Declare Your Language

Chapter 9: Dynamic Semantics

Eelco Visser

IN4303 Compiler Construction
TU Delft
November 2017

TU Delft

Outline

The meaning of programs

Operational semantics

DynSem: A DSL for dynamic semantics specification

Interpreter generation

Scopes describe frames

Reading Material

Executable specification of dynamic semantics

RTA 2015

http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365

DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, and Eelco Visser

Delft University of Technology
Delft, The Netherlands
{v.a.vergu|p.j.m.neron|visser}@tudelft.nl

Abstract

The formal semantics of a programming language and its implementation are typically separately defined, with the risk of divergence such that properties of the formal semantics are not properties of the implementation. In this paper, we present DynSem, a domain-specific language for the specification of the dynamic semantics of programming languages that aims at supporting both formal reasoning and efficient interpretation. DynSem supports the specification of the *operational semantics* of a language by means of *statically typed conditional term reduction rules.* DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors. DynSem supports *modular* specification by adopting implicit propagation of semantic components from I-MSOS, which allows omitting propagation of components such as environments and stores from rules that do not affect those. DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages (D.3.1.)

Keywords and phrases programming languages, dynamic semantics, reduction semantics, semantics engineering, IDE, interpreters, modularity

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.365

1 Introduction

The specification of the dynamic semantics is the core of a programming language design as it describes the runtime behavior of programs. In practice, the implementation of a compiler or an interpreter for the language often stands as the *only* definition of the semantics of a language. Such implementations, in a traditional programming language, often lack the clarity and the conciseness that a specification in a formal semantics framework provides. Therefore, they are a poor source of *documentation* about the semantics. On the other hand, formal definitions are not executable to the point that they can be used as implementations to run programs. Even when both a formal specification and an implementation co-exist, they typically diverge. As a result, important properties of a language as established based on its formal semantics may not hold for its implementation. Our goal is to unify the semantics engineering and language engineering of programming language designs [22] by providing a notation for the specification of the dynamic semantics that can serve both as a readable formalization as well as the source of an execution engine.

In this paper, we present DynSem, a DSL for the concise, modular, statically typed, and executable specification of the dynamic semantics of programming languages. DynSem

© Vlad Vergu, Pierre Neron, and Eelco Visser;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 365–378
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Alignment between static binding/type system and dynamic memory layout

For the interested

ECOOP 2016

http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20

Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics (Artifact)*

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[3], and Eelco Visser[4]

1 Delft University of Technology
c.b.poulsen@tudelft.nl
2 French Network and Information Security Agency (ANSSI)
pierre.neron@ssi.gouv.fr
3 Portland State University
tolmach@pdx.edu
4 Delft University of Technology
visser@acm.org

Abstract

Our paper introduces a systematic approach to the alignment of names in the static structure of a program, and memory layout and access during its execution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

This Coq artifact showcases how our uniform model for memory layout in dynamic semantics provides structure to type soundness proofs. The artifact contains type soundness proofs mechanized in Coq for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent framework formalizing scope graphs and frame heaps.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics

Digital Object Identifier 10.4230/DARTS.2.1.10

Related Article Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser, "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics", in Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), LIPIcs, Vol. 56, pp. 20:1–20:26, 2016.
http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20

Related Conference 30th European Conference on Object-Oriented Programming (ECOOP 2016), July 18–22, 2016, Rome, Italy

1 Scope

The artifact is designed to document and support repeatability of the type soundness proofs in the companion paper [2], using the Coq proof assistant.[1] In particular, the artifact provides a

* This work was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

1 https://coq.inria.fr/

© Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, Eelco Visser;
licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)
Dagstuhl Artifacts Series, Vol. 2, Issue 1, Artifact No. 10, pp. 10:1–10:3
Dagstuhl Artifacts Series
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Automatically checking type safety of interpreters.

Just out of the oven

POPL 2018

http://casperbp.net/store/intrinsicallytyped.pdf

https://doi.org/10.1145/3158104

Intrinsically-Typed Definitional Interpreters for Imperative Languages

CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ARJEN ROUVOET, Delft University of Technology, The Netherlands
ANDREW TOLMACH, Portland State University, USA
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

A definitional interpreter defines the semantics of an object language in terms of the (well-known) semantics of a host language, enabling understanding and validation of the semantics through execution. Combining a definitional interpreter with a separate type system requires a separate type safety proof. An alternative approach, at least for pure object languages, is to use a dependently-typed language to encode the object language type system in the definition of the abstract syntax. Using such intrinsically-typed abstract syntax definitions allows the host language type checker to verify automatically that the interpreter satisfies type safety. Does this approach scale to larger and more realistic object languages, and in particular to languages with mutable state and objects?

In this paper, we describe and demonstrate techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle rich object languages with non-trivial binding structures and mutable state. While the resulting interpreters are certainly more complex than the simply-typed λ-calculus interpreter we start with, we claim that they still meet the goals of being concise, comprehensible, and executable, while guaranteeing type safety for more elaborate object languages. We make the following contributions: (1) A *dependent-passing style* technique for hiding the weakening of indexed values as they propagate through monadic code. (2) An Agda library for programming with *scope graphs* and *frames*, which provides a uniform approach to dealing with name binding in intrinsically-typed interpreters. (3) Case studies of intrinsically-typed definitional interpreters for the simply-typed λ-calculus with references (STLC+Ref) and for a large subset of Middleweight Java (MJ).

CCS Concepts: • **Theory of computation** → **Program verification**; *Type theory*; • **Software and its engineering** → **Formal language definitions**;

Additional Key Words and Phrases: definitional interpreters, dependent types, scope graphs, mechanized semantics, Agda, type safety, Java

ACM Reference Format:
Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (January 2018), 34 pages. https://doi.org/10.1145/3158104

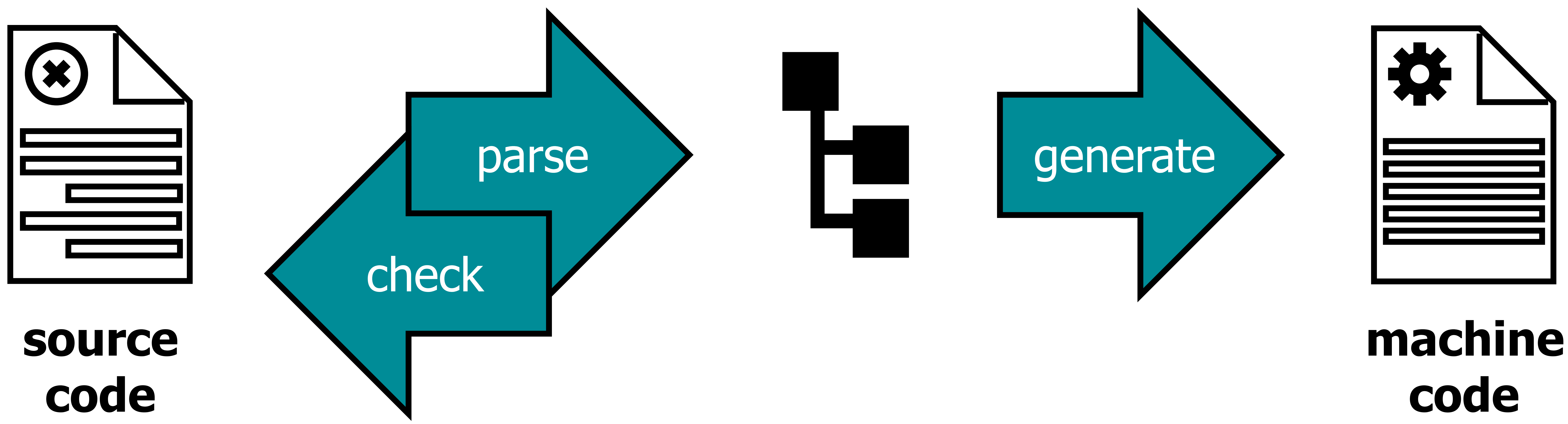
Authors' addresses: Casper Bach Poulsen, Delft University of Technology, The Netherlands, c.b.poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, The Netherlands, a.j.rouvoet@tudelft.nl; Andrew Tolmach, Portland State University, Oregon, USA, tolmach@pdx.edu; Robbert Krebbers, Delft University of Technology, The Netherlands, r.j.krebbers@tudelft.nl; Eelco Visser, Delft University of Technology, The Netherlands, e.visser@tudelft.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART16
https://doi.org/10.1145/3158104

Semantics

What is the meaning of a program?

