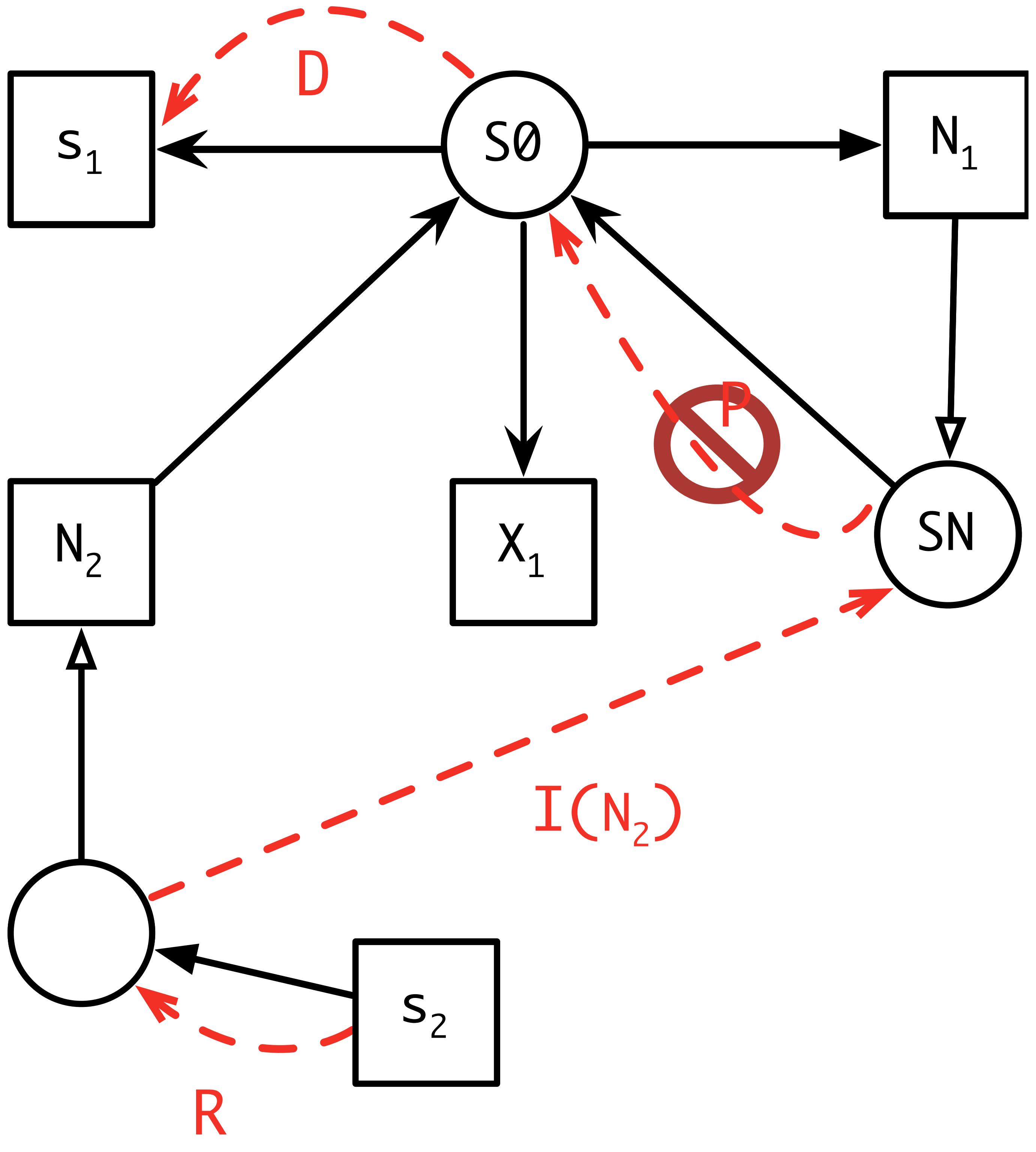
```
def z3 = 2                S0

module A1 {
  def z1 = 5              SA
}

import A2
include A2
def x1 = 1 + z2
```



