The Spoofax Language Workbench

Eelco Visser

TU Delft

Online Workshop: Modern Compiler Technologies 2020
Huawei | Moscow | November 11, 2020

What is Spoofax?

A tool for implementing programming languages

- Open source and freely available
- Used in education, research, and industry
- Requires a lot of software engineering to maintain

A long term research project

- Incubator for language engineering research
- Basis for implementation and evaluation of 100+ papers
- Imperfect approximation of a language designer’s workbench

Spoofax in Action

Research

- Language Engineering, Language Prototyping

Education

- Compiler Construction (MiniJava, ChocoPy)
- Language Engineering Project

Academic Workflow Engineering

- WebDSL (researchr.org, WebLab, …)

Industry

- Oracle Labs: Graph Analytics
- Canon: Several DSLs
- Philips: Software Restructuring

Meta-Languages for Language Definition

Meta-languages

- **Syntax definition with SDF3**
- **Static semantics with Statix**
- Data-flow analysis with FlowSpec
- Transformation with Stratego
- Dynamic Semantics with DynSem/Dynamix
- Editor service definition with ESV

Declarative Syntax Definition with SDF3

Multi-purpose Syntax Definition with SDF3

Luís Eduardo de Souza Amorim[1] and Eelco Visser[2](✉)

[1] Australian National University, Canberra, Australia
[2] Delft University of Technology, Delft, The Netherlands
e.visser@tudelft.nl

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permissive syntax, layout constraints, formatting templates, placeholder syntax, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, scannerless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · Programming language · Parsing

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a *grammar formalism* in the tradition of Chomsky's context-free grammars [14] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches.

The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [44,56]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through incremental and modular scanner and parser generation [24–26] in order to provide fast turnaround times during language development.

F. de Boer and A. Cerone (Eds.): SEFM 2020, LNCS 12310, pp. 1–23, 2020.
https://doi.org/10.1007/978-3-030-58768-0_1

Declarative Syntax Definition

Representation

- Syntax trees

Specification Formalism: SDF3

- Productions + Constructors + Templates + Disambiguation

Declarative Semantics

- Well-formedness of syntax trees wrt syntax definition

Language-Independent Tools

- Parser
- Formatting based on layout hints in grammar
- Syntactic completion

Syntax = Structure

```
module structure

imports Common

context-free start-symbols Exp

context-free syntax

  Exp.Var = ID

  Exp.Int = INT

  Exp.Add = Exp "+" Exp

  Exp.Fun = "function" "(" {ID ","}* ")" "{" Exp "}"

  Exp.App = Exp "(" {Exp ","}* ")"

  Exp.Let = "let" Bnd* "in" Exp "end"

  Bnd.Bnd = ID "=" Exp
```

```
let
  inc = function(x) { x + 1 }
 in
  inc(3)
end
```

```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), [Int("3")])
)
```

Parsing = Formatting^{-1}

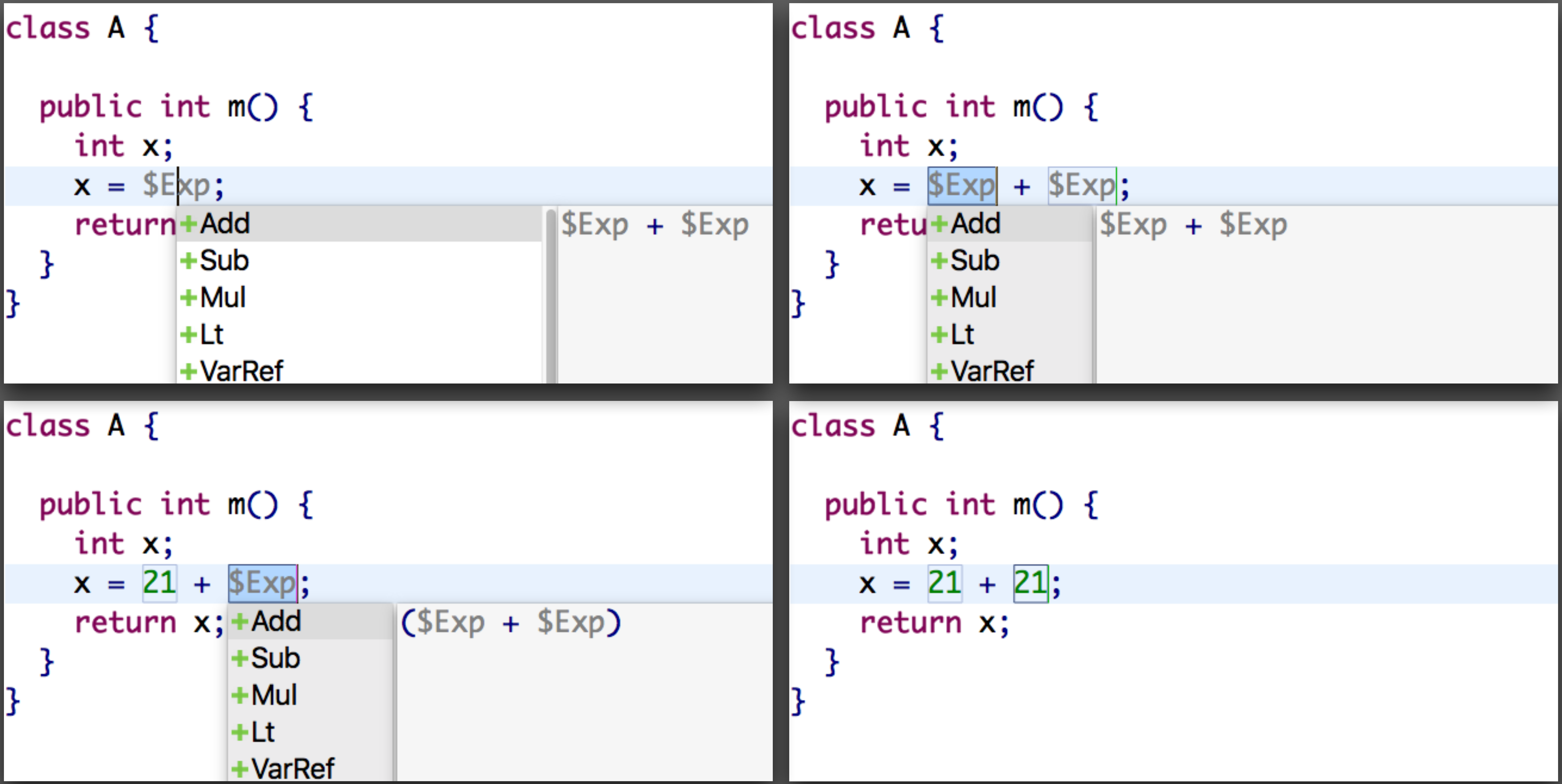
```
context-free syntax

  Exp.Var = <<ID>>

  Exp.Int = <<INT>>

  Exp.Add = <<Exp> + <Exp>>

  Exp.Fun = <
    function(<{ID ","}*>){
      <Exp>
    }
  >

  Exp.App = <<Exp>(<{Exp ","}*>)>

  Exp.Let = <
    let
      <Bnd*>
     in
      <Exp>
    end
  >

  Bnd.Bnd = <<ID> = <Exp>>
```

```
let
  inc = function(x) { x + 1 }
 in
  inc(3)
end
```

```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), [Int("3")])
)
```

```
let
  inc = function(x){
    x + 1
  }
 in
  inc(3)
end
```

Completion = Rewrite(Incomplete Structure)



Disambiguation

Ambiguity = Multiple Possible Parses

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>>

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>>

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>>
```

```
a + b + c
```

↓

```
amb(
  [ Add(Var("a"), Add(Var("b"), Var("c")))
  , Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)
```

Disambiguation = Select(Structure)

```
context-free syntax
  Exp        = <(<Exp>)> {bracket}

  Exp.Int    = INT
  Exp.Var    = ID
  Exp.Add    = <<Exp> + <Exp>>

  Exp.Fun    = <function(<{ID ","}*>) <Exp>>
  Exp.App    = <<Exp> <Exp>>

  Exp.Let    = <let <Bnd*> in <Exp>>

  Bnd.Bnd    = <<ID> = <Exp>>

  Exp.If     = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match  = <match <Exp> with <{Case "|"}+>>
  Case.Case  = [[Pat] → [Exp]]

  Pat.PVar   = ID
  Pat.PApp   = <<Pat> <Pat>>
```

```
a + b + c
```

↓

```
amb(
  [ Add(Var("a"), Add(Var("b"), Var("c")))
  , Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)
```

↓

```
Add(Add(Var("a"), Var("b")), Var("c"))
```

Declarative Disambiguation = Separate Concern

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] -> [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

Associativity = Solve Intra Operator Ambiguity

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
a + b + c
```

```
amb(
  [ Add(Var("a"), Add(Var("b"), Var("c")))
  , Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)
```

```
Add(Add(Var("a"), Var("b")), Var("c"))
```

Priority = Solve Inter Operator Ambiguity

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
f a + b
```

```
amb(
  [ Add(App(Var("f"), Var("a")), Var("b"))
  , App(Var("f"), Add(Var("a"), Var("b")))
  ]
)
```

```
Add(App(Var("f"), Var("a")), Var("b"))
```

Dangling Else = Operators with Overlapping Prefix

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
if(1) if(2) 3 else 4
```

```
amb(
  [ IfElse(
      Int("1")
    , If(Int("2"), Int("3"))
    , Int("4")
    )
  , If(
      Int("1")
    , IfElse(Int("2"), Int("3"), Int("4"))
    )
  ]
)
```

```
If(
  Int("1")
, IfElse(Int("2"), Int("3"), Int("4"))
)
```

Parenthesize

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
(a + b) + c
```

↓

```
Add(Add(Var("a"), Var("b")), Var("c"))
```

↓

```
a + b + c
```

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp        = <(<Exp>)> {bracket}

  Exp.Int    = INT
  Exp.Var    = ID
  Exp.Add    = <<Exp> + <Exp>> {left}

  Exp.Fun    = <function(<{ID ","}*>) <Exp>>
  Exp.App    = <<Exp> <Exp>> {left}

  Exp.Let    = <let <Bnd*> in <Exp>>

  Bnd.Bnd    = <<ID> = <Exp>>

  Exp.If     = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match  = <match <Exp> with <{Case "|"}+>>
               {longest-match}
  Case.Case  = [[Pat] → [Exp]]

  Pat.PVar   = ID
  Pat.PApp   = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
a + (let x = b in (c + d))
```

```
Add(
  Var("a")
, Let(
    [Bnd("x", Var("b"))]
  , Add(Var("c"), Var("d"))
  )
)
```

```
a + let
  x = b
in
  c + d
```

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp         = <(<Exp>)> {bracket}

  Exp.Int     = INT
  Exp.Var     = ID
  Exp.Add     = <<Exp> + <Exp>> {left}

  Exp.Fun     = <function(<{ID ","}*>) <Exp>>
  Exp.App     = <<Exp> <Exp>> {left}

  Exp.Let     = <let <Bnd*> in <Exp>>

  Bnd.Bnd     = <<ID> = <Exp>>

  Exp.If      = <if(<Exp>) <Exp>>
  Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

  Exp.Match   = <match <Exp> with <{Case "|"}+>>
                {longest-match}
  Case.Case   = [[Pat] → [Exp]]

  Pat.PVar    = ID
  Pat.PApp    = <<Pat> <Pat>> {left}
context-free priorities
  Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun
```