`meaning(p) = behavior(p)`



`meaning(p)` = what happens when executing the generated (byte) code to which p is compiled

What is the meaning of a program?

```
meaning(p) = behavior(p)
```

What *is* behavior?

How can we *observe* behavior?

Mapping input to output

Changes to state of the system

Which behavior is essential, which accidental?

How can we define the semantics of a program?

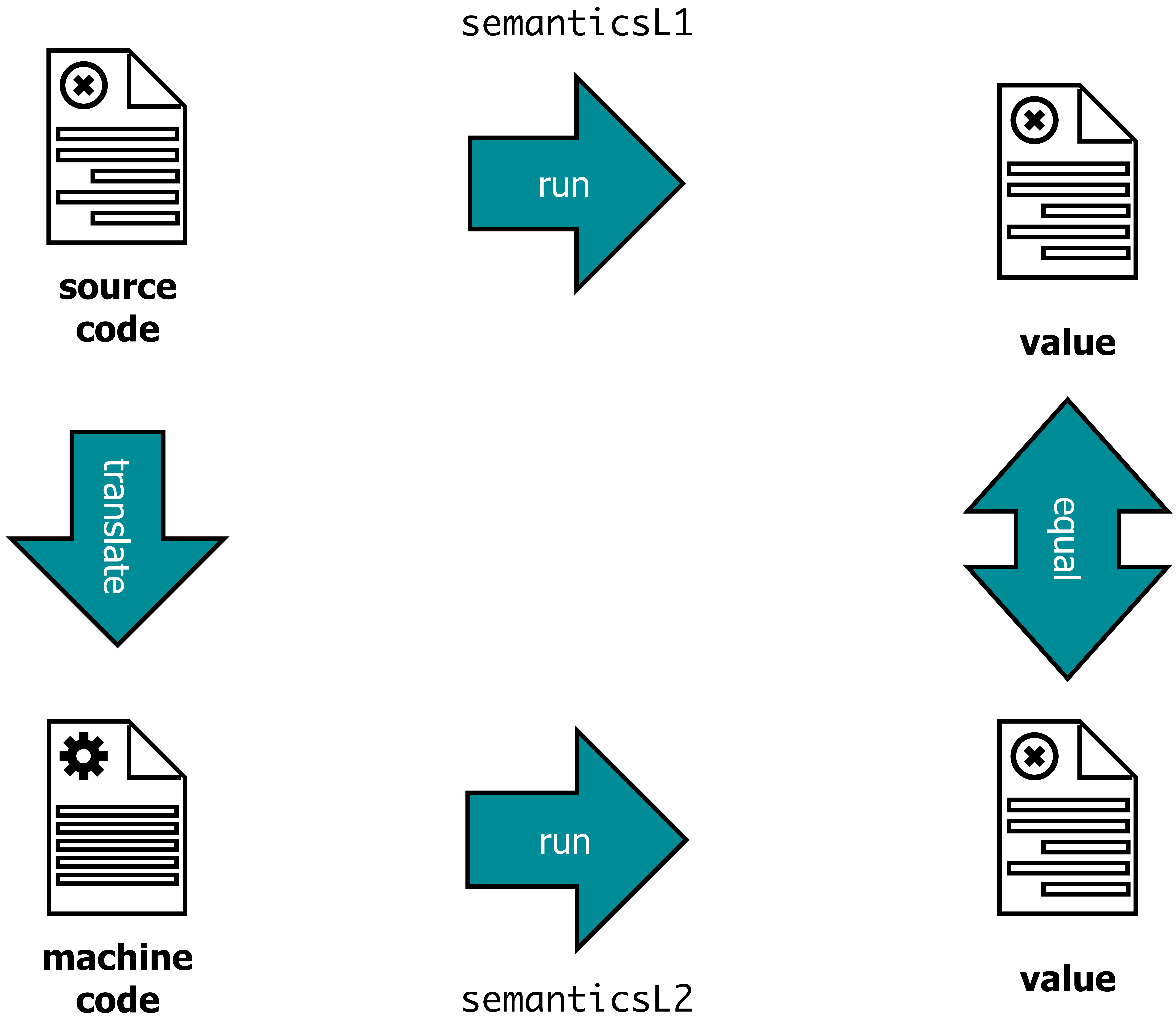
Compiler defines *translational* semantics

```
semanticsL1(p) = semanticsL2(translate(p))
```

Requires understanding `translate` and `semanticsL2`

How do we know that `translate` is correct?

Is there a more ***direct description*** of `semanticsL1`?


semanticsL1
run
source
code
value
translate
equal
machine
code
run
semanticsL2
value

Verifying Compiler Correctness

Direct semantics of source language provides a specification

How to check correctness?

Testing: for ***many*** programs p (and inputs i) ***test*** that

```
run(p)(i)  == run(translate(p))(i)
```

Verification: for ***all*** programs p (and inputs i) ***prove*** that

```
run(p)(i)  == run(translate(p))(i)
```

Validating Semantics

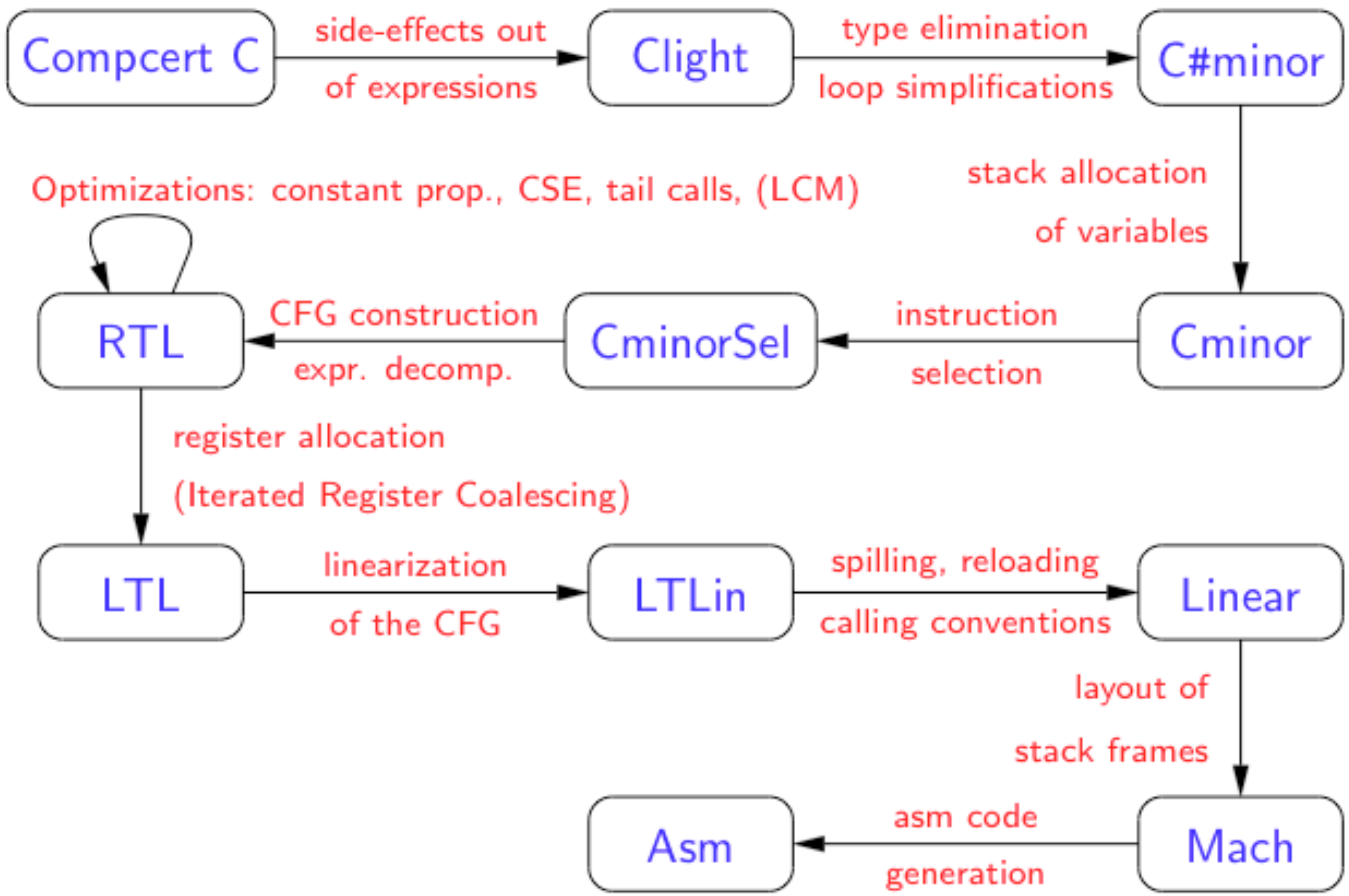
Is this the right semantics?