~~D < I(_).p'~~ $\Rightarrow$ R.D < R.I(A_2).D

Import Parents

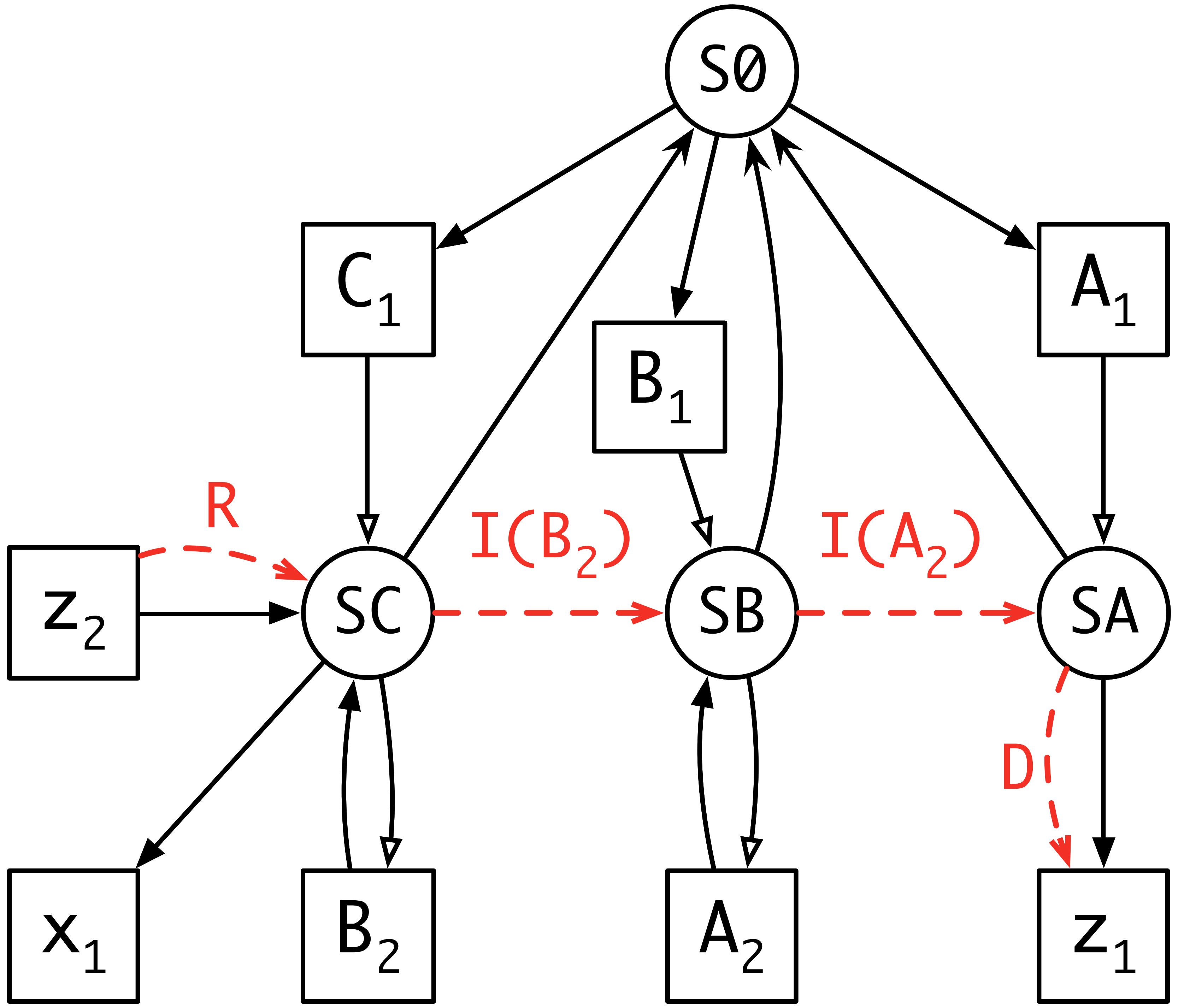




Well formed path: R.P*.I(_)*.D

Transitive vs. Non-Transitive

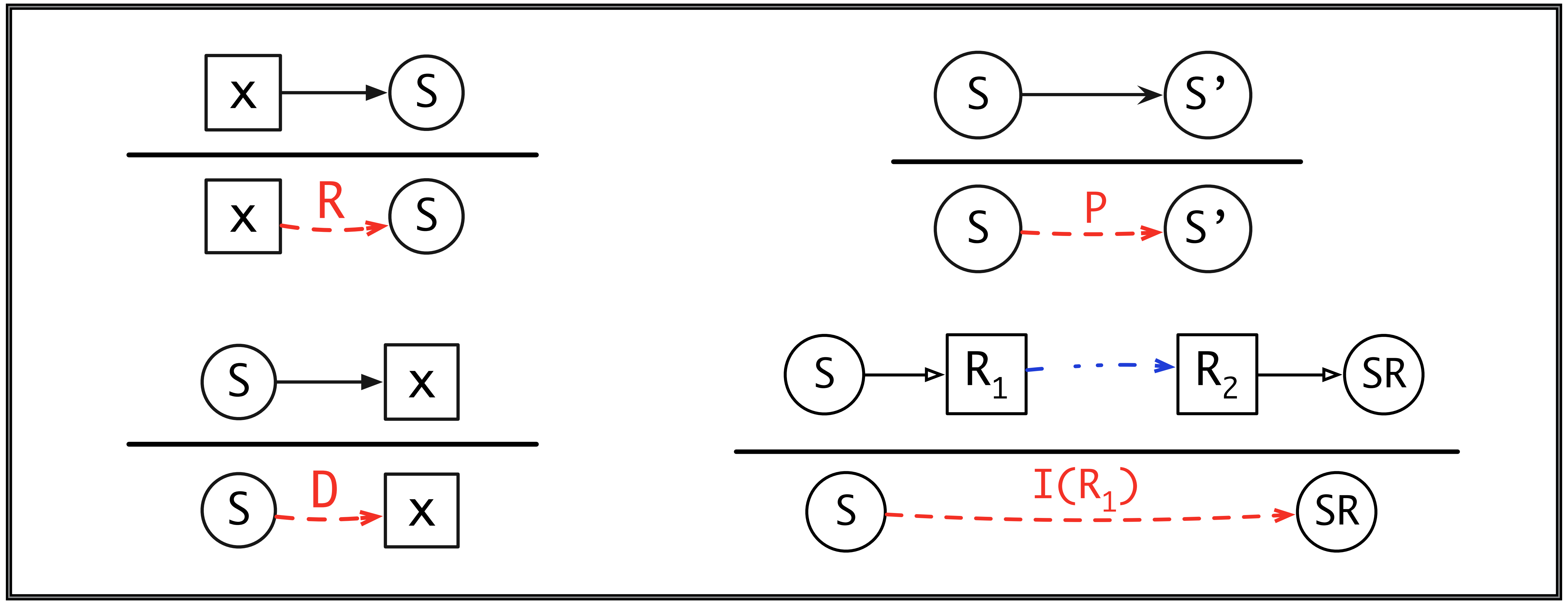
```
module A1 {
  def z1 = 5              SA
}                  ??
module B1 {
  import A2               SB
}
module C1 {
  import B2               SC
  def x1 = 1 + z2
}
```