```
(a + (let x = b in c)) + d
```

```
Add(
  Add(
    Var("a")
  , Let([Bnd("x", Var("b"))], Var("c"))
  )
, Var("d")
)
```

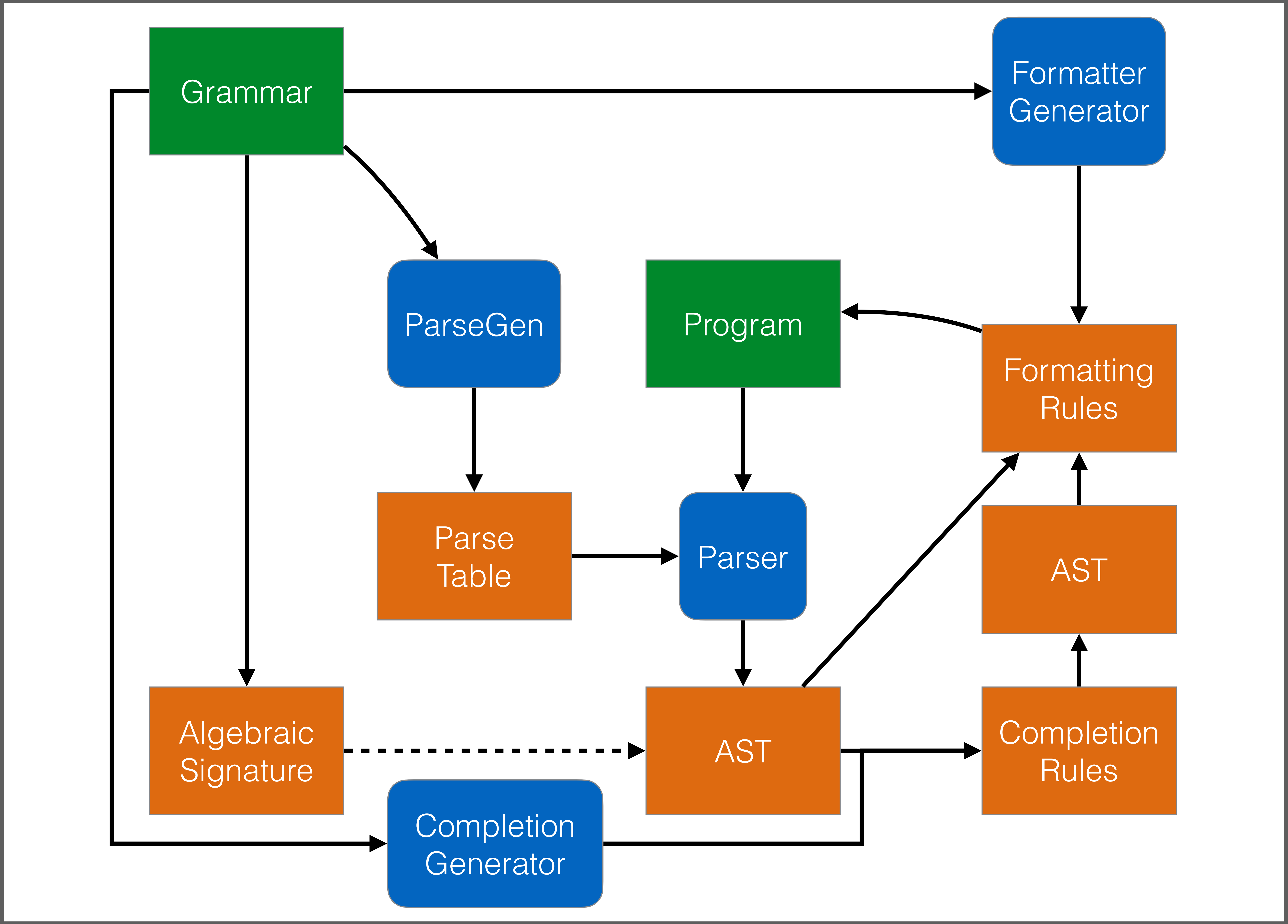
```
a + (let
  x = b
in
  c) + d
```

SDF3 Interpretations

```
Statement.If = <
  if(<Exp>)
    <Statement>
  else
    <Statement>
>
```

- Parser
- Error recovery
- Pretty-printer
- Abstract syntax tree schema
- Syntactic coloring
- Syntactic completion
- Folding rules
- Outline rules

Generating Artifacts from Syntax Definitions



Declarative Type System Specification with Statix

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

Knowing When to Ask

Sound Scheduling of Name Resolution in Type Checkers Derived from Declarative Specifications

ARJEN ROUVOET, Delft University of Technology, The Netherlands
HENDRIK VAN ANTWERPEN, Delft University of Technology, The Netherlands
CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ROBBERT KREBBERS, Radboud University and Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

There is a large gap between the specification of type systems and the implementation of their type checkers, which impedes reasoning about the soundness of the type checker with respect to the specification. A vision to close this gap is to automatically obtain type checkers from declarative programming language specifications. This moves the burden of proving correctness from a case-by-case basis for concrete languages to a single correctness proof for the specification language. This vision is obstructed by an aspect common to all programming languages: name resolution. Naming and scoping are pervasive and complex aspects of the static semantics of programming languages. Implementations of type checkers for languages with name binding features such as modules, imports, classes, and inheritance interleave collection of binding information (i.e., declarations, scoping structure, and imports) and querying that information. This requires scheduling those two aspects in such a way that query answers are stable—i.e., they are computed only after all relevant binding structure has been collected. Type checkers for concrete languages accomplish stability using language-specific knowledge about the type system.

In this paper we give a language-independent characterization of necessary and sufficient conditions to guarantee stability of name and type queries during type checking in terms of *critical edges in an incomplete scope graph*. We use critical edges to give a formal small-step operational semantics to a declarative specification language for type systems, that achieves soundness by delaying queries that may depend on missing information. This yields type checkers for the specified languages that are sound by construction—i.e., they schedule queries so that the answers are stable, and only accept programs that are name- and type-correct according to the declarative language specification. We implement this approach, and evaluate it against specifications of a small module and record language, as well as subsets of Java and Scala.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; **Operational semantics**.

Additional Key Words and Phrases: Name Binding, Type Checker, Statix, Static Semantics, Type Systems

ACM Reference Format:
Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Knowing When to Ask: Sound Scheduling of Name Resolution in Type Checkers Derived from Declarative Specifications. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 180 (November 2020), 28 pages. https://doi.org/10.1145/3428248

Authors' addresses: Arjen Rouvoet, a.j.rouvoet@tudelft.nl, Delft University of Technology, The Netherlands; Hendrik van Antwerpen, h.vanantwerpen@tudelft.nl, Delft University of Technology, The Netherlands; Casper Bach Poulsen, c.b.poulsen@tudelft.nl, Delft University of Technology, The Netherlands; Robbert Krebbers, mail@robbertkrebbers.nl, Radboud University and Delft University of Technology, The Netherlands; Eelco Visser, e.visser@tudelft.nl, Delft University of Technology, The Netherlands.

2475-1421/2020/11-ART180
https://doi.org/10.1145/3428248

Declarative Static Semantics Definition with Statix

Representation

- Scope graph

Specification Formalism: Statix

- Type constraints + scope graph constraints + resolution policies

Declarative Semantics

- Scope graph of program satisfies specification

Language-Independent Tools

- Type checking
- Refactoring / Renaming
- Code completion

Logic Programming

Predicates Represent Program Properties

```
rules // type of ...

  typeOfType : scope * Type  → TYPE
  typeOfExp  : scope * Exp   → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Use `maps` to apply a predicate to all elements of a list

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

`typeOfExp(s, e) == T`

expression `e` has type `T` in scope `s`

`typeOfType(s, t) == T`

syntactic type `t` has semantic type `T` in scope `s`

`declOk(s, d)`

declaration `d` is well-defined (Ok) in scope `s`

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

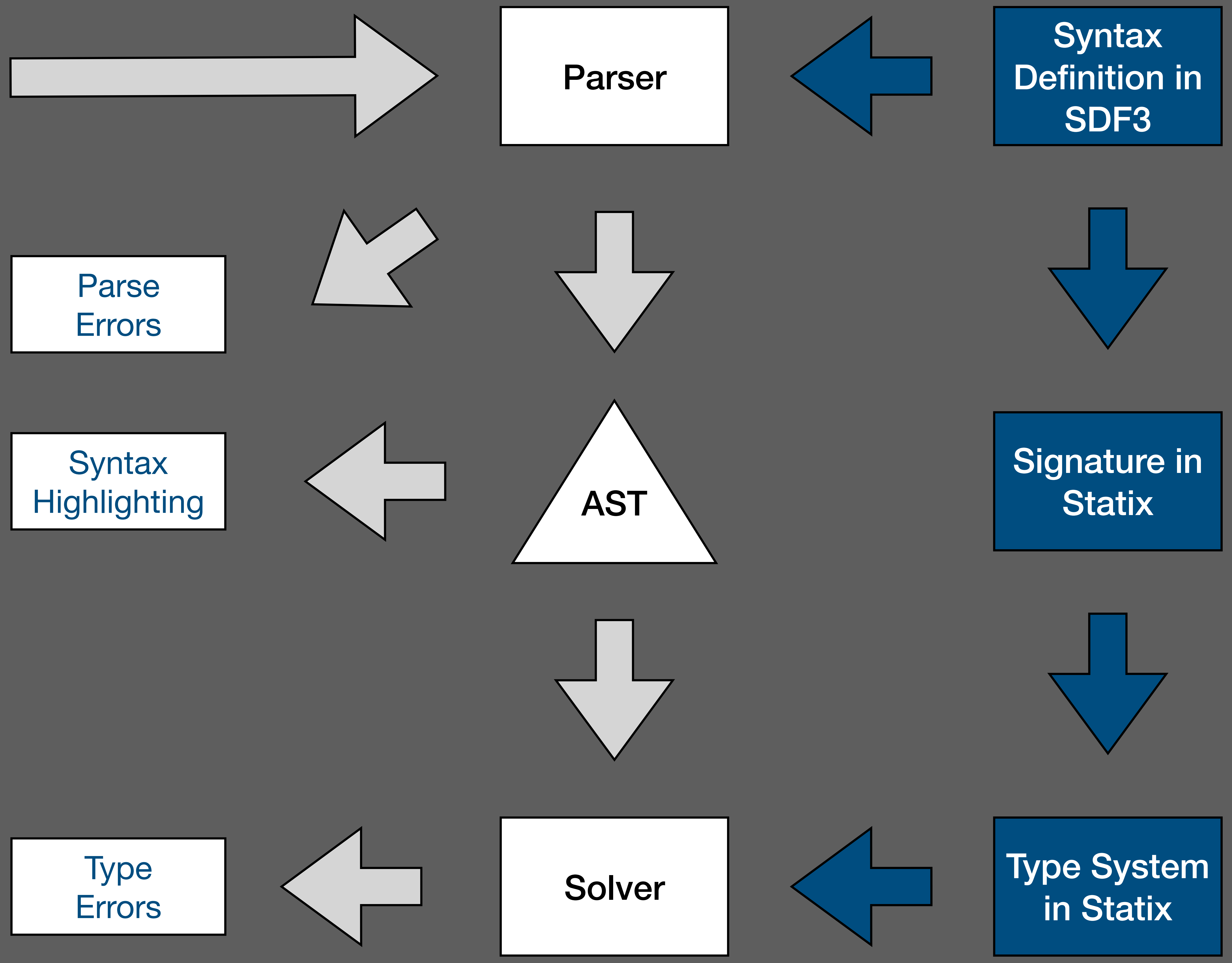
Premises

For all s, e1, e2

If the premises are true, the head is true

From Declarative Definition to Type Checker

```
> 1 + 2 * 3

> true && false

> 1 ^ 2

> true + 4

> 1 && (true || false)

> if 1 == 1 then
    true
  else
    1 == 3

> if 1 == 1 then
    true
  else
    2
```



Programs with Names

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x)
         if x == 0 then true else odd(x - 1)
  }

  module Odd {
    import Even
    def odd = fun(x)
          if x == 0 then false else even(x - 1)
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x)
           Result{ input = x, output = Odd.odd x }
  }