Testing: for ***many*** programs p (and inputs i) ***test*** that

```
run(p)(i)  ==  v
```

Requires specifying desired <p, i, v> combinations
(aka unit testing)

The CompCert C Compiler



http://compcert.inria.fr/compcert-C.html

Compiler Construction Courses of the Future

Language Specification

syntax definition
name binding
type system
dynamic semantics
translation
transformation

safety properties

Language Implementation

generating implementations
from specifications

parser generation
constraint resolution
partial evaluation
…

Language Testing

test generation

Language Verification

proving correctness

Operational Semantics

Operational Semantics

What is the **result** of execution of a program and **how** is that result achieved?

Structural Operational Semantics: What are the individual steps of an execution?

Natural Semantics: How is overall result of execution obtained?

Defined using a **transition system**

Transition System

rule

$$\frac{e_1 \rightarrow e_1' \quad \dots \quad e_n \rightarrow e_n'}{e \rightarrow e'}$$

premises

conclusion

axiom $e \rightarrow e'$

reduction $p \rightarrow v$

derivation tree to prove
v is value of program p

Structural Operational (Small-Step) Semantics

```
e = x | i | e + e | \ x.e | e e | ifz(e) e else e
```

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2}$$

$$\frac{e_2 \rightarrow e_2'}{e_1 + e_2 \rightarrow e_1 + e_2'}$$

$$i + j \rightarrow i \pm j$$

$$\frac{e_1 \rightarrow e_1'}{\mathtt{ifz}(e_1)\ e_2\ \mathtt{else}\ e_3 \rightarrow \mathtt{ifz}(e_1')\ e_2\ \mathtt{else}\ e_3}$$

$$\mathtt{ifz}(0)\ e_2\ \mathtt{else}\ e_3 \rightarrow e_2$$

$$\frac{i\ \mathtt{!=}\ 0}{\mathtt{ifz}(i)\ e_2\ \mathtt{else}\ e_3 \rightarrow e_3}$$

$$e \rightarrow e$$

reducing expressions

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v\ e_2 \rightarrow v\ e_2'}$$

$$(\backslash x.e)\ v_1 \rightarrow e[x := v_1]$$

order of evaluation?

Structural Operational (Small-Step) Semantics

```
e = x | i | e + e | \ x.e | e e | ifz(e) e else e
```

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$$

$$\frac{e_2 \to e_2'}{v_1 + e_2 \to v_1 + e_2'}$$

$$i + j \to i \pm j$$

$$\frac{e_1 \to e_1'}{\mathtt{ifz}(e_1)\ e_2\ \mathtt{else}\ e_3 \to \mathtt{ifz}(e_1')\ e_2\ \mathtt{else}\ e_3}$$

$$\mathtt{ifz}(0)\ e_2\ \mathtt{else}\ e_3 \to e_2$$

$$\frac{i\ \mathtt{!=}\ 0}{\mathtt{ifz}(i)\ e_2\ \mathtt{else}\ e_3 \to e_3}$$

$$e \to e$$

reducing expressions

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$$

$$\frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$$(\backslash x.e)\ v_1 \to e[x := v_1]$$

order of evaluation?

Natural (Big-Step) Semantics

```
e = x | i | e + e | \ x.e | e e | ifz(e) e else e
```

$$E \vdash e \Rightarrow v$$

reducing expressions to values

$$E \vdash i \Rightarrow \mathrm{NumV}(i)$$

$$\frac{E \vdash e_1 \Rightarrow \mathrm{NumV}(i) \quad E \vdash e_2 \Rightarrow \mathrm{NumV}(j)}{E \vdash e_1 + e_2 \Rightarrow \mathrm{NumV}(i + j)}$$

$$\frac{E \vdash e_1 \Rightarrow \mathrm{NumV}(0) \quad E \vdash e_2 \Rightarrow v}{E \vdash \mathrm{if}(e_1)\ e_2\ \mathrm{else}\ e_3 \Rightarrow v}$$

$$\frac{E \vdash e_1 \Rightarrow \mathrm{NumV}(i),\ i\ != 0 \quad E \vdash e_3 \Rightarrow v}{E \vdash \mathrm{if}(e_1)\ e_2\ \mathrm{else}\ e_3 \Rightarrow v}$$

$$\frac{E[x] = v}{E \vdash x \Rightarrow v}$$

$$E \vdash \backslash x.e \Rightarrow \mathrm{ClosV}(x, e, E)$$

$$\frac{E_1 \vdash e_1 \Rightarrow \mathrm{ClosV}(x, e, E_2) \quad E_1 \vdash e_2 \Rightarrow v_1 \quad \{x \mapsto v_1, E_2\} \vdash e \Rightarrow v_2}{E_1 \vdash e_1\ e_2 \Rightarrow v_2}$$

DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, Eelco Visser

RTA 2015

TU Delft

Interpreters for Spoofax Languages

gcdAB.pbox

```
let
  gcd = box(0)
in
  let f =
      fun (a, b) {
        if (b == 0)
          a
        else
          unbox(gcd)(b, a % b)
        end
      }
  in
    setbox(gcd, f);
    unbox(gcd)(1134903170, 1836)
  end
end
```

gcdAB.aterm

```
Let(
  [Bind("gcd", Box(Num("0")))]
, Let(
    [ Bind(
        "f"
      , Fun(
          ["a", "b"]
        , If(
            Eq(Var("b"), Num("0"))
          , Var("a")
          , App(
              Unbox(Var("gcd"))
            , [Var("b"), Mod(Var("a"), Var("b"))]
            )
          )
        )
      )
    ]
  , Seq(
      SetBox(Var("gcd"), Var("f"))
    , App(
        Unbox(Var("gcd"))
      , [Num("1134903170"), Num("1836")]
      )
    )
  )
)
```

gcdAB.evaluated.aterm

```
R_default_V(
  NumV(34)
, Map(
    "Store"
  , Bind(923, RefV(925))
  , Bind(
      925
    , ObjV(
        Map(
          "Env"
        , Bind("outer", 922)
        , Bind("self", 923)
        , Bind("super", 924)
        )
      )
    )
  , Bind(
      926
    , ClosV(
        Fun(
          ["a", "b"]
        , If(
            Eq(Var("b"), Num("0"))
          , Var("a")
          , App(
              Unbox(Var("gcd"))
            , [Var("b"), Mod(Var("a"), V
            )
```

Design Goals

Executable	Modular
Portable	Concise
High-performance	Unsurprising
	Statically Typed

⇩ Big-Step

⇩ I-MSOS

M. Churchill, P. D. Mosses, and P. Torrini. Reusable components of semantic specifications. In MODULARITY, April 2014.

Example: DynSem Semantics of PAPL-Box

```
let
  fac = box(0)
in
  let f = fun (n) {
      if (n == 0)
        1
      else
        n * (unbox(fac) (n - 1))
      end
    }
  in
    setbox(fac, f);
    unbox(fac)(10)
   end
end
```