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

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

A Calculus for Name Resolution



Reachability

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

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

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

Visibility

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

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

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

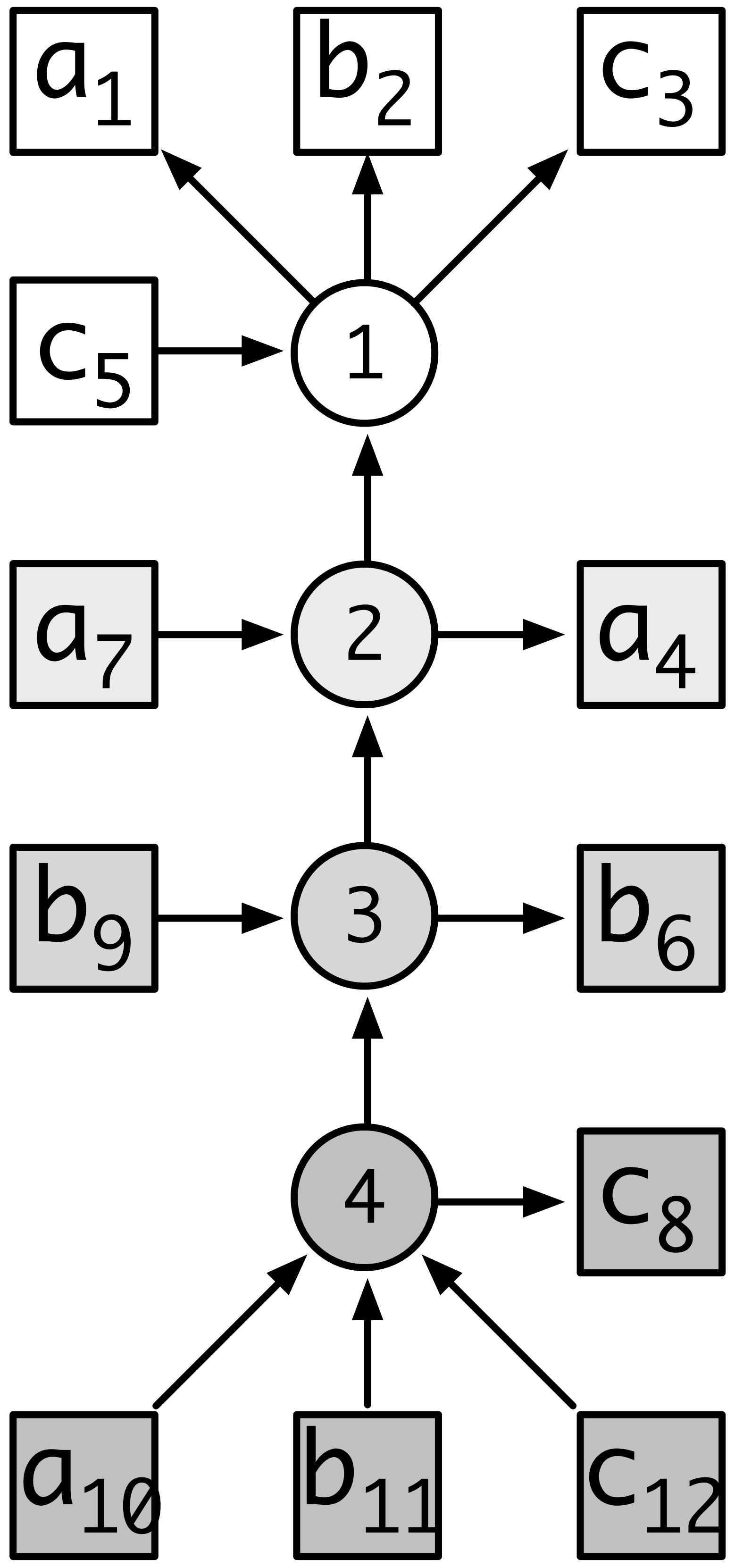
More Examples

From TUD-SERG-2015-001

Let Bindings

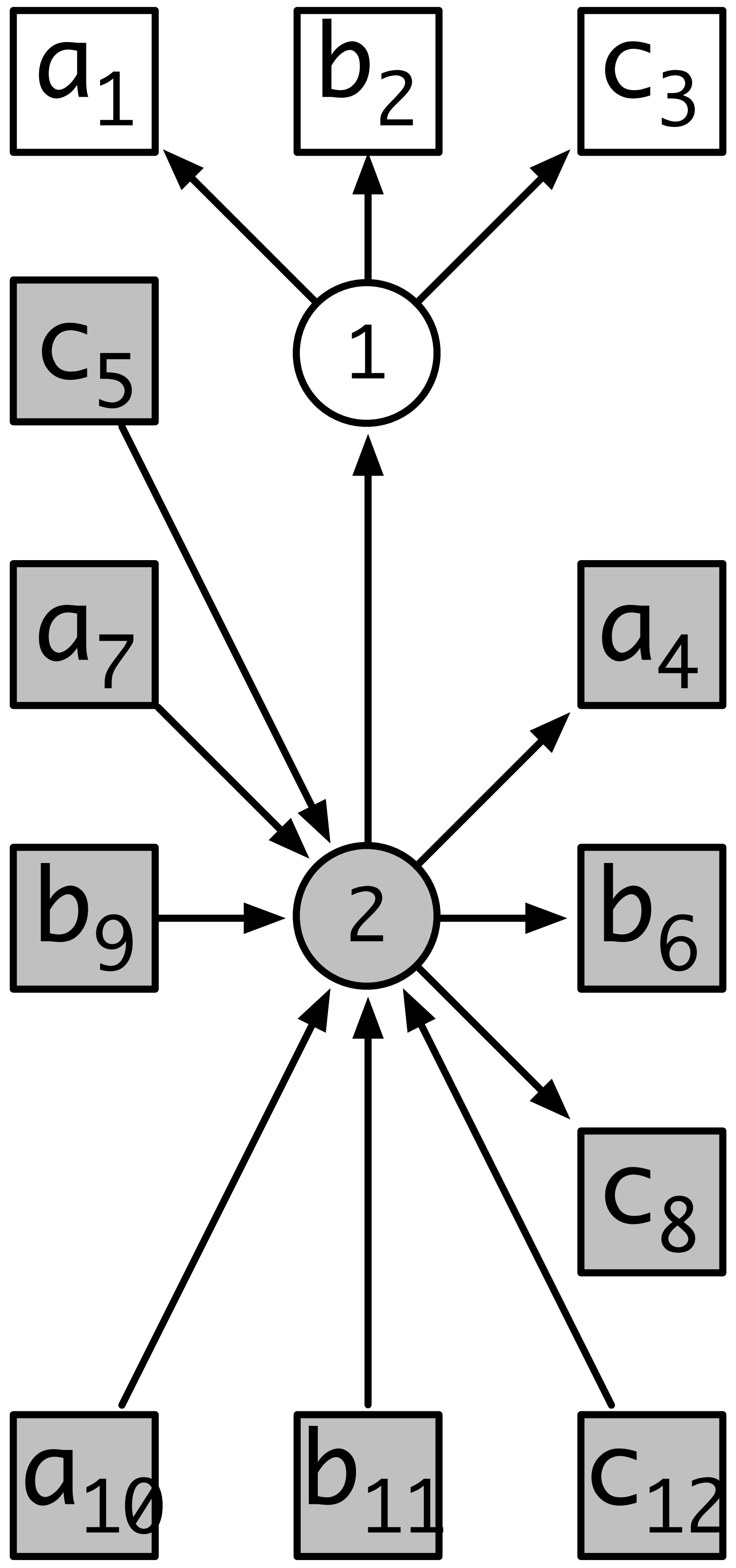
$\mathtt{def}\ a_1 = 0$
$\mathtt{def}\ b_2 = 1$
$\mathtt{def}\ c_3 = 2$

$\mathtt{let}$
$\quad a_4 = c_5$
$\quad b_6 = a_7$
$\quad c_8 = b_9$
$\mathtt{in}$
$\quad a_{10} + b_{11} + c_{12}$



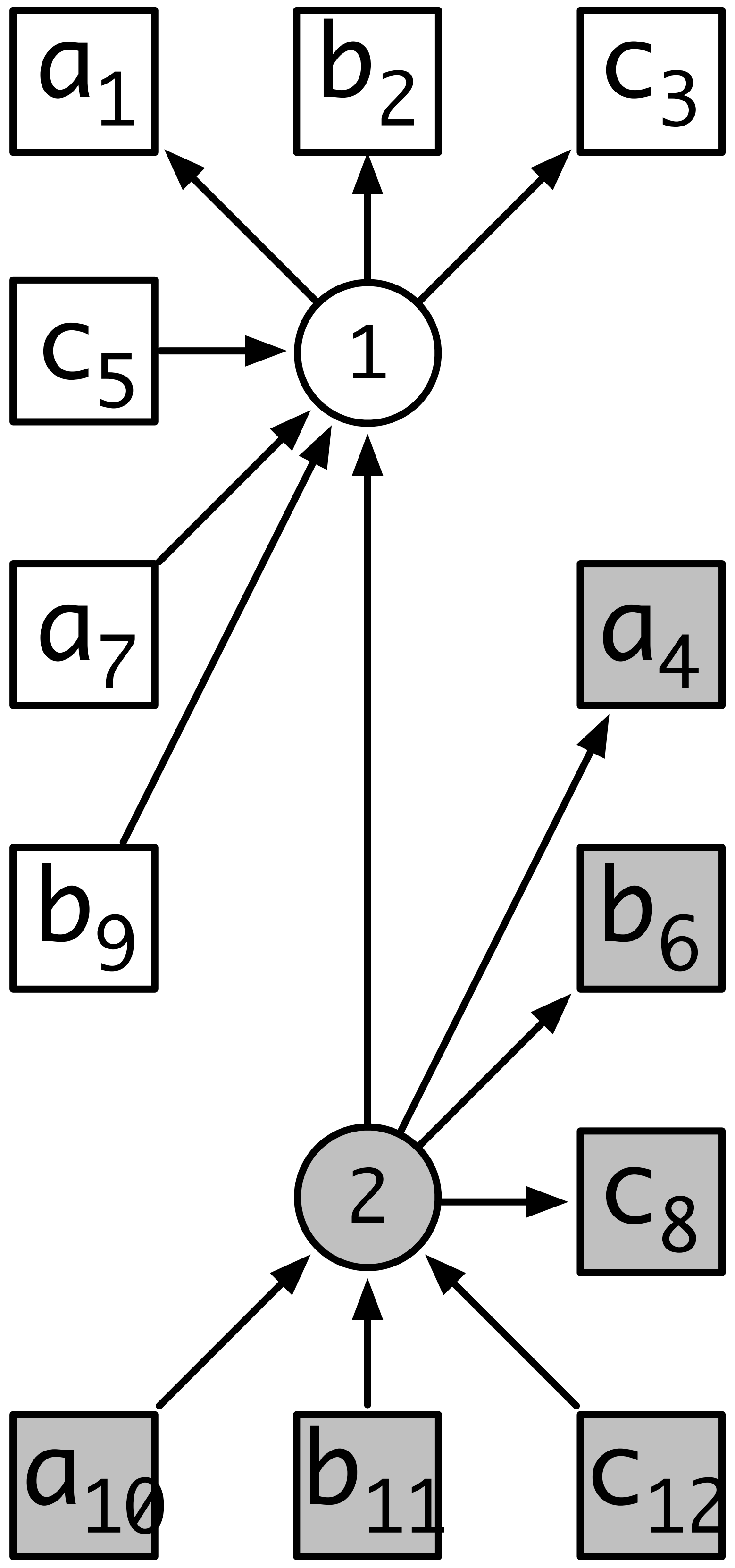
$\mathtt{def}\ a_1 = 0$
$\mathtt{def}\ b_2 = 1$
$\mathtt{def}\ c_3 = 2$