}
```

- Name binding key in programming languages
- Many name binding patterns
- Deal with erroneous programs
- Name resolution complicates type checkers, compilers
- Ad hoc non-declarative treatment
- A systematic, uniform approach to name resolution?

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

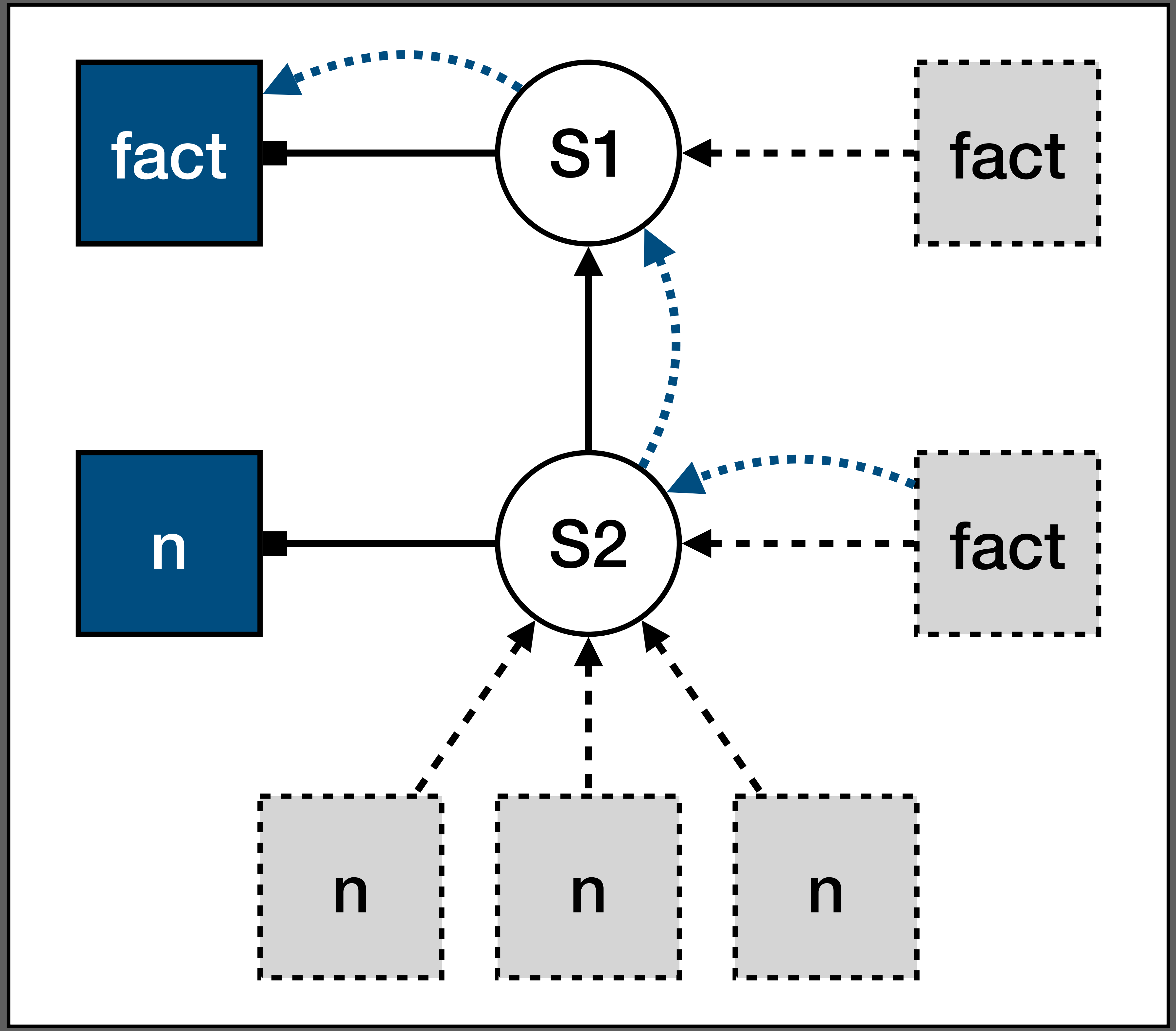
J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 205–231, 2015.
DOI: 10.1007/978-3-662-46669-8_9

Name Resolution with Scope Graphs

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

Declaring and Resolving Names

Declarations and References

```
signature
  constructors
    Var   : ID → Exp
    Def   : Bind → Decl
    Bind  : ID * Exp → Bind
```

```
rules

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
```

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  typeOfExp(s, Var(x)) = typeOfVar(s, x).

  declOk(s, Def(bind)) :-
    bindOk(s, s, bind).

  bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}
    typeOfExp(s_ctx, e) == T,
    declareVar(s_bnd, x, T).
```

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE
```

Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE
```

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

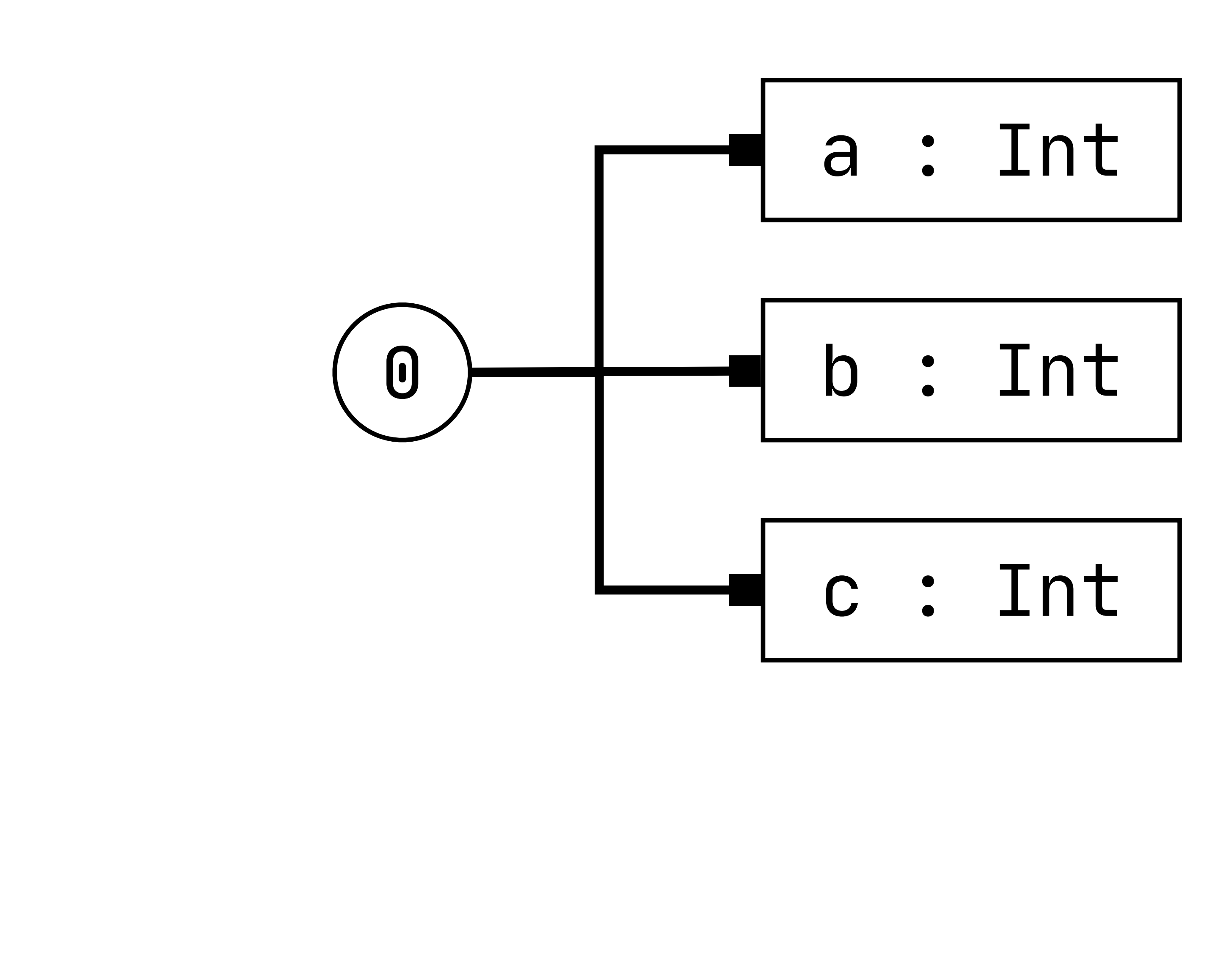
declaration relation

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference

```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T.
```

variable x is declared in scope s with type T



Representing Name Binding with Scope Graphs

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter e
  relations
    typeOfDecl : occurrence → TYPE
```

namespace

resolution policy

declaration relation

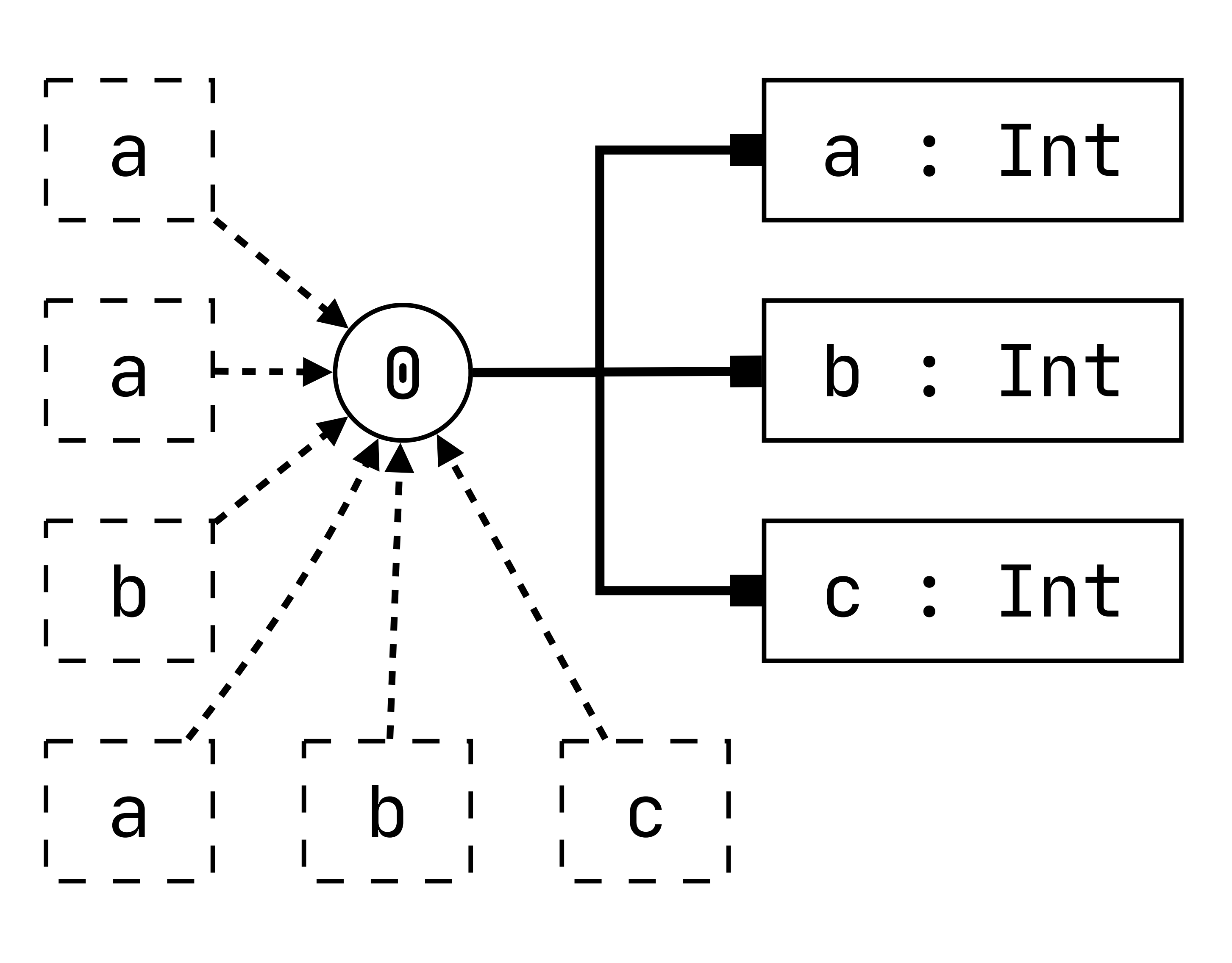
```
rules

  declareVar : scope * string * TYPE
  typeOfVar  : scope * string → TYPE

  declareVar(s, x, T) :-
    s → Var{x} with typeOfDecl T.

  typeOfVar(s, x) = T :- {x'}
    typeOfDecl of Var{x} in s ↦ [(_, (Var{x'}, T))].
```

variable x is declared in scope s with type T

variable x in scope s resolves to declaration x' with type T