Features

- Arithmetic
- Booleans
- Comparisons
- Mutable variables
- Functions
- Boxes

Components

- Syntax in SDF3
- Dynamic Semantics in DynSem

Abstract Syntax from Concrete Syntax

```
module Arithmetic

imports Expressions
imports Common

context-free syntax

  Expr.Num    = INT

  Expr.Plus  = [[Expr] + [Expr]] {left}
  Expr.Minus = [[Expr] - [Expr]] {left}
  Expr.Times = [[Expr] * [Expr]] {left}
  Expr.Mod   = [[Expr] % [Expr]] {left}

context-free priorities

    {left: Expr.Times Expr.Mod }
  > {left: Expr.Minus Expr.Plus }
```

src-gen/ds-signatures/Arithmetic-sig

```
module Arithmetic-sig

imports Expressions-sig
imports Common-sig

signature
    sorts
      Expr
    constructors
      Num : INT -> Expr
      Plus : Expr * Expr -> Expr
      Minus : Expr * Expr -> Expr
      Times : Expr * Expr -> Expr
      Mod : Expr * Expr -> Expr
```

Values, Meta-Variables, and Arrows

```
module values

signature
  sorts V Unit
  constructors
    U : Unit
  variables
    v: V
```

```
module expressions

imports values
imports Expressions-sig

signature
  arrows
    Expr --> V
  variables
    e : Expr
    x : String
```

Term Reduction Rules

```
module arithmetic-explicit

imports expressions primitives Arithmetic-sig

signature
  constructors
    NumV: Int -> V

rules

  Num(__String2INT__(n)) --> NumV(str2int(n)).

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1); e2 --> NumV(i2).

  Minus(e1, e2) --> NumV(minusI(i1, i2))
  where
    e1 --> NumV(i1); e2 --> NumV(i2).
```

```
module primitives
signature
  native operators
    str2int : String -> Int
    plusI   : Int * Int -> Int
    minusI  : Int * Int -> Int
```

Native Operations

```
module primitives
signature
  native operators
    str2int : String -> Int
    plusI   : Int * Int -> Int
    minusI  : Int * Int -> Int
```

```
public class Natives {

   public static int plusI_2(int i1, int i2) {
      return i1 + i2;
   }

   public static int str2int_1(String s) {
      return Integer.parseInt(s);
   }
}
```

Arrows as Coercions

```
rules

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).
```

```
signature
  constructors
    Plus : Expr * Expr -> Expr
    NumV : Int -> V
  arrows
    Expr --> V
```

```
rules

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).
```

Modular

```
module arithmetic

imports Arithmetic-sig
imports expressions
imports primitives

signature
  constructors
    NumV: Int -> V

rules

  Num(str) --> NumV(str2int(str)).

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).

  Minus(NumV(i1), NumV(i2)) --> NumV(minusI(i1, i2)).

  Times(NumV(i1), NumV(i2)) --> NumV(timesI(i1, i2)).

  Mod(NumV(i1), NumV(i2)) --> NumV(modI(i1, i2)).
```

```
module boolean

imports Booleans-sig expressions

signature
  constructors
    BoolV : Bool -> V

rules

  True() --> BoolV(true).
  False() --> BoolV(false).

  Not(BoolV(false)) --> BoolV(true).
  Not(BoolV(true)) --> BoolV(false).

  Or(BoolV(true), _) --> BoolV(true).
  Or(BoolV(false), e) --> e.

  And(BoolV(false), _) --> BoolV(false).
  And(BoolV(true), e) --> e.
```

```
module comparison

imports Comparisons-sig arithmetic boolean

rules

  Gt(NumV(i1), NumV(i2)) --> BoolV(gtI(i1, i2)).

  Eq(NumV(i1), NumV(i2)) --> BoolV(eqI(i1, i2)).

  Eq(BoolV(b1), BoolV(b2)) --> BoolV(eqB(b1, b2)).
```

Control-Flow

```
module controlflow

imports ControlFlow-sig
imports expressions
imports boolean

rules

  Seq(v, e2) --> e2.

  If(BoolV(true), e1, _) --> e1.

  If(BoolV(false), _, e2) --> e2.
```

```
module controlflow

imports ControlFlow-sig
imports expressions
imports boolean

rules

  Seq(e1, e2) --> v2
  where
    e1 --> v1;
    e2 --> v2.

  If(e1, e2, e3) --> v
  where
    e1 --> BoolV(true);
    e2 --> v.

  If(e1, e2, e3) --> v
  where
    e1 --> BoolV(false);
    e3 --> v.
```

Immutable Variables: Environment Passing

```
constructors
  Let : ID * Expr * Expr -> Expr
  Var : ID -> Expr
```

```
module variables

imports Variables-sig environment

rules

  E |- Let(x, v: V, e2) --> v2
  where
    Env {x |--> v, E} |- e2 --> v2.

  E |- Var(x) --> E[x].
```

```
module environment

imports values

signature
  sort aliases
    Env = Map<String, V>
  variables
    E : Env
```

First-Class Functions: Environment in Closure

```
constructors
  Fun : ID * Expr -> Expr
  App : Expr * Expr -> Expr
```

```
module unary-functions

imports expressions environment

signature
  constructors
    ClosV : String * Expr * Env -> V
rules

  E |- Fun(x, e) --> ClosV(x, e, E).

  E |- App(e1, e2) --> v
  where
    E |- e1 --> ClosV(x, e, E');
    E |- e2 --> v2;
    Env {x |--> v2, E'} |- e --> v.
```

```
module environment

imports values

signature
  sort aliases
    Env = Map<String, V>
  variables
    E : Env
```

Implicit Propagation

```
rules

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).
```

```
rules

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).
```

```
rules

  E |- Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    E |- e1 --> NumV(i1);
    E |- e2 --> NumV(i2).
```

Mutable Boxes: Store

```
module store

imports values

signature
  sort aliases
   Store = Map<Int, V>
  variables
    S : Store
```

```
module box
imports store arithmetic
signature
  constructors
    Box    : Expr -> Expr
    Unbox  : Expr -> Expr
    SetBox : Expr * Expr -> Expr
  constructors
    BoxV: Int -> V
rules

  Box(e) :: S --> BoxV(loc) :: Store {loc |--> v, S'}
  where e :: S --> v :: S';
        fresh => loc.

  Unbox(BoxV(loc)) :: S --> S[loc].

  SetBox(BoxV(loc), v) :: S --> v :: Store {loc |--> v, S}.
```

Mutable Variables: Environment + Store

```
constructors
  Let : ID * Expr * Expr -> Expr
  Var : ID -> Expr
  Set : String * Expr -> Expr
```

```
module variables-mutable
imports Variables-sig store
rules

  E |- Var(x) :: S --> v :: S
  where E[x] => loc; S[loc] => v.

  E |- Let(x, v, e2) :: S1 --> v2 :: S3
  where
    fresh => loc;
    {loc |--> v, S1} => S2;
    Env {x |--> loc, E} |- e2 :: S2 --> v2 :: S3.

  E |- Set(x, v) :: S --> v :: Store {loc |--> v, S}
  where E[x] => loc.
```

```
module store

imports values

signature
  sort aliases
    Env = Map<ID, Int>
    Store = Map<Int, V>
  variables
    E : Env
    S : Store
```

Implicit Store Threading

```
rules

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).
```

```
rules

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).
```

```
rules

  E |- Plus(e1, e2) :: S1 --> NumV(plusI(i1, i2)) :: S3
  where
    E |- e1 :: S1 --> NumV(i1) :: S2;
    E |- e2 :: S2 --> NumV(i2) :: S3.
```

Abstraction: Env/Store Meta Functions

```
module store

imports values

signature
  sort aliases
    Env = Map<String, Int>
    Store = Map<Int, V>
  variables
    E : Env
    S : Store
  arrows
    readVar  : String --> V
    bindVar  : String * V --> Env
    writeVar : String * V --> V

    allocate : V --> Int
    write    : Int * V --> V
    read     : Int --> V
```

```
rules

  allocate(v) --> loc
  where
    fresh => loc;
    write(loc, v) --> _.

  write(loc, v) :: S -->
    v :: Store {loc |--> v, S}.

  read(loc) :: S --> S[loc] :: S.

rules

  bindVar(x, v) --> {x |--> loc}
  where allocate(v) --> loc.

  E |- readVar(x) --> read(E[x]).

  E |- writeVar(x, v) --> write(E[x], v).
```