$\mathtt{letrec}$
$\quad a_4 = c_5$
$\quad b_6 = a_7$
$\quad c_8 = b_9$
$\mathtt{in}$
$\quad a_{10} + b_{11} + c_{12}$



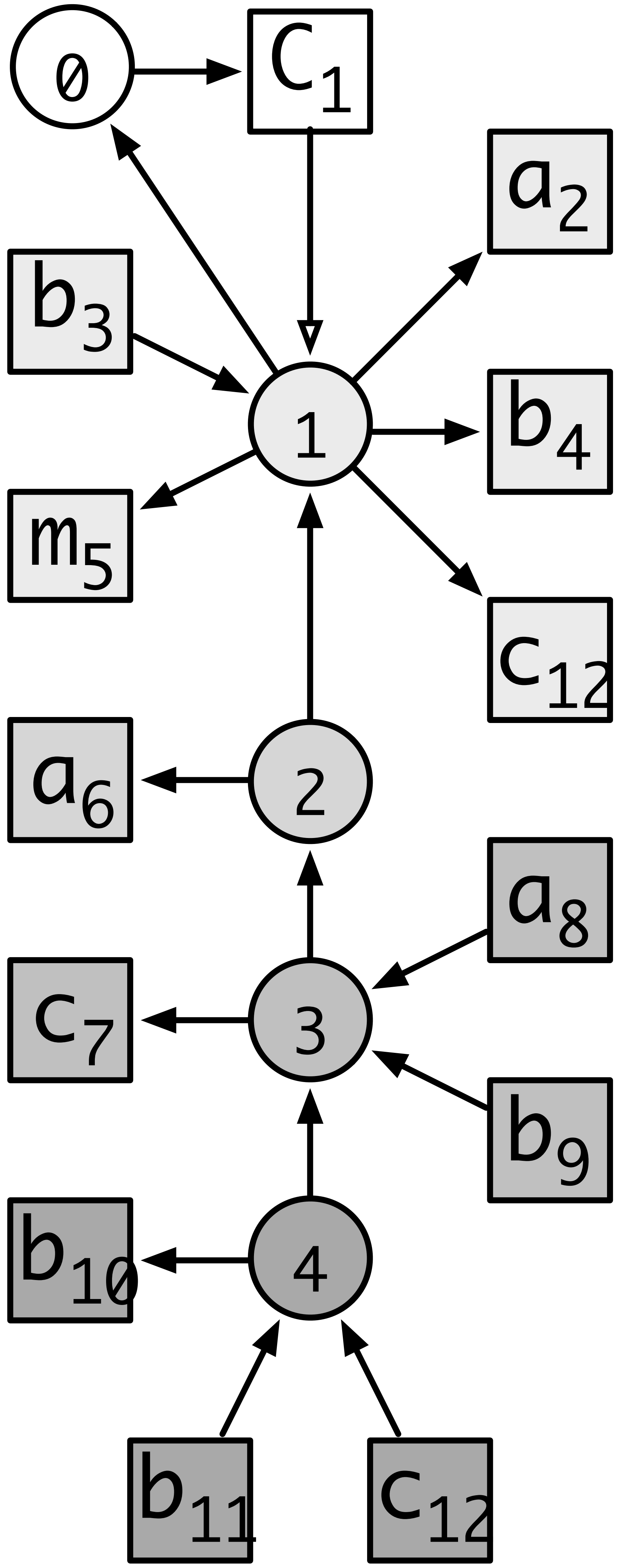
$\mathtt{def}\ a_1 = 0$
$\mathtt{def}\ b_2 = 1$
$\mathtt{def}\ c_3 = 2$