```
def a = 0
def b = a + 1
def c = a + b
> a + b + c
```

declaration and reference



How about shadowing?

Lexical Scope

New Scope and Scope Edge Constraints

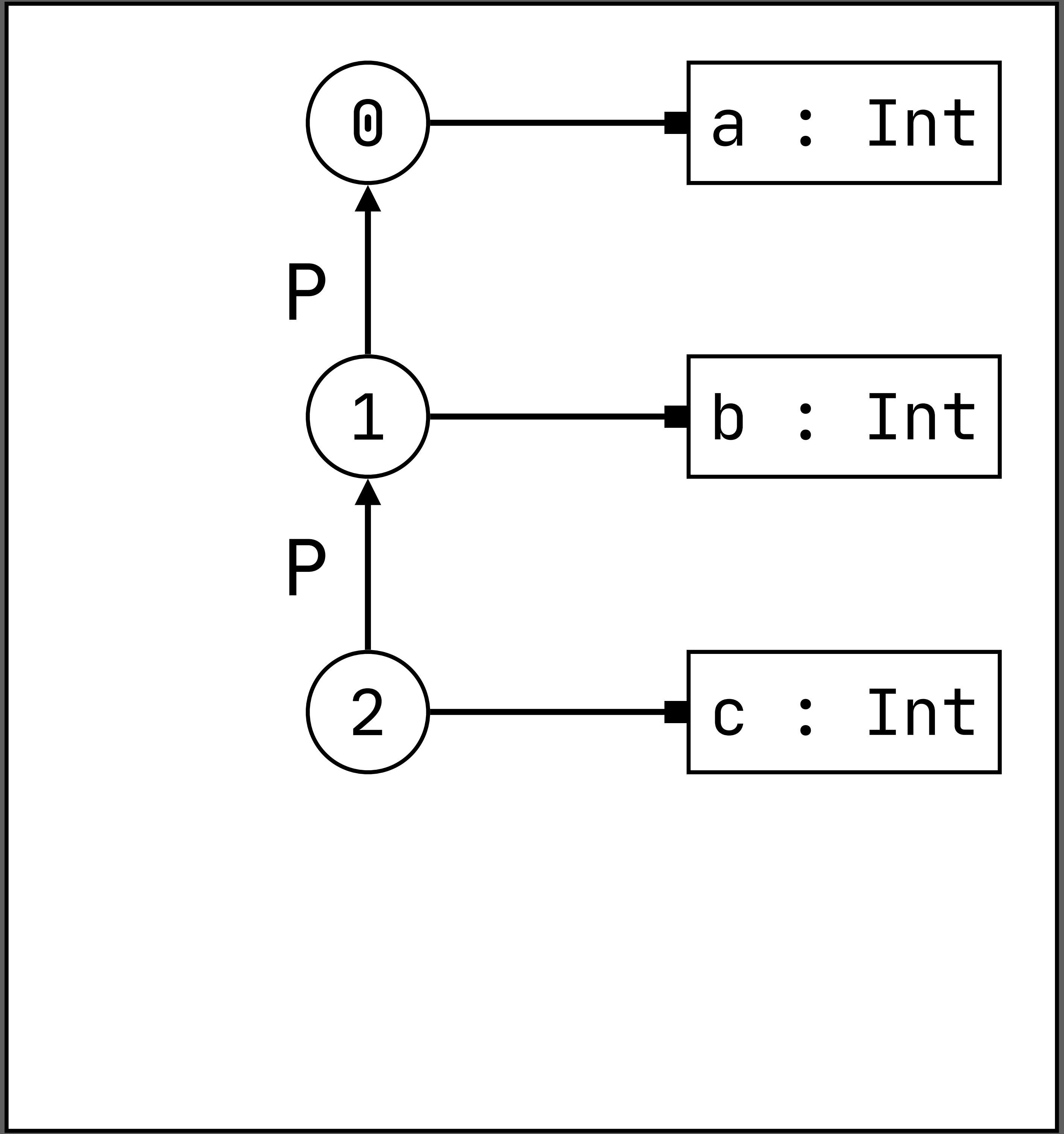
```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let,
     s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

scope edge



Path Wellformedness: Reachability

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let,
     s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

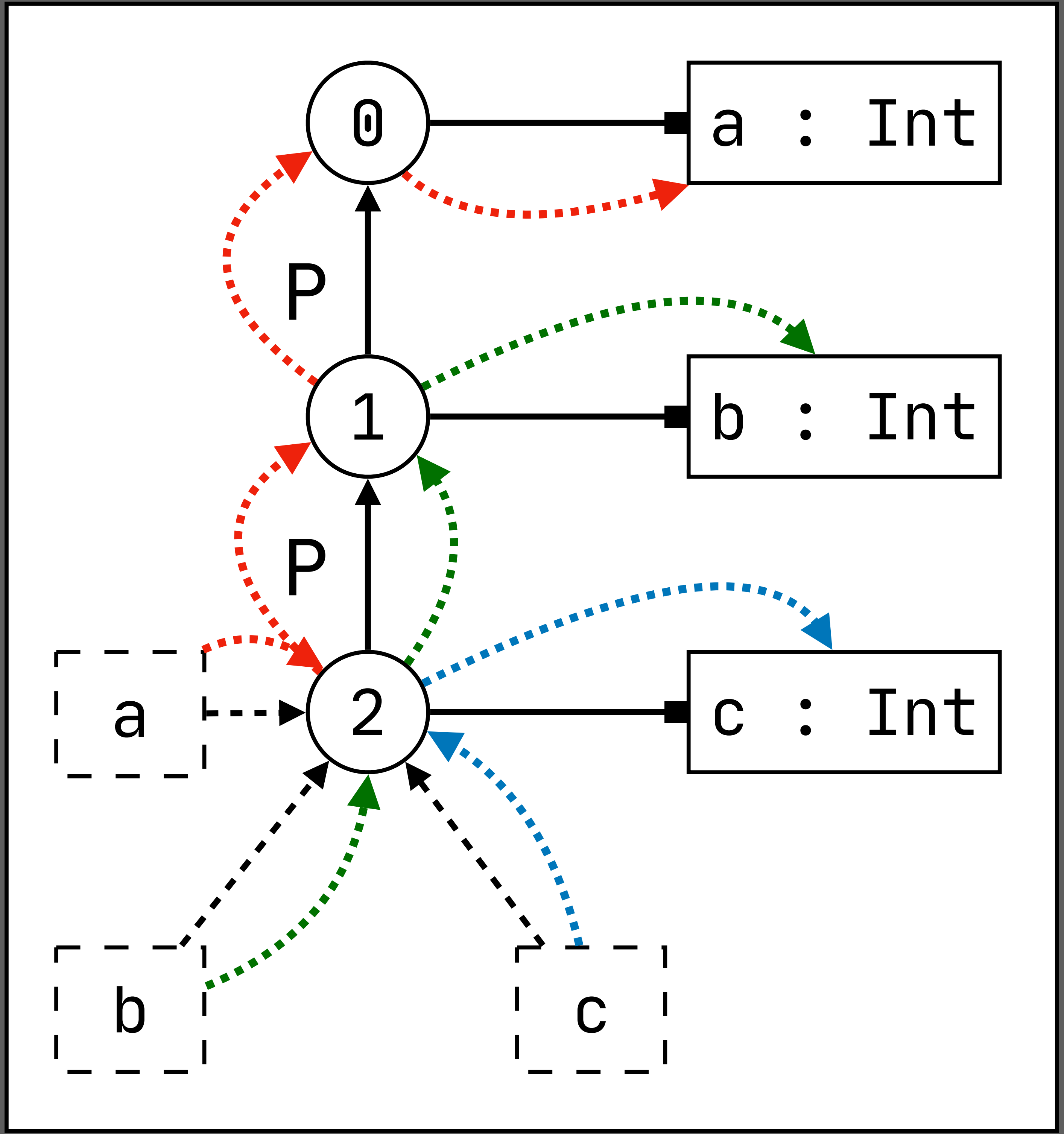
new scope

scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P*
```

path $P*$ allows resolution through zero or more P edges

```
let a = 1 in
let b = 2 in
let c = 3 in
  a + b + c
```



Path Specificity: Visibility (Shadowing)

```
signature
  constructors
    Let : ID * Exp * Exp → Exp
```

```
rules

  typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
     typeOfExp(s, e1) == S,
     new s_let,
     s_let -P→ s,
     declareVar(s_let, x, S),
     typeOfExp(s_let, e2) == T.
```

new scope

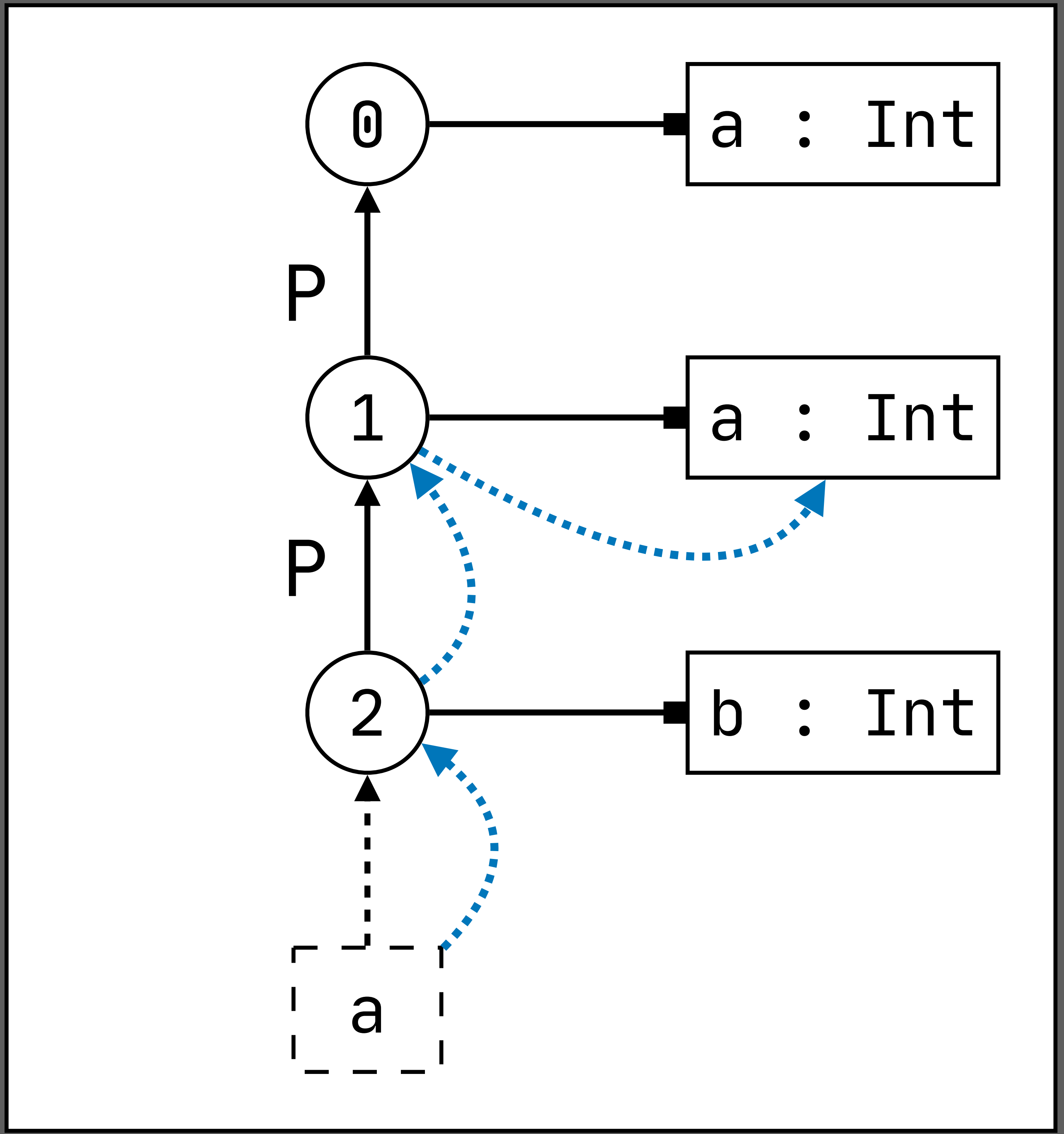
scope edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var filter P* min $ < P
```

path *P** allows resolution through zero or more *P* edges

prefer local scope ($) over parent scope (*P*)

```
let a = 1 in
let a = 2 in
let b = 3 in
  a
```



How about non-lexical bindings?