Boxes with Env/Store Meta Functions

```
module boxes

signature
  constructors
    Box    : Expr -> Expr
    Unbox  : Expr -> Expr
    SetBox : Expr * Expr -> Expr
  constructors
    BoxV: V -> V

rules

  Box(v) --> BoxV(NumV(allocate(v))).

  Unbox(BoxV(NumV(loc))) --> read(loc).

  SetBox(BoxV(NumV(loc)), v) --> write(loc, v).
```

Mutable Variables with Env/Store Meta Functions

```
constructors
  Let : String * Expr * Expr -> Expr
  Var : String -> Expr
  Set : String * Expr -> Expr
```

```
module variables

imports expressions store

rules

  Var(x) --> readVar(x).

  E |- Let(x, v1, e) --> v2
  where
    bindVar(x, v1) --> E';
    Env {E', E} |- e --> v2.

  Set(x, v) --> v
  where
    writeVar(x, v) --> _.
```

Functions with Multiple Arguments

```
module functions

imports Functions-sig
imports variables

signature
  constructors
    ClosV    : List(ID) * Expr * Env -> V
    bindArgs : List(ID) * List(Expr) --> Env

rules

  E |- Fun(xs, e) --> ClosV(xs, e, E).

  App(ClosV(xs, e_body, E_clos), es) --> v'
  where
    bindArgs(xs, es) --> E_params;
    Env {E_params, E_clos} |- e_body --> v'.

  bindArgs([], []) --> {}.

  bindArgs([x | xs], [e | es]) --> {E, E'}
  where
    bindVar(x, e) --> E;
    bindArgs(xs, es) --> E'.
```

Tiger in DynSem

store.ds

```
imports dynamics/values

signature // lvalue
  sorts LValue
  arrows
    LValue -lval-> Int
  variables
    lv : LValue

signature // environment
  sorts Id
  sort aliases
    // Address = Int
    Env = Map(Id, Int)
  variables
    a : Int
  components
    E : Env
  arrows
    lookup(Id) --> Int
    bind(Id, Int) --> Env


rules

  E |- lookup(x) --> E[x].

  E |- bind(x, a) --> {x |--> a, E}.

signature // heap
  sort aliases
    Heap = Map(Int, V)
  components
    H : Heap
  arrows
    read(Int) --> V
    allocate(V) --> Int
    write(Int, V) --> V

rules

  read(a) :: H --> H[a].

  write(a, v) :: H --> v :: H {a |--> v, H

  allocate(v) --> a
  where
    fresh => a;
    write(a, v) --> _.
```

numbers.ds | functions.ds | equality.ds

```
module functions

imports ds-signatures/Functions-sig
imports dynamics/base
imports dynamics/store
imports dynamics/bindings

signature
  constructors
    ClosureV : List(FArg) * Exp * Env -> V
  arrows
    E |- funEnv(List(FunDec)) :: H --> Env :: H
    E |- evalFuns(List(FunDec)) :: H --> Env :: H
    E |- evalArgs(List(FArg), List(Exp)) :: H --> Env ::

rules // function definition

  FunDecs(fds) --> E
  where
    funEnv(fds) --> E;
    E |- evalFuns(fds) --> _.

  E |- funEnv([]) --> E.

  funEnv([FunDec(f, _, _, _) | fds]) --> E
  where
    E bindVar(f, UndefV()) |- funEnv(fds) --> E.

  E |- evalFuns([]) --> E.

  E |- evalFuns([FunDec(f, args, _, e) | fds]) --> evalFu
  where
    writeVar(f, ClosureV(args, e, E)) --> _.

rules // function call

  Call(f, es) --> v
  where
    readVar(f) --> ClosureV(args, e, E);
    evalArgs(args, es) --> E';
    E {E', E} |- e --> v.

  evalArgs([], []) --> {}.

  evalArgs([FArg(x, _) | args], [v | es]) --> {x |--> a,
  where
    allocate(v) --> a;
    evalArgs(args, es) --> E.

rules // procedure definition
```

control-flow.ds | arrays.ds

```
signature
  sort aliases
    Idx = Map(Int, Int)
  variables
    I : Idx
  constructors
    ArrayV : Idx -> V
  arrows
    initArray(Int, Int, V, Idx) --> Idx

rules

  Array(_, IntV(i), v) --> ArrayV(I)
  where
    initArray(0, i, v, {}) --> I.

  initArray(i, j, v, I) --> I'
  where
    case ltI(i, j) of {
      1 =>
        allocate(v) --> a;
        initArray(addI(i, 1), j, v, {i |--> a, I})
      0 =>
        I => I'
    }.

  Subscript(a, IntV(i)) -lval-> I[i]
  where
    read(a) --> ArrayV(I).
```

records.ds

```
signature
  constructors
    NilV : V
    RecordV : Env -> V
  arrows
    initFields(List(InitField)) --> Env

rules // records

  NilExp() --> NilV().

  Record(_, fields) --> RecordV(E)
  where initFields(fields) --> E.

  initFields([]) --> {}.

  initFields([InitField(f, v) | fields]) --> {f |-
  where
    allocate(v) --> a; initFields(fields) --> E.
```

prettyprint.tig

```
let

type tree = {key: string, left : tree, right: tree

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);
```

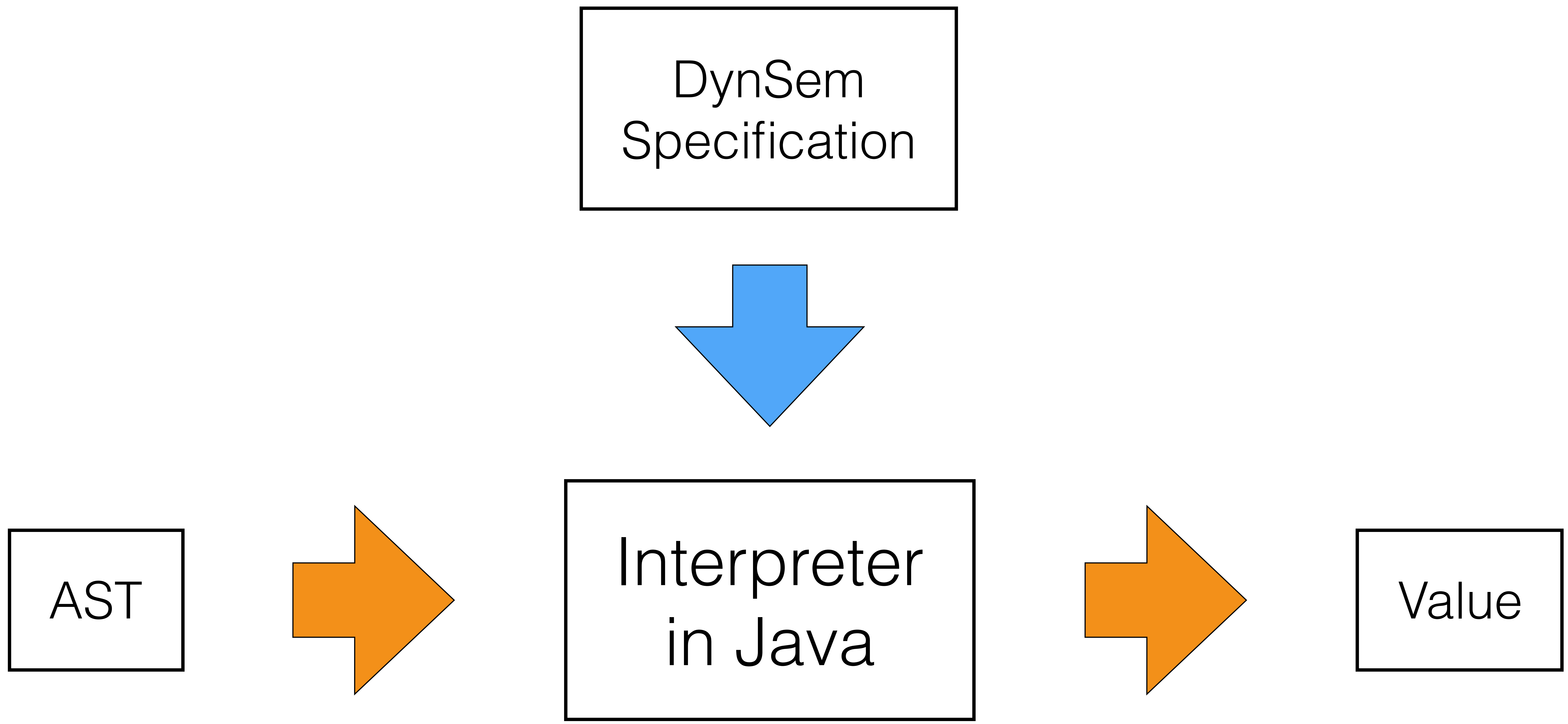
prettyprint.aterm

```
Mod(
  Let(
    [ TypeDecs(
        [ TypeDec(
            "tree"
          , RecordTy(
              [ Field("key", Tid("string"))
              , Field("left", Tid("tree"))
              , Field("right", Tid("tree"))
              ]
            )
          )
        ]
      )
    , FunDecs(
        [ FunDec(
            "prettyprint"
          , [FArg("tree", Tid("tree"))]
          , Tid("string")
          , Let(
              [ VarDecNoType("output", String("\"
              , FunDecs(
                  [ ProcDec(
                      "write"
                    , [FArg("s", Tid("string"))]
```