$\mathtt{letpar}$
$\quad a_4 = c_5$
$\quad b_6 = a_7$
$\quad c_8 = b_9$
$\mathtt{in}$
$\quad a_{10} + b_{11} + c_{12}$



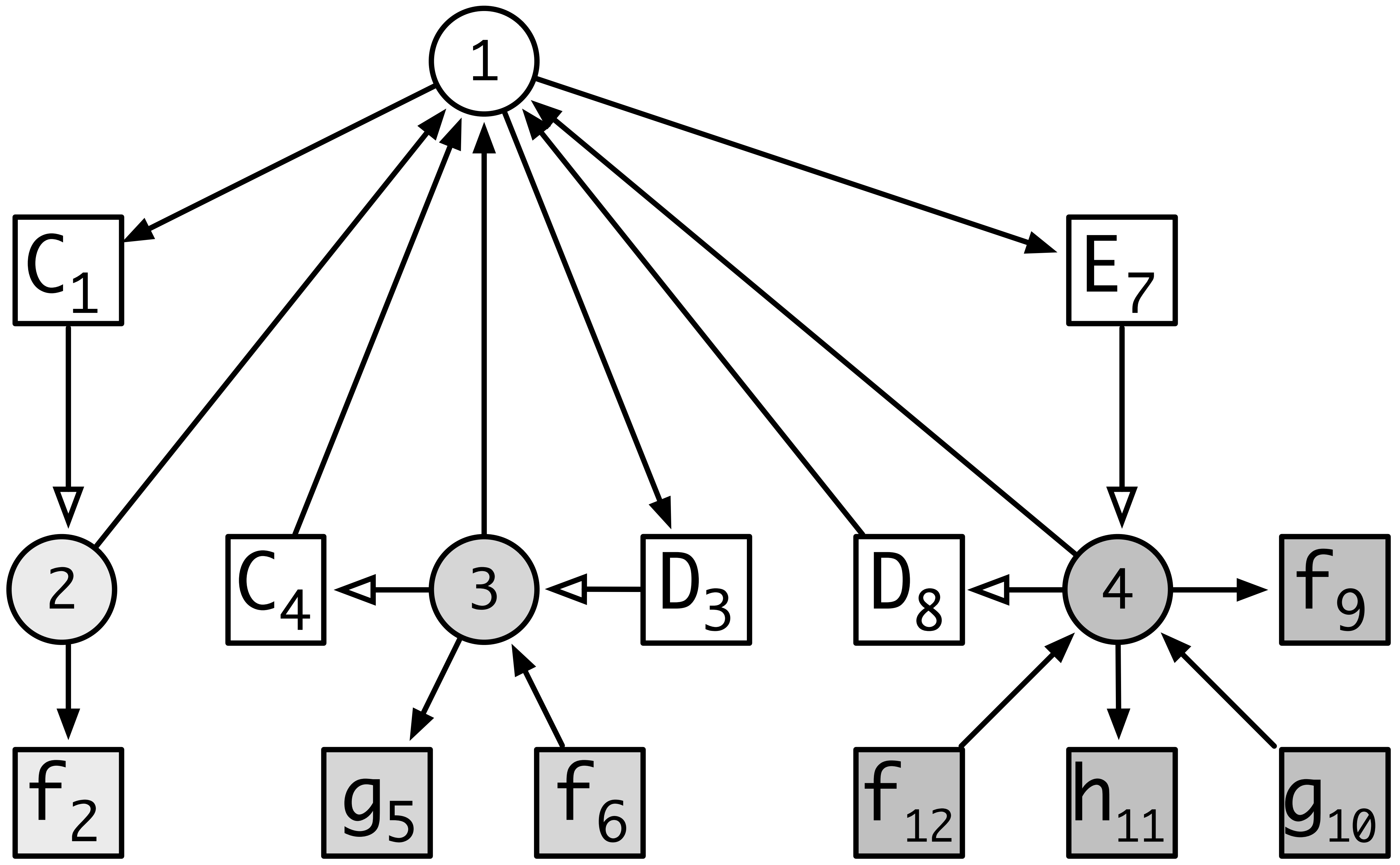
Definition before Use / Use before Definition