Non-Lexical Scope (Modules)

Modules: Scopes as Types

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

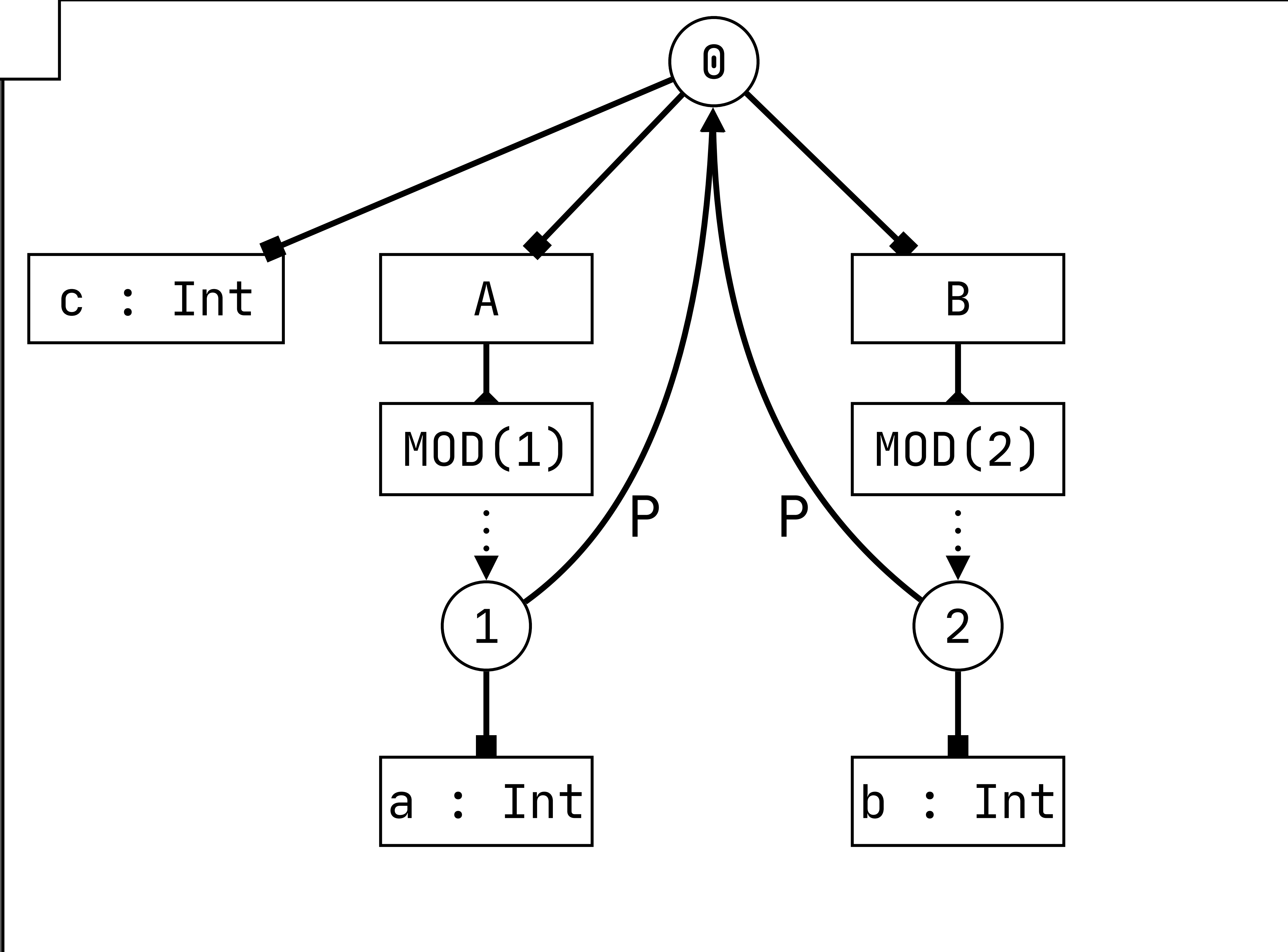
```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).
```

lexical scope

scope as type

```
signature
  namespaces
    Mod  : string
```



Resolving Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

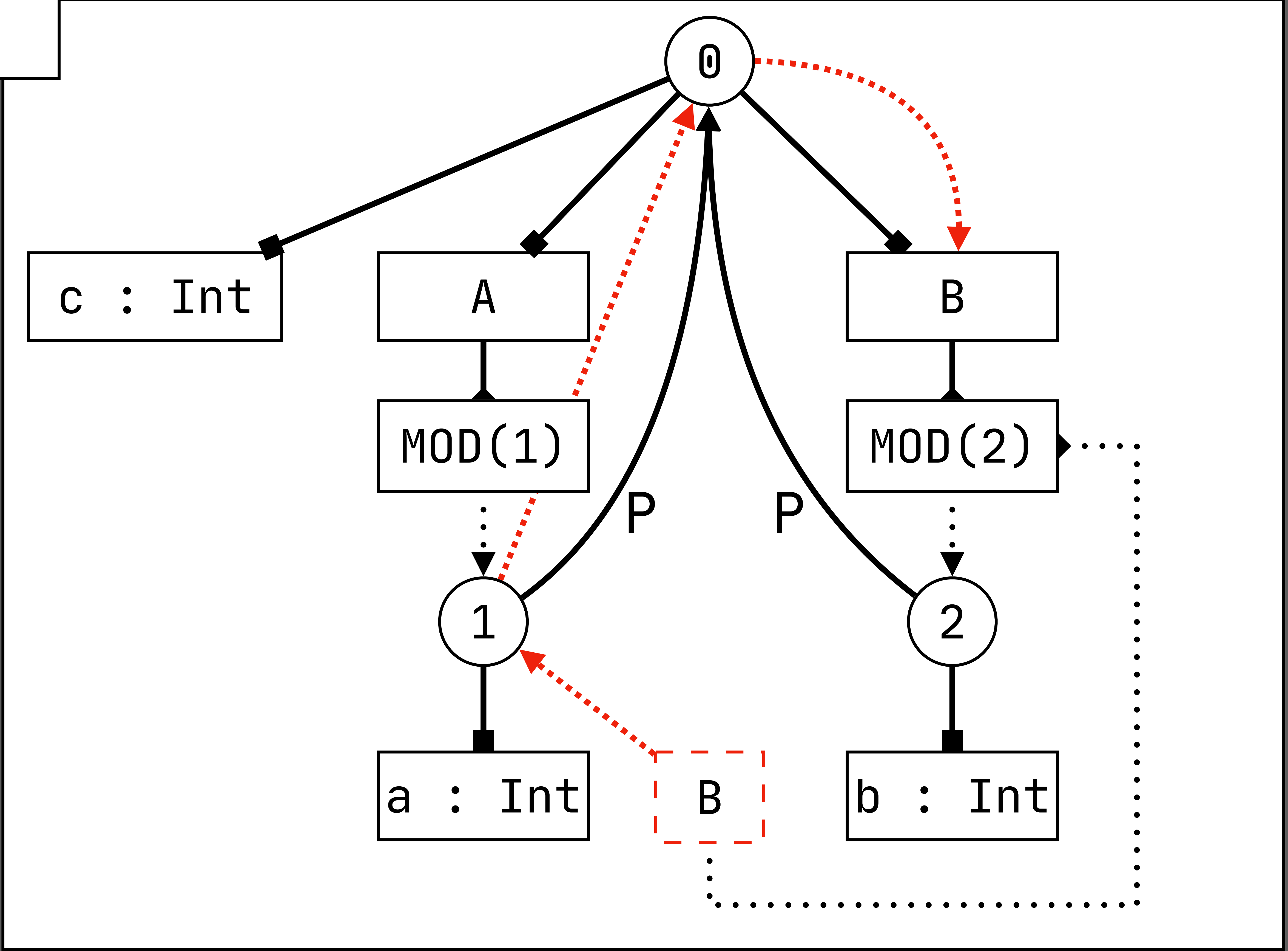
lexical scope

scope as type

resolve import

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P*
      min $ < I, $ < P, I < P
```

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Import Edge

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

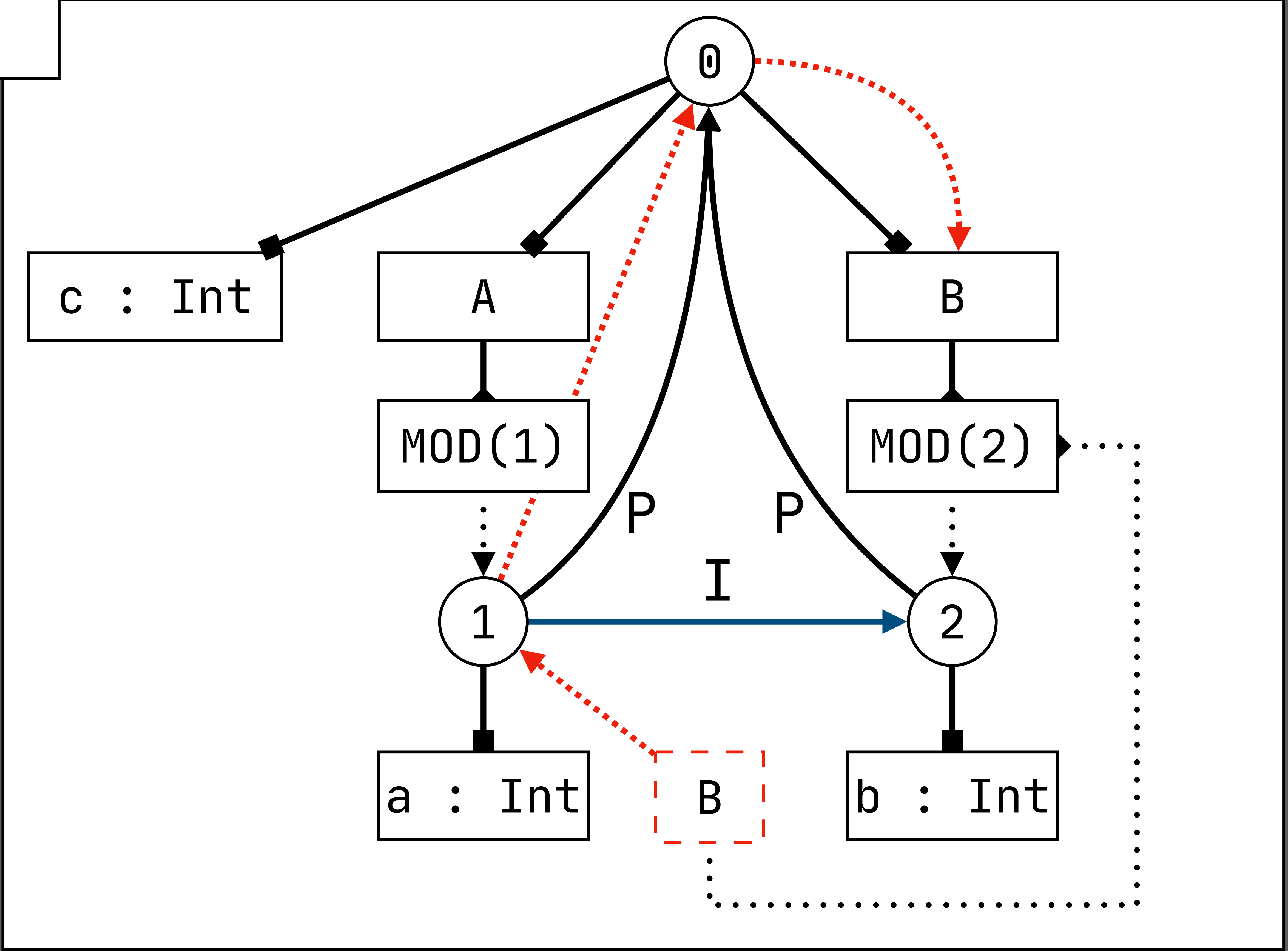
scope as type

resolve import

import edge