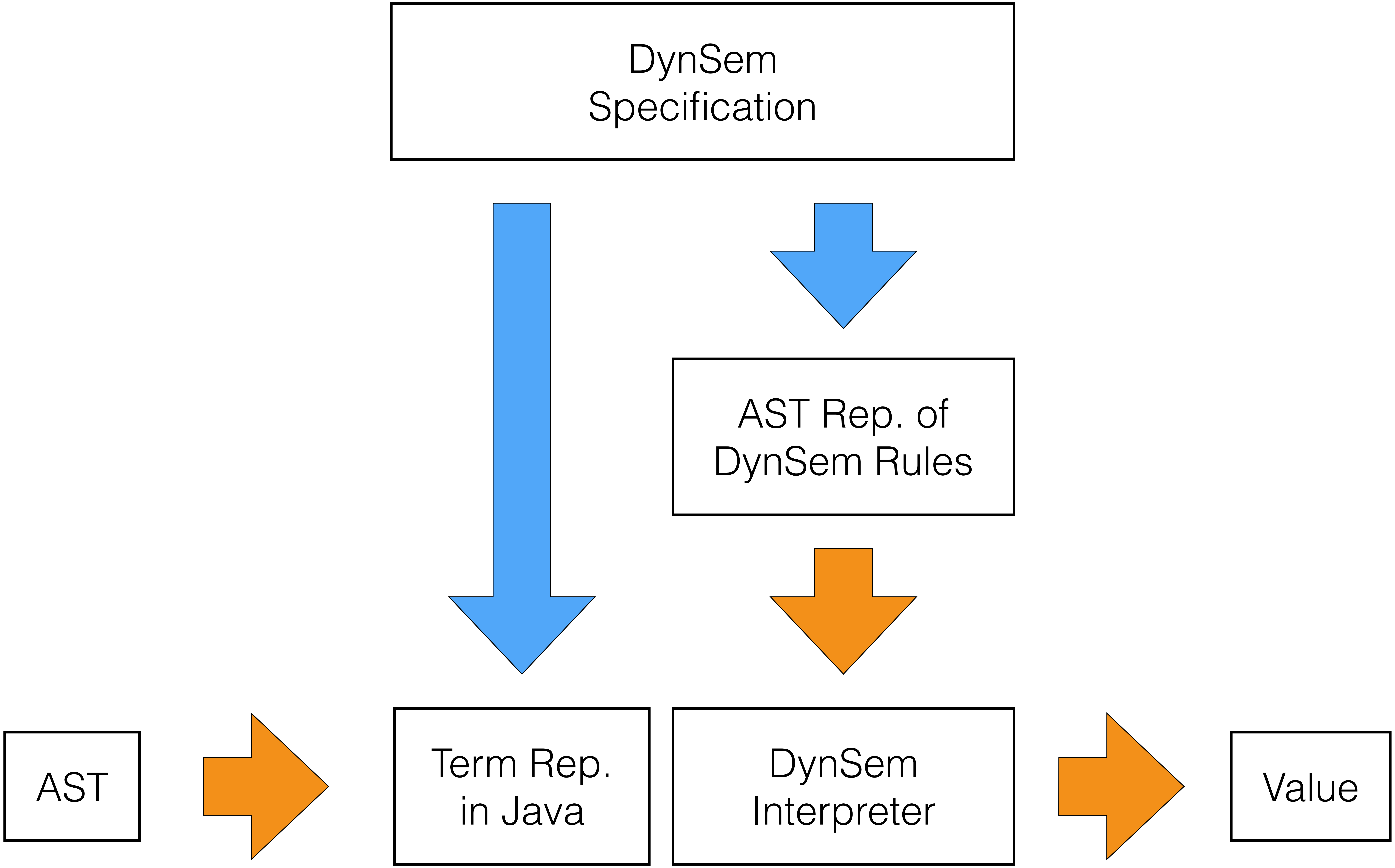
https://github.com/MetaBorgCube/metaborg-tiger

Interpreter Generation

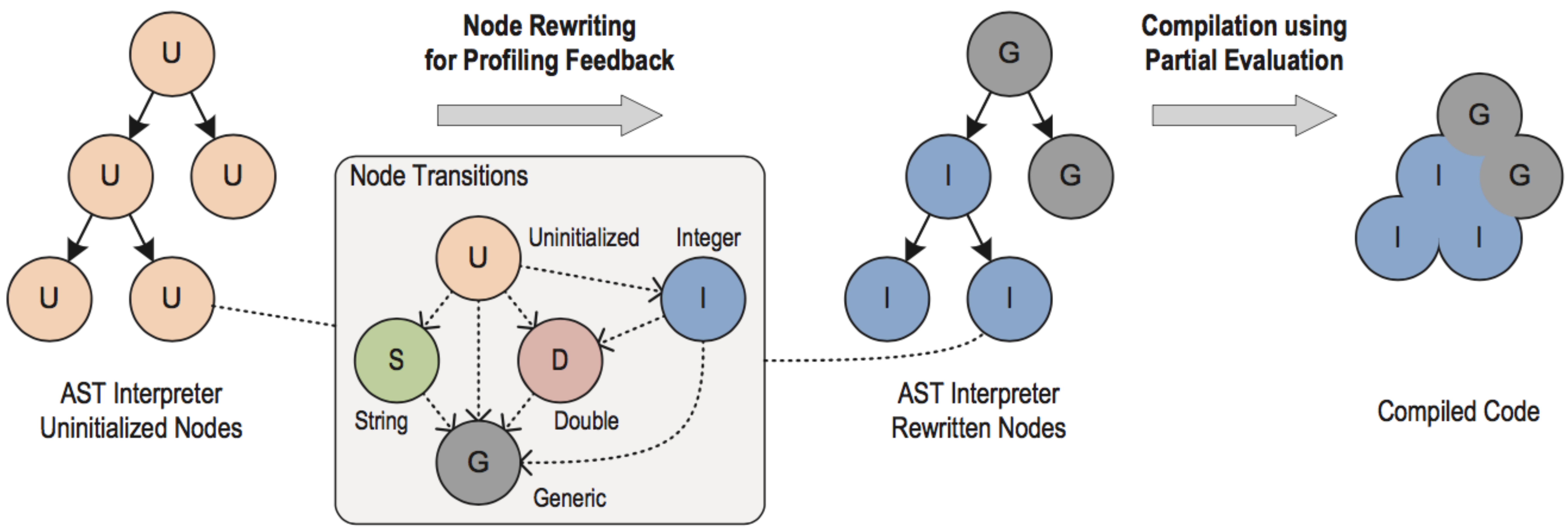
Generating AST Interpreter from Dynamic Semantics