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



Inheritance

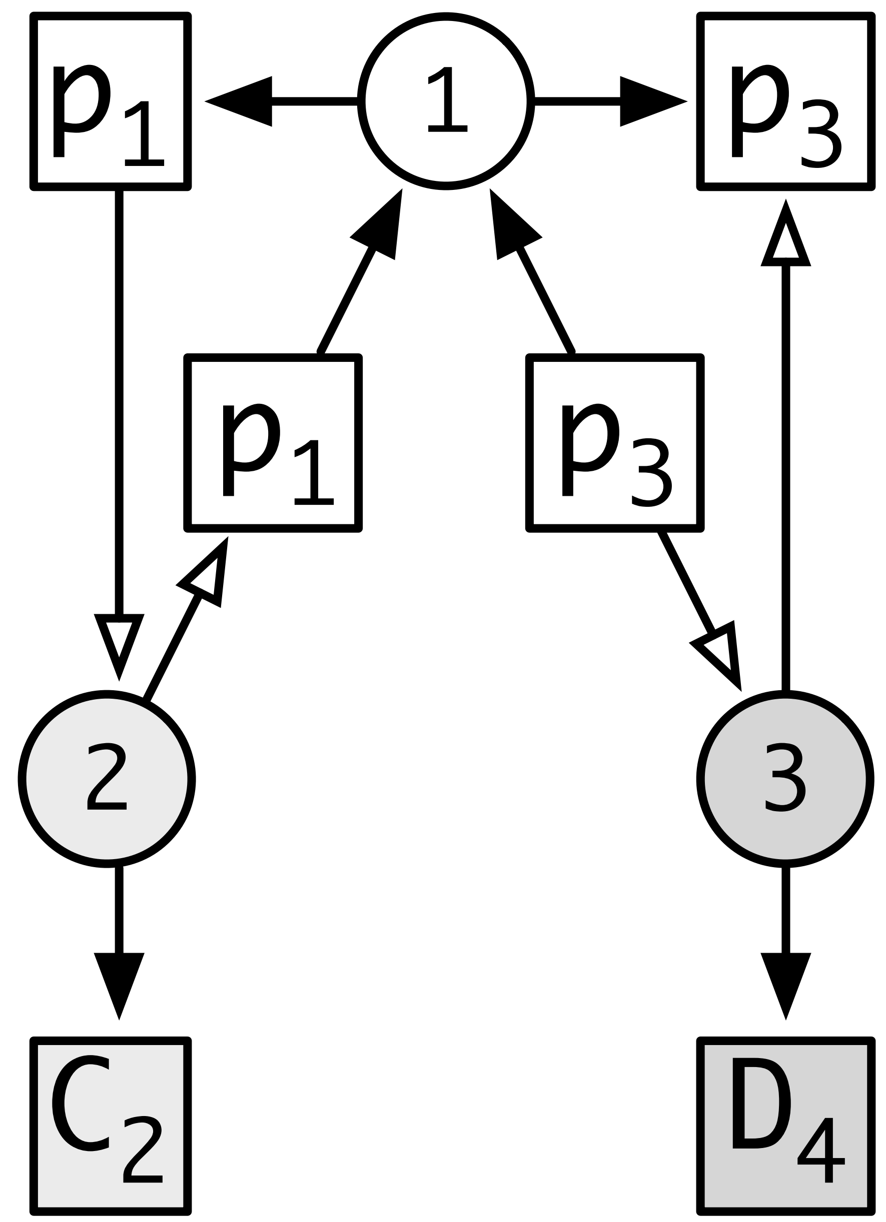
```
class C₁ {
  int f₂ = 42;
}
class D₃ extends C₄ {
  int g₅ = f₆;
}
class E₇ extends D₈ {
  int f₉  = g₁₀;
  int h₁₁ = f₁₂;
}
```



Java Packages

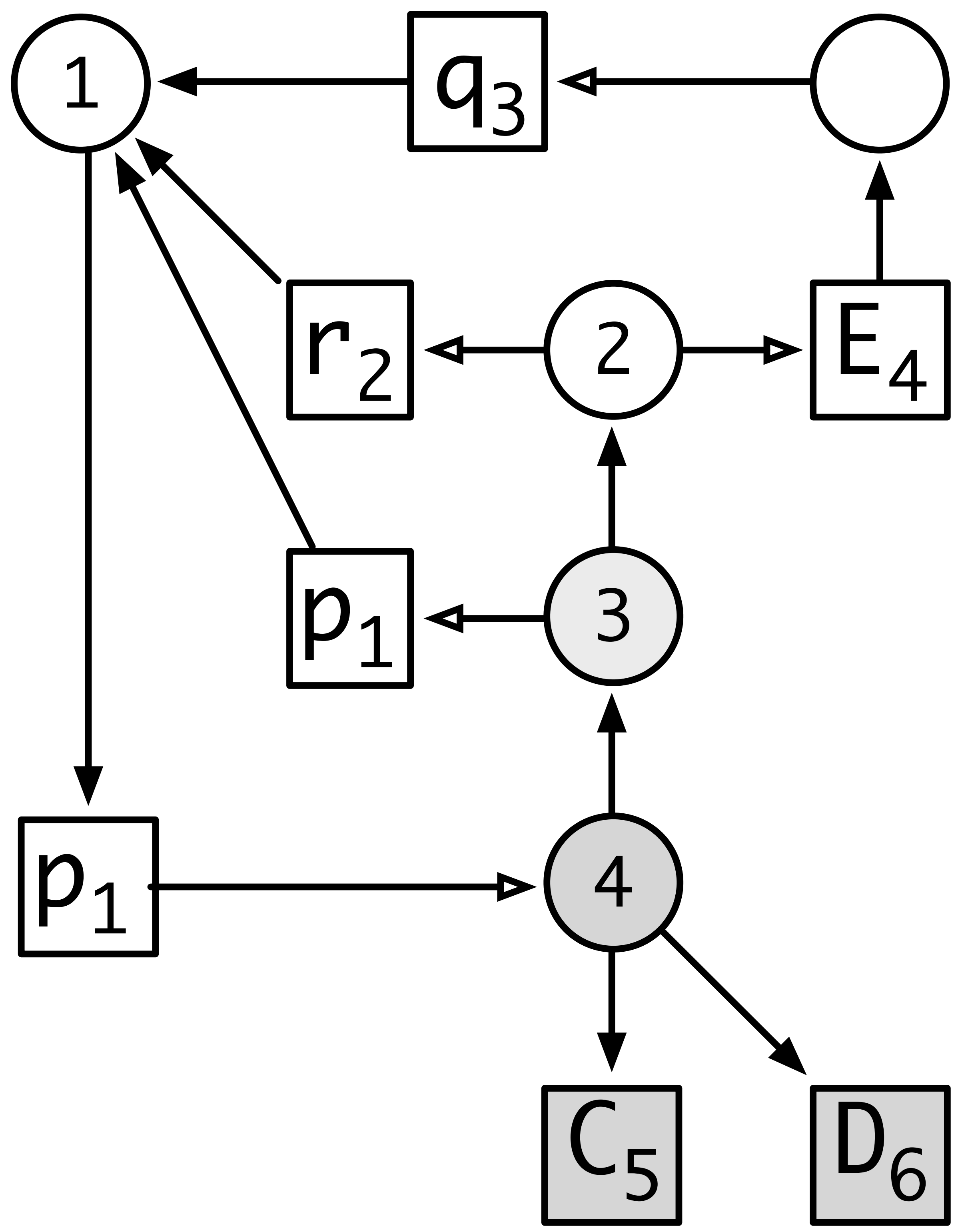
```
package p1;
    class C2 {}
```

```
package p3;
    class D4 {}
```