```
signature
  namespaces
    Mod  : string
  name-resolution
    resolve Mod
      filter P*
      min $ < I, $ < P, I < P
```

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Resolving through Import Edge

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

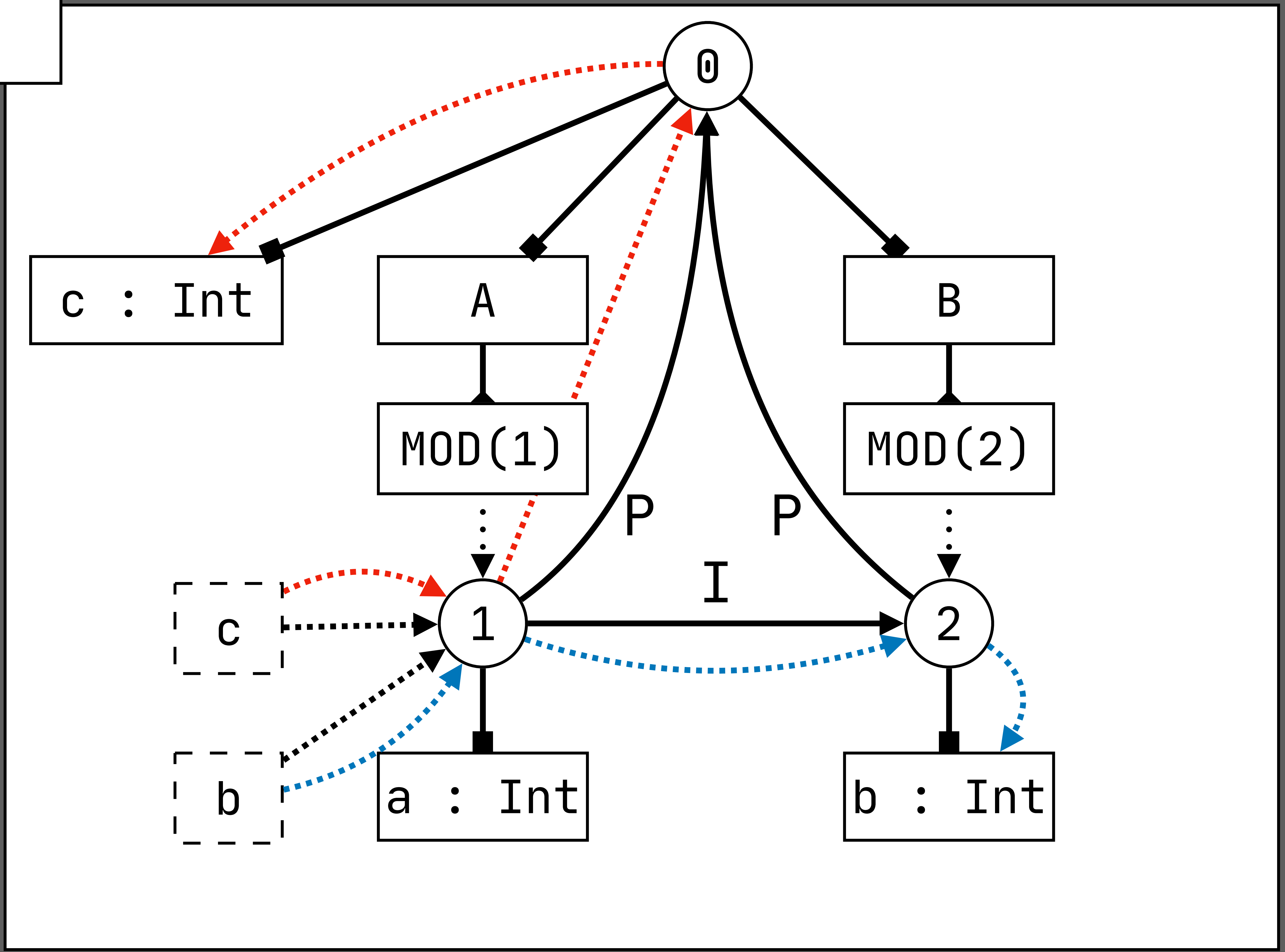
resolve import

import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```



Import vs Parent

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

lexical scope

scope as type

resolve import

import edge

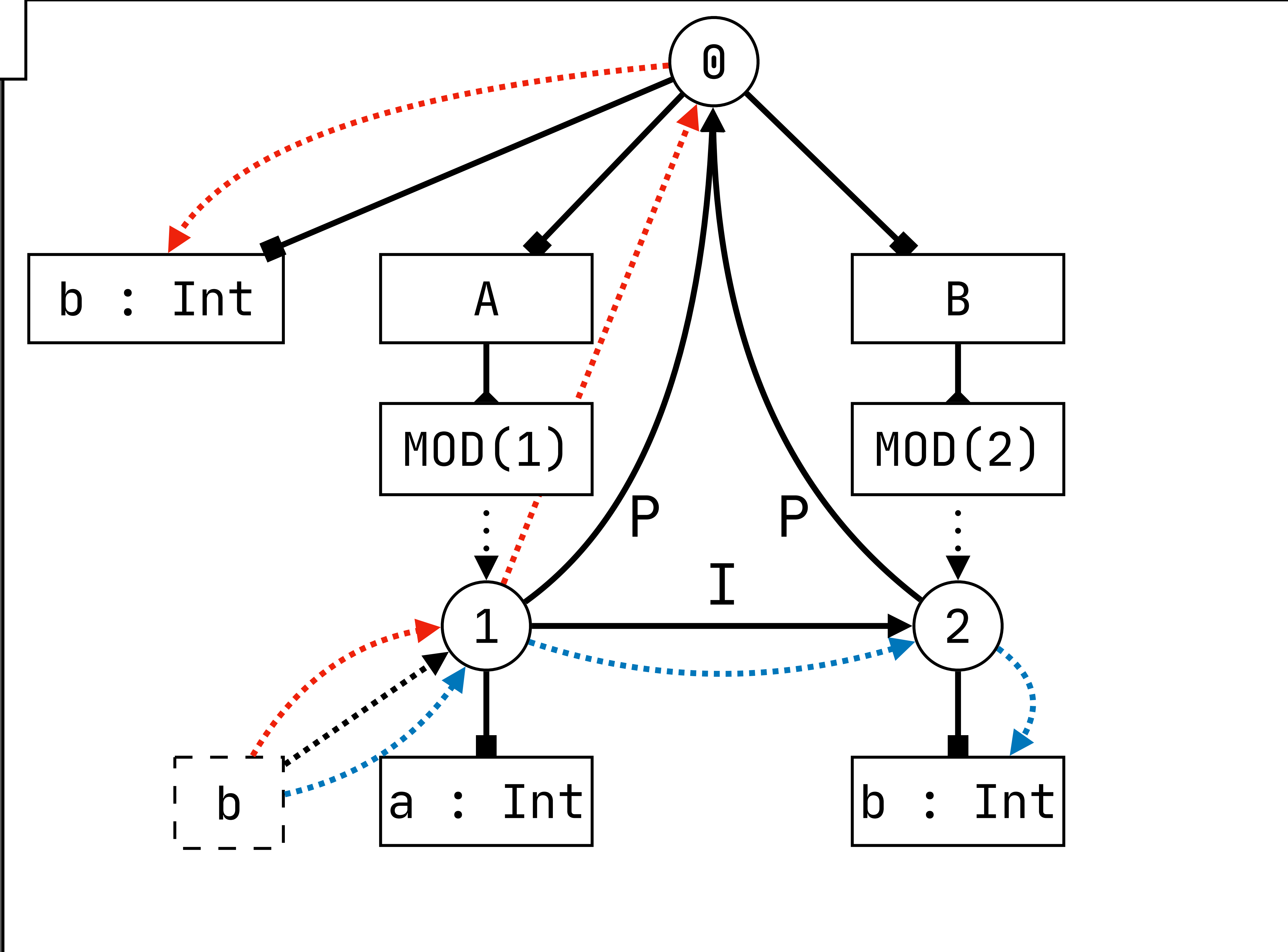
```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

prefer import

```
def b = 0
module A {
  import B
  def a = b
}
module B {
  def b = 2
}
```

prefer blue path over red path



Mutual Imports

```
signature
  constructors
    MOD     : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

scope as type

resolve import

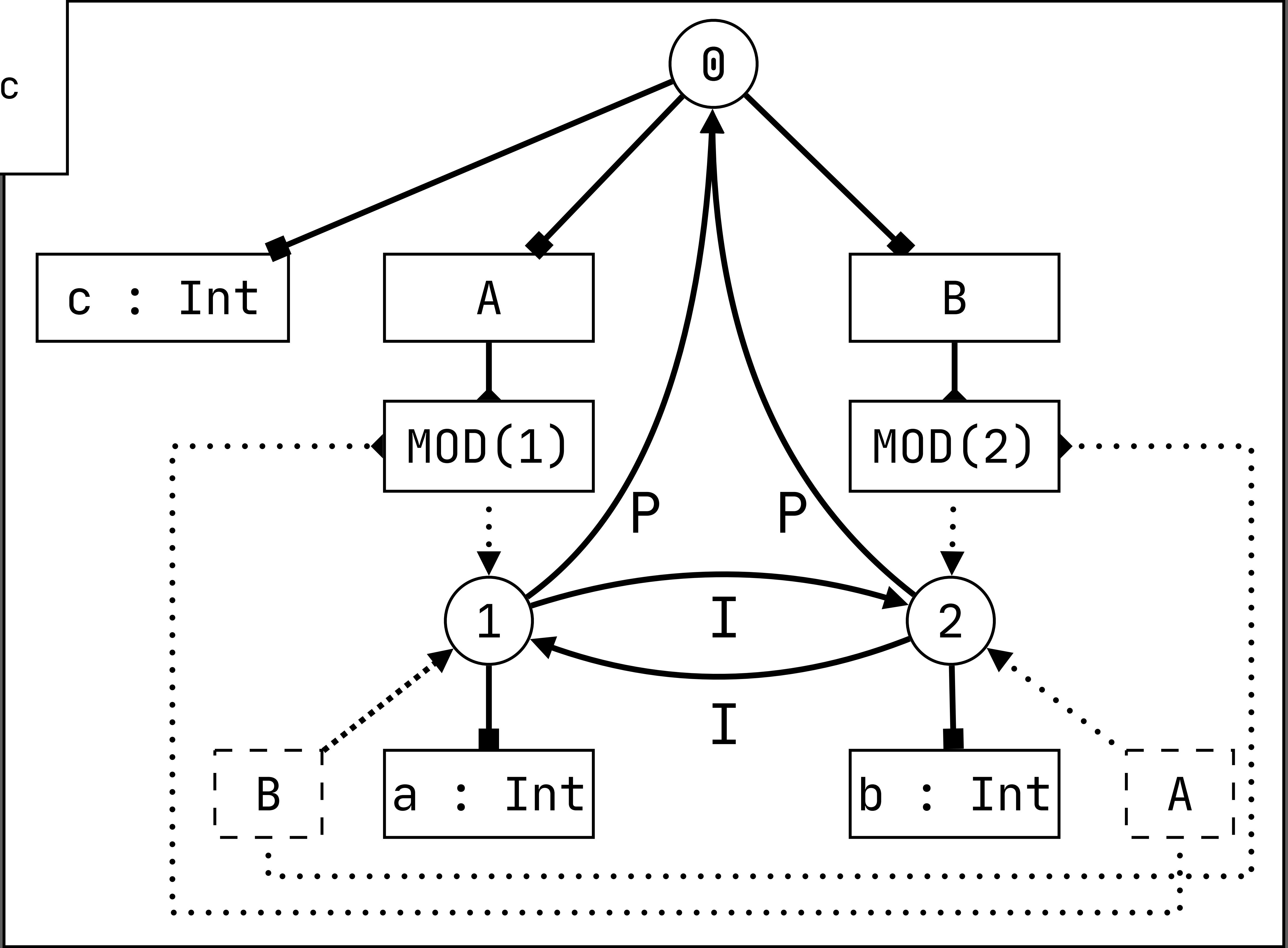
import edge

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

import after parent

prefer import

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```



Mutual Imports

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

scope as type

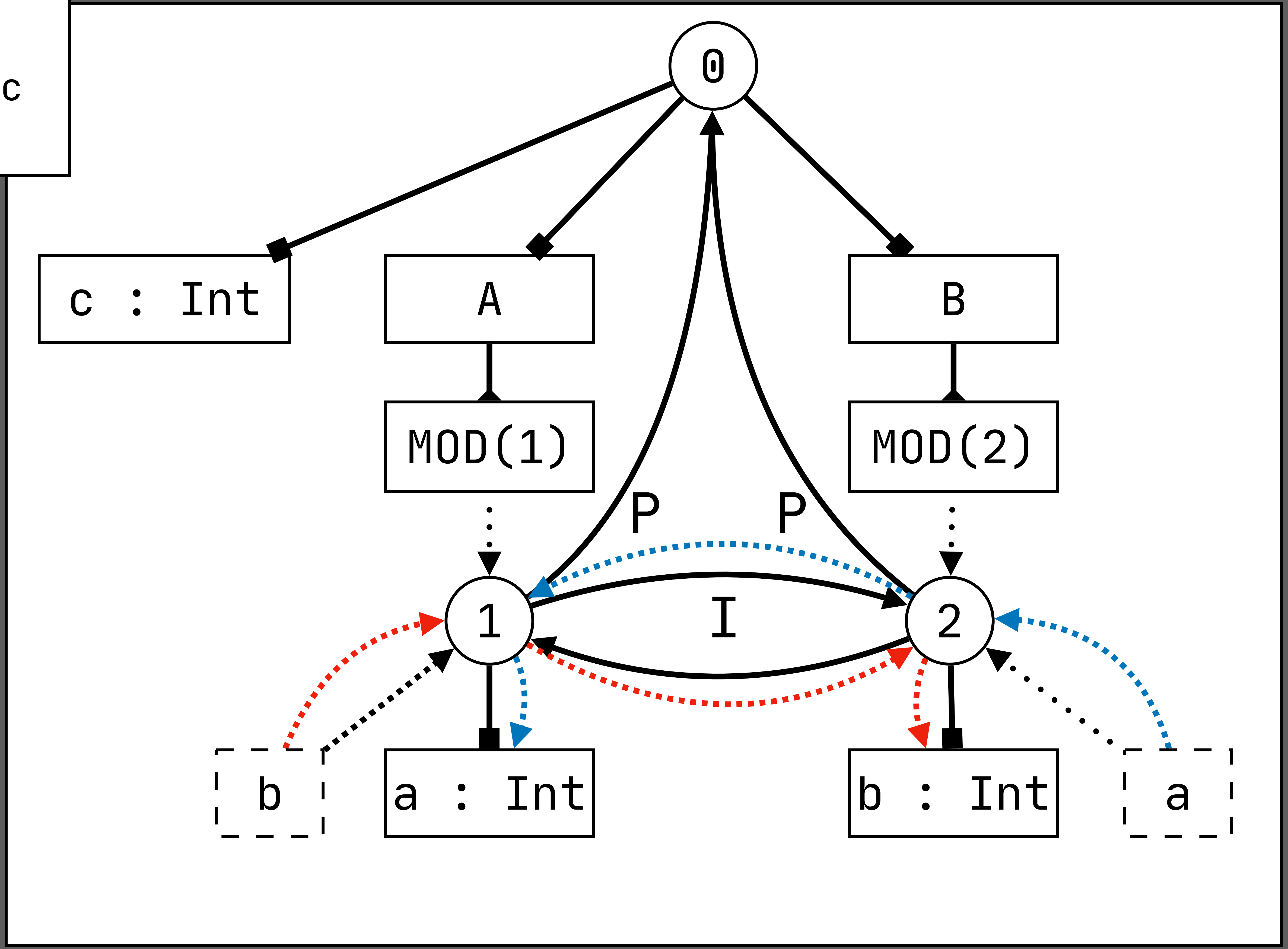
```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