DynSem Meta-Interpreter



Truffle: Partial Evaluation of AST Interpreters



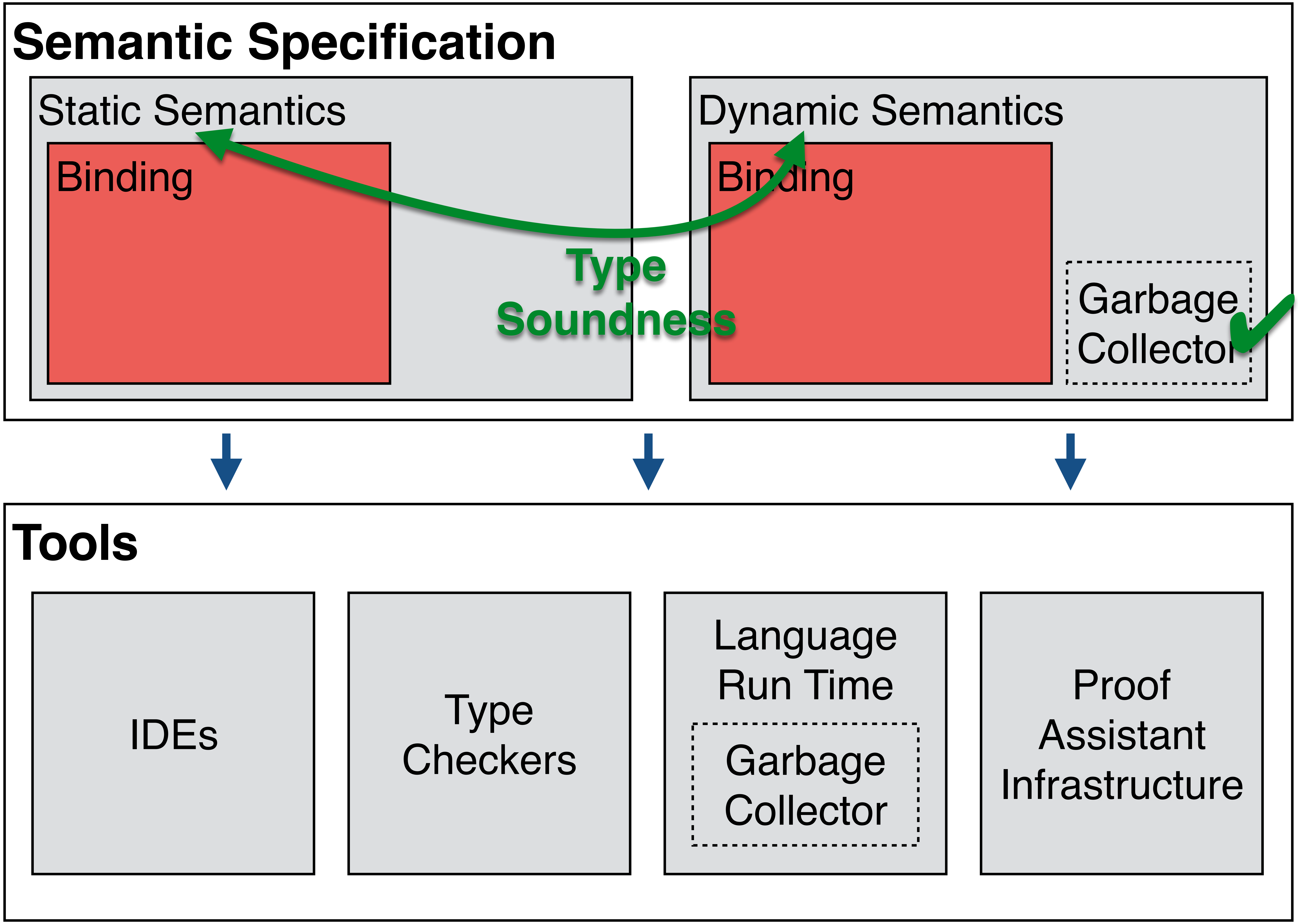
Würthinger et al. One VM to rule them all. Onward! 2013

Scopes Describe Frames: A uniform model for memory layout in dynamic semantics

Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, Eelco Visser

ECOOP 2016

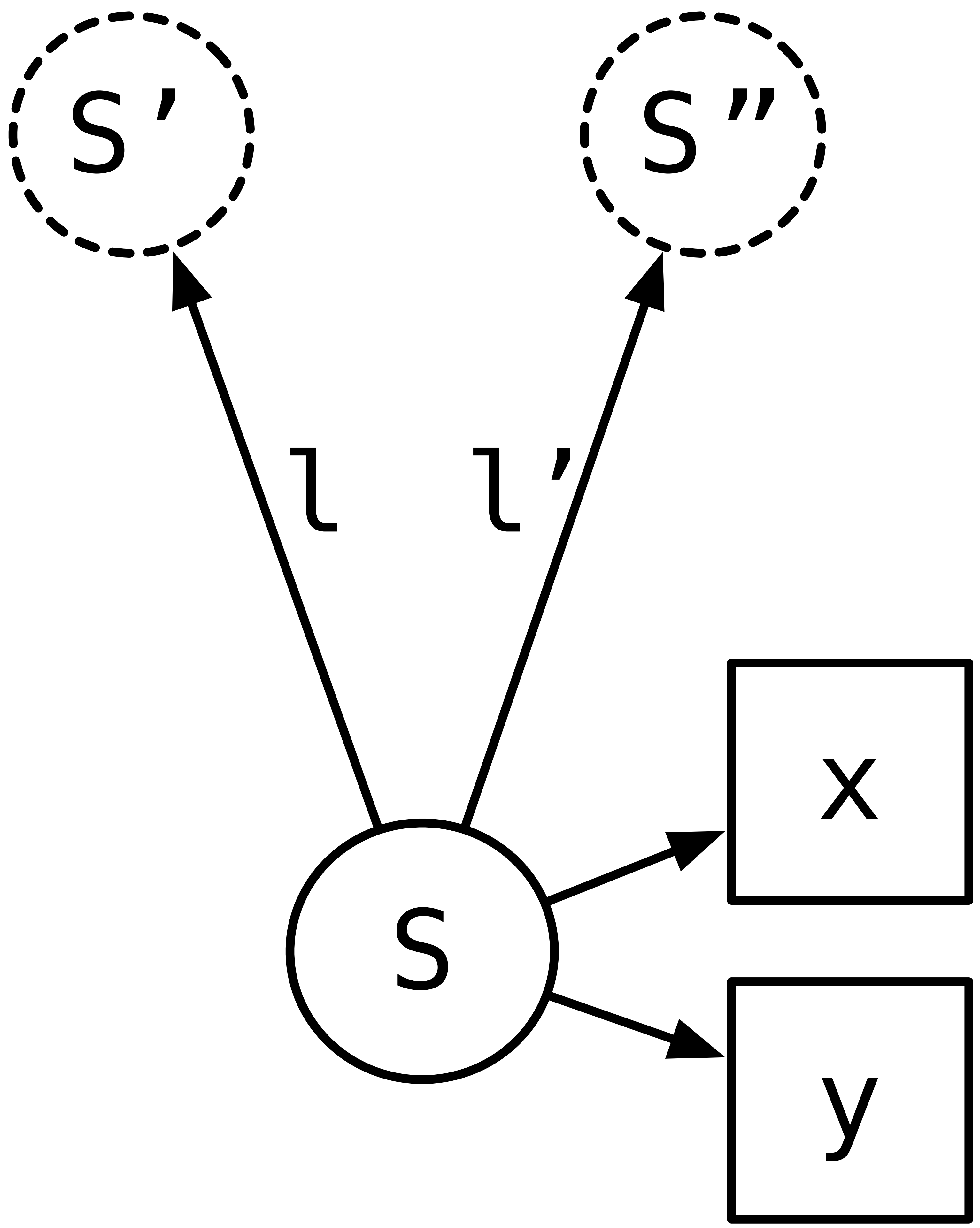
TU Delft


Semantic Specification
Static Semantics
Binding
Dynamic Semantics
Binding
Type
Soundness
Garbage
Collector
Tools
IDEs
Type
Checkers
Language
Run Time
Garbage
Collector
Proof
Assistant
Infrastructure

		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`	Typing Contexts Type Substitution	Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`	Typing Contexts Store Typing	Stores/Heaps

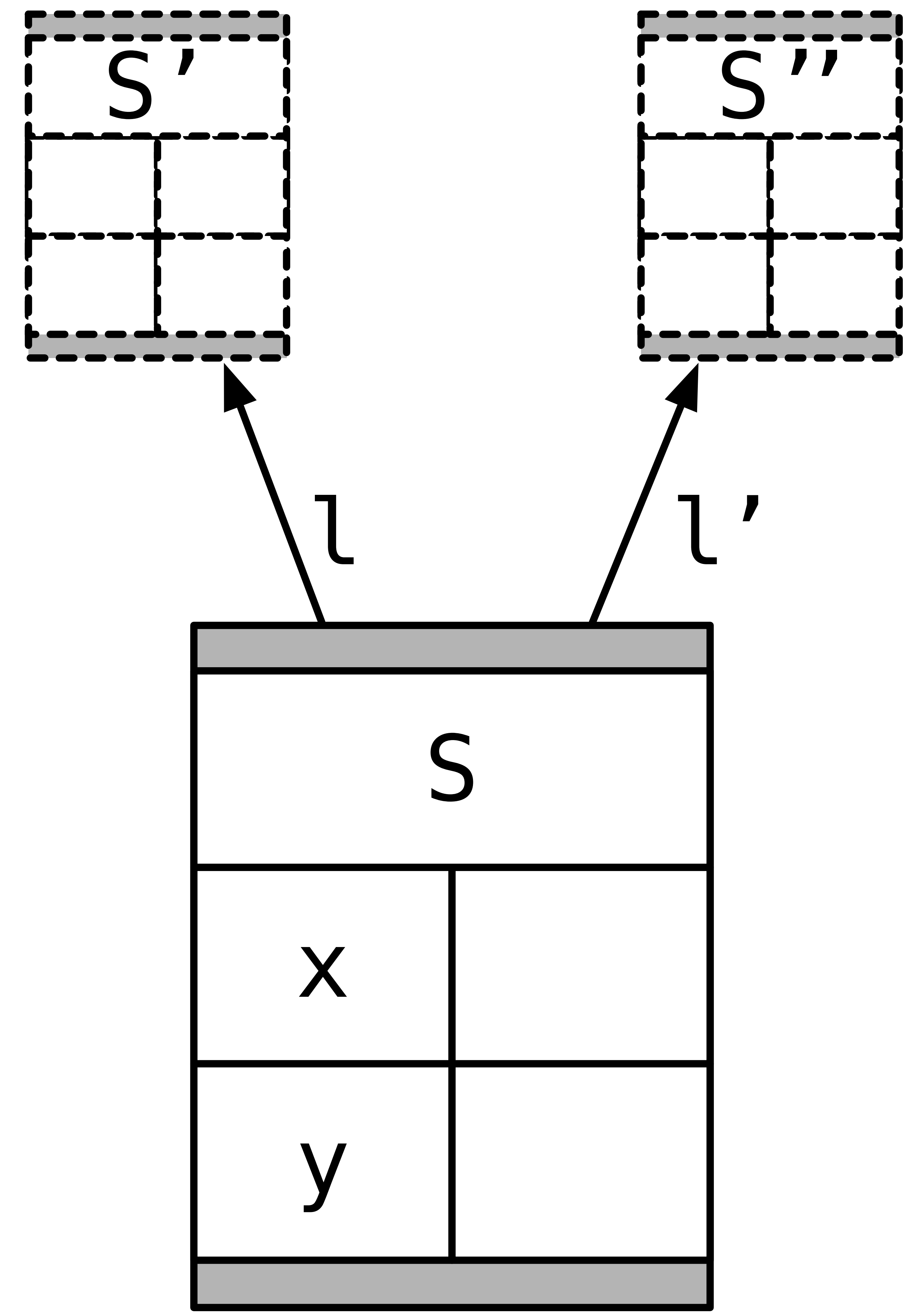
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
		Class Tables	Mutable Objects
			Stores/Heaps

Scope



[ESOP'15]

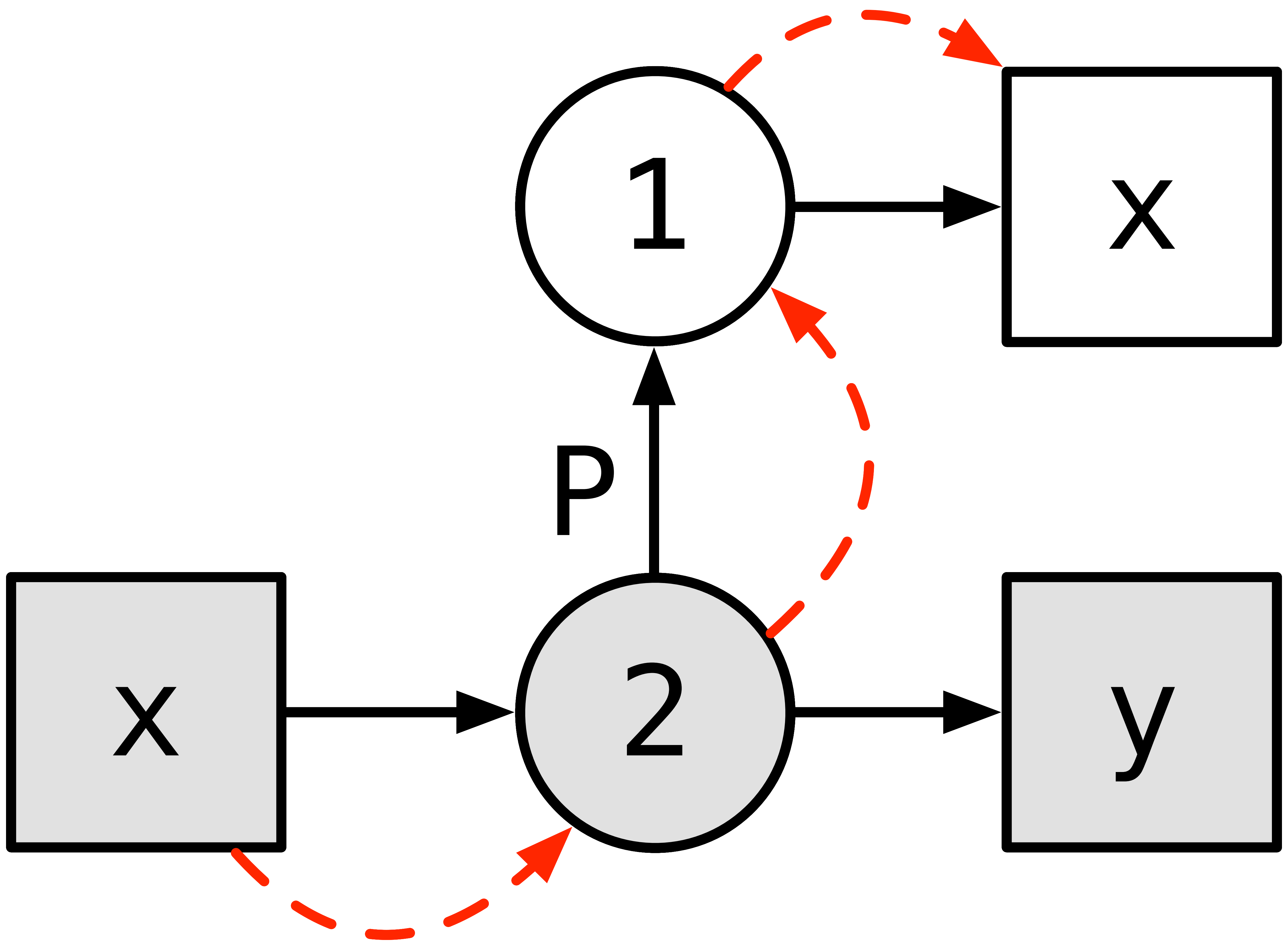
Frame