Java Import

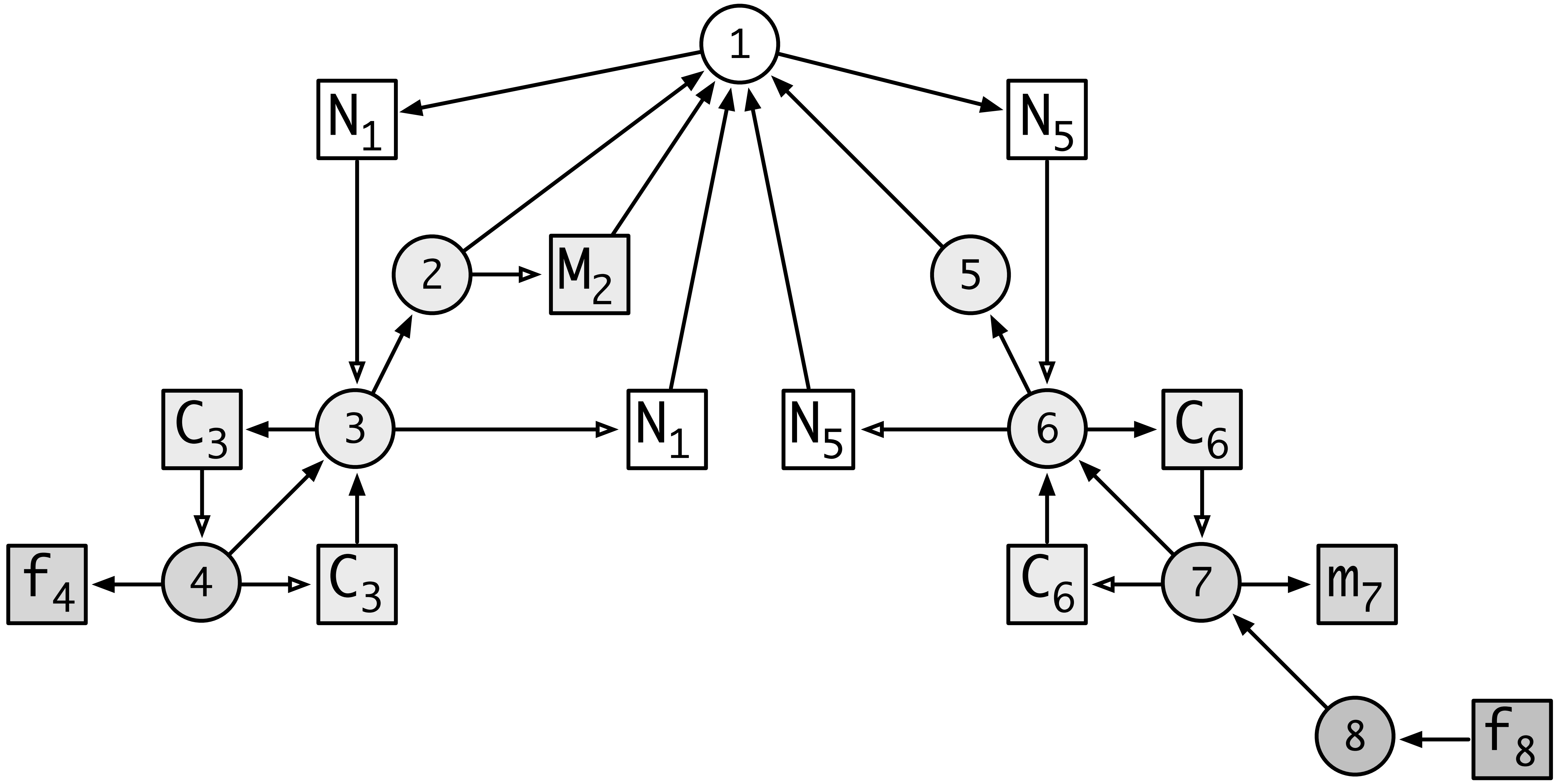
```
package p1;

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

- 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]

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Next: Constraint-Based Type Checkers