scope as type

resolve import

import edge




```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

resolve through import edges

prefer import

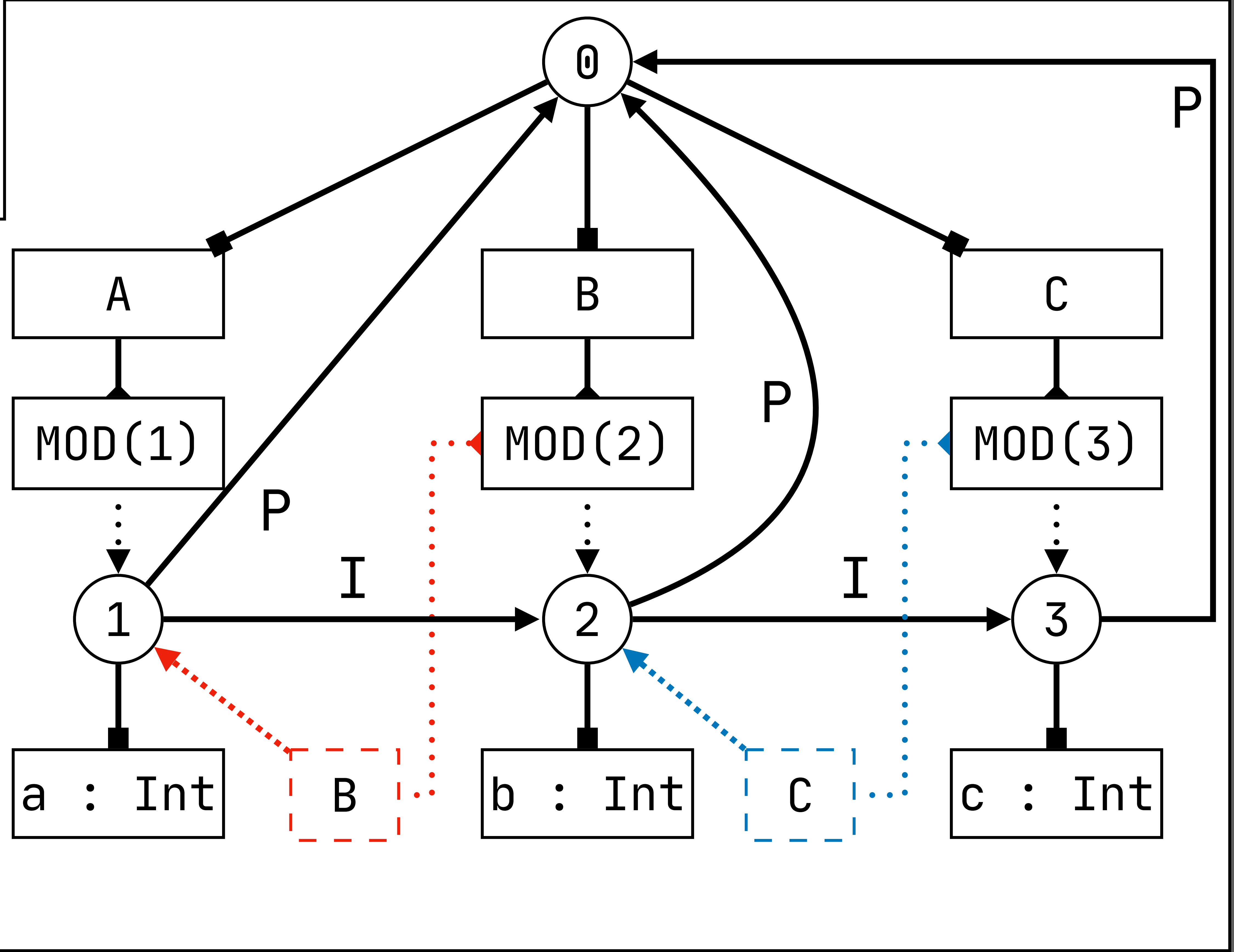
Transitive Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P-> s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I-> s_mod.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

```
module A {
  import B
  def a = b + c
}
module B {
  import C
  def b = c + 2
}
module C {
  def c = 1
}
```



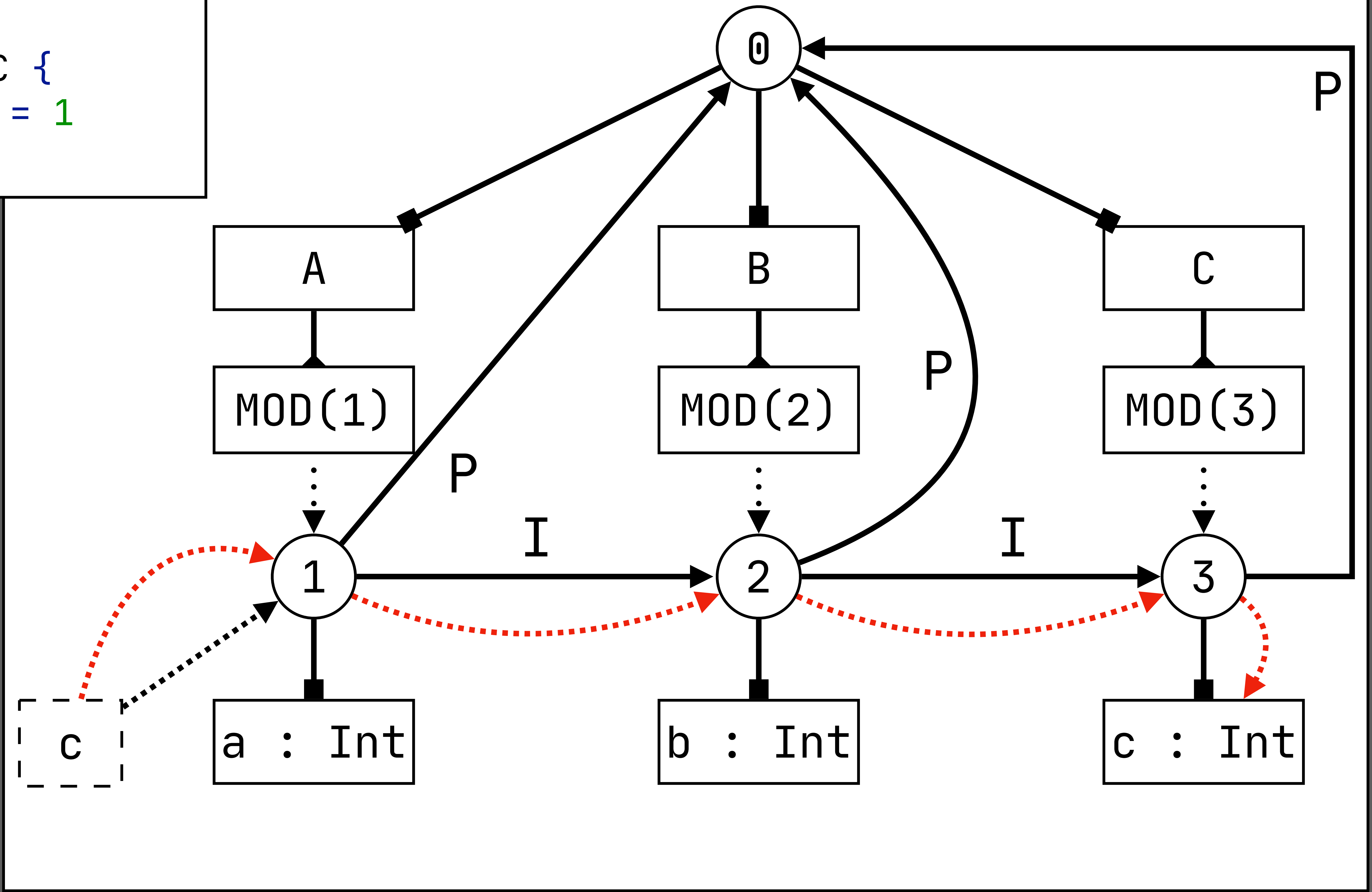
Transitive Import

```
signature
  constructors
    MOD    : scope → TYPE
    Module : ID * list(Decl) → Decl
    Import : ID → Decl
```

```
rules

  declOk(s, Module(m, decls)) :- {s_mod}
    new s_mod, s_mod -P→ s,
    declareMod(s, m, MOD(s_mod)),
    declsOk(s_mod, decls).

  declOk(s, Import(p)) :- {s_mod s_end}
    typeOfModRef(s, p) == MOD(s_mod),
    s -I→ s_mod.
```

```
signature
  namespaces
    Var : string
  name-resolution
    resolve Var
      filter P* I*
      min $ < I, $ < P, I < P
```

```
module A {
  import B
  def a = b + c
}
module B {
  import C
  def b = c + 2
}
module C {
  def c = 1
}
```



Statix Interpretations

Statix Interpretations (In Progress)

Declarative Semantics [OOPSLA'18]

- `G ⊨ programOk(s, p)`
- Does program `p` satisfy the `programOk` predicate in scope `s`, given scope graph `G`?

Type Checking

- Given a program term `p`, what is valid scope graph `G`?
- Operational semantics is safe wrt declarative semantics [OOPSLA'20]
- Type check programs concurrently and/or incrementally

Code Completion [ECOOP'19]

- Given a hole (placeholder) in an incomplete program, what are valid completions?

Renaming

- Given a name x in a program, can it be renamed to y, without being captured?

Quick Fixes

- Given a name/type error in a program, what is repair that would solve the error?

Random Term Generation

- Given a placeholder (and type), randomly generate a program that is syntactically, binding, and type correct

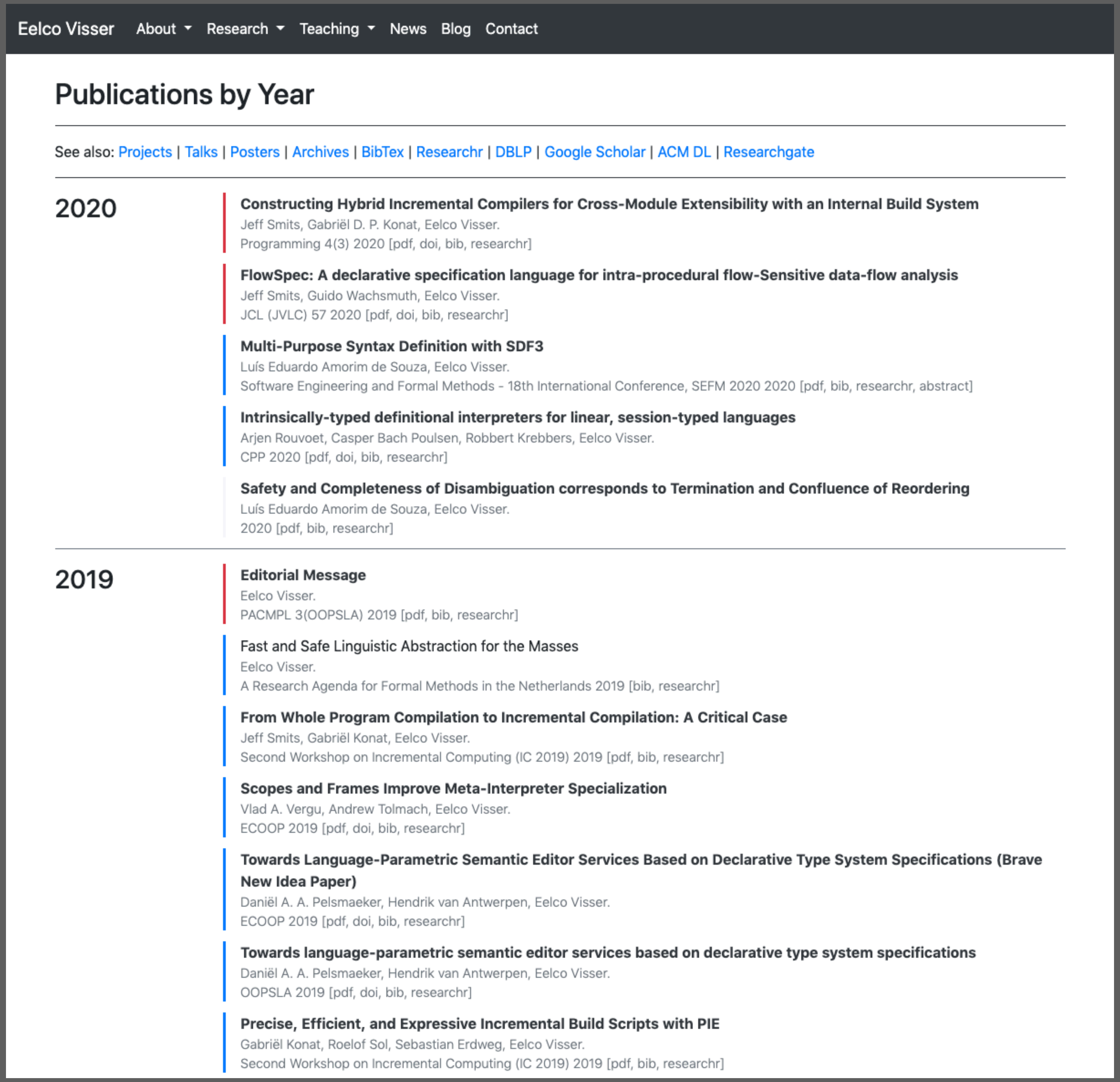
Conclusion


Language Design
Syntax Definition
Static Semantics
Dynamic Semantics
Transform
Desktop — bash — 37×16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
Fib.java
public class Fib {
public static int calc(int n) {
if(n < 2)
}
}
The Java™ Language Specification
Java SE 7 Edition
2012-07-27
Describing the Semantics of Java and Proving Type Soundness
Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
Multi-purpose Declarative Meta-Languages

Multi-purpose Declarative Meta-Languages


Language Design
SDF3
Statix
DynSem
Dynamix
Stratego
Desktop — bash — 37×16
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
Fib.java
public class Fib {
public static int calc(int n) {
if(n < 2)
}
}
The Java™ Language
Specification
Java SE 7 Edition
2012-07-27
Describing the Semantics of Java
and Proving Type Soundness
Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
no formal definition. Java adopts the Smalltalk [15] approach whereby all object
variables are implicitly pointers.
Furthermore, although there are a large number of studies of the seman-
tics of isolated programming language features or of minimal programming lan-
guages [1], [31], [33], there have not been many studies of the formal semantics of
actual programming languages. In addition, the interplay of features which are
very well understood in isolation, might introduce unexpected effects.
Multi-purpose Declarative Meta-Languages

More Information



Eelco Visser About Research Teaching News Blog Contact

Publications by Year

See also: Projects | Talks | Posters | Archives | BibTex | Researchr | DBLP | Google Scholar | ACM DL | Researchgate

2020

Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System
Jeff Smits, Gabriël D. P. Konat, Eelco Visser.
Programming 4(3) 2020 [pdf, doi, bib, researchr]

FlowSpec: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis
Jeff Smits, Guido Wachsmuth, Eelco Visser.
JCL (JVLC) 57 2020 [pdf, doi, bib, researchr]

Multi-Purpose Syntax Definition with SDF3
Luís Eduardo Amorim de Souza, Eelco Visser.
Software Engineering and Formal Methods - 18th International Conference, SEFM 2020 2020 [pdf, bib, researchr, abstract]

Intrinsically-typed definitional interpreters for linear, session-typed languages
Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, Eelco Visser.
CPP 2020 [pdf, doi, bib, researchr]

Safety and Completeness of Disambiguation corresponds to Termination and Confluence of Reordering
Luís Eduardo Amorim de Souza, Eelco Visser.
2020 [pdf, bib, researchr]

2019

Editorial Message
Eelco Visser.
PACMPL 3(OOPSLA) 2019 [pdf, bib, researchr]

Fast and Safe Linguistic Abstraction for the Masses
Eelco Visser.
A Research Agenda for Formal Methods in the Netherlands 2019 [bib, researchr]

From Whole Program Compilation to Incremental Compilation: A Critical Case
Jeff Smits, Gabriël Konat, Eelco Visser.
Second Workshop on Incremental Computing (IC 2019) 2019 [pdf, bib, researchr]

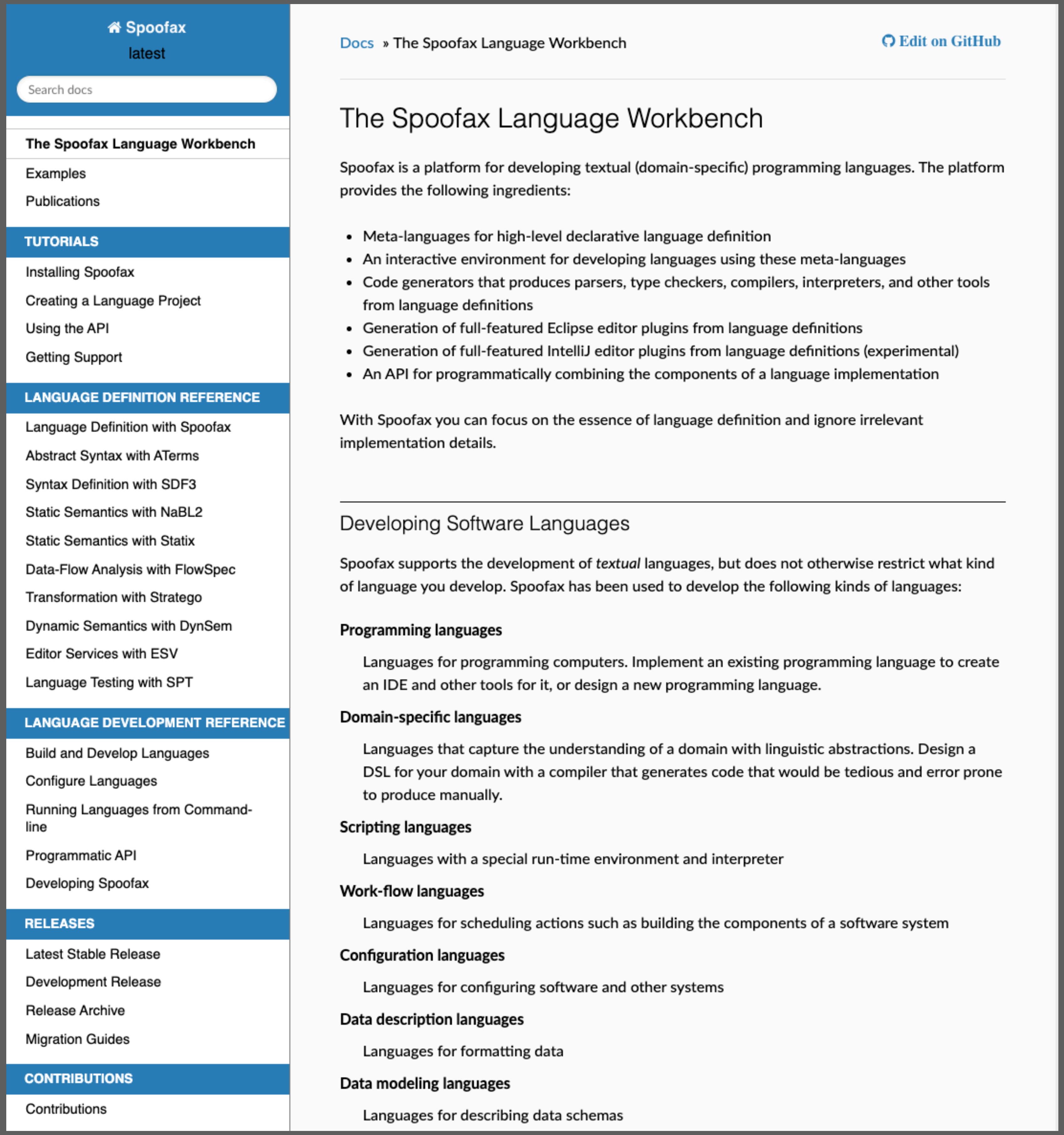
Scopes and Frames Improve Meta-Interpreter Specialization
Vlad A. Vergu, Andrew Tolmach, Eelco Visser.
ECOOP 2019 [pdf, doi, bib, researchr]

Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)
Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Eelco Visser.
ECOOP 2019 [pdf, doi, bib, researchr]

Towards language-parametric semantic editor services based on declarative type system specifications
Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Eelco Visser.
OOPSLA 2019 [pdf, doi, bib, researchr]

Precise, Efficient, and Expressive Incremental Build Scripts with PIE
Gabriël Konat, Roelof Sol, Sebastian Erdweg, Eelco Visser.
Second Workshop on Incremental Computing (IC 2019) 2019 [pdf, bib, researchr]

http://eelcovisser.org



Spoofax
latest
Search docs

The Spoofax Language Workbench
Examples
Publications

TUTORIALS
Installing Spoofax
Creating a Language Project
Using the API
Getting Support

LANGUAGE DEFINITION REFERENCE
Language Definition with Spoofax
Abstract Syntax with ATerms
Syntax Definition with SDF3
Static Semantics with NaBL2
Static Semantics with Statix
Data-Flow Analysis with FlowSpec
Transformation with Stratego
Dynamic Semantics with DynSem
Editor Services with ESV
Language Testing with SPT

LANGUAGE DEVELOPMENT REFERENCE
Build and Develop Languages
Configure Languages
Running Languages from Command-line
Programmatic API
Developing Spoofax

RELEASES
Latest Stable Release
Development Release
Release Archive
Migration Guides

CONTRIBUTIONS
Contributions

Docs » The Spoofax Language Workbench — Edit on GitHub

The Spoofax Language Workbench

Spoofax is a platform for developing textual (domain-specific) programming languages. The platform provides the following ingredients:

- Meta-languages for high-level declarative language definition
- An interactive environment for developing languages using these meta-languages
- Code generators that produces parsers, type checkers, compilers, interpreters, and other tools from language definitions
- Generation of full-featured Eclipse editor plugins from language definitions
- Generation of full-featured IntelliJ editor plugins from language definitions (experimental)
- An API for programmatically combining the components of a language implementation

With Spoofax you can focus on the essence of language definition and ignore irrelevant implementation details.

Developing Software Languages

Spoofax supports the development of *textual* languages, but does not otherwise restrict what kind of language you develop. Spoofax has been used to develop the following kinds of languages:

Programming languages
Languages for programming computers. Implement an existing programming language to create an IDE and other tools for it, or design a new programming language.

Domain-specific languages
Languages that capture the understanding of a domain with linguistic abstractions. Design a DSL for your domain with a compiler that generates code that would be tedious and error prone to produce manually.

Scripting languages
Languages with a special run-time environment and interpreter

Work-flow languages
Languages for scheduling actions such as building the components of a software system

Configuration languages
Languages for configuring software and other systems

Data description languages
Languages for formatting data

Data modeling languages
Languages for describing data schemas

http://metaborg.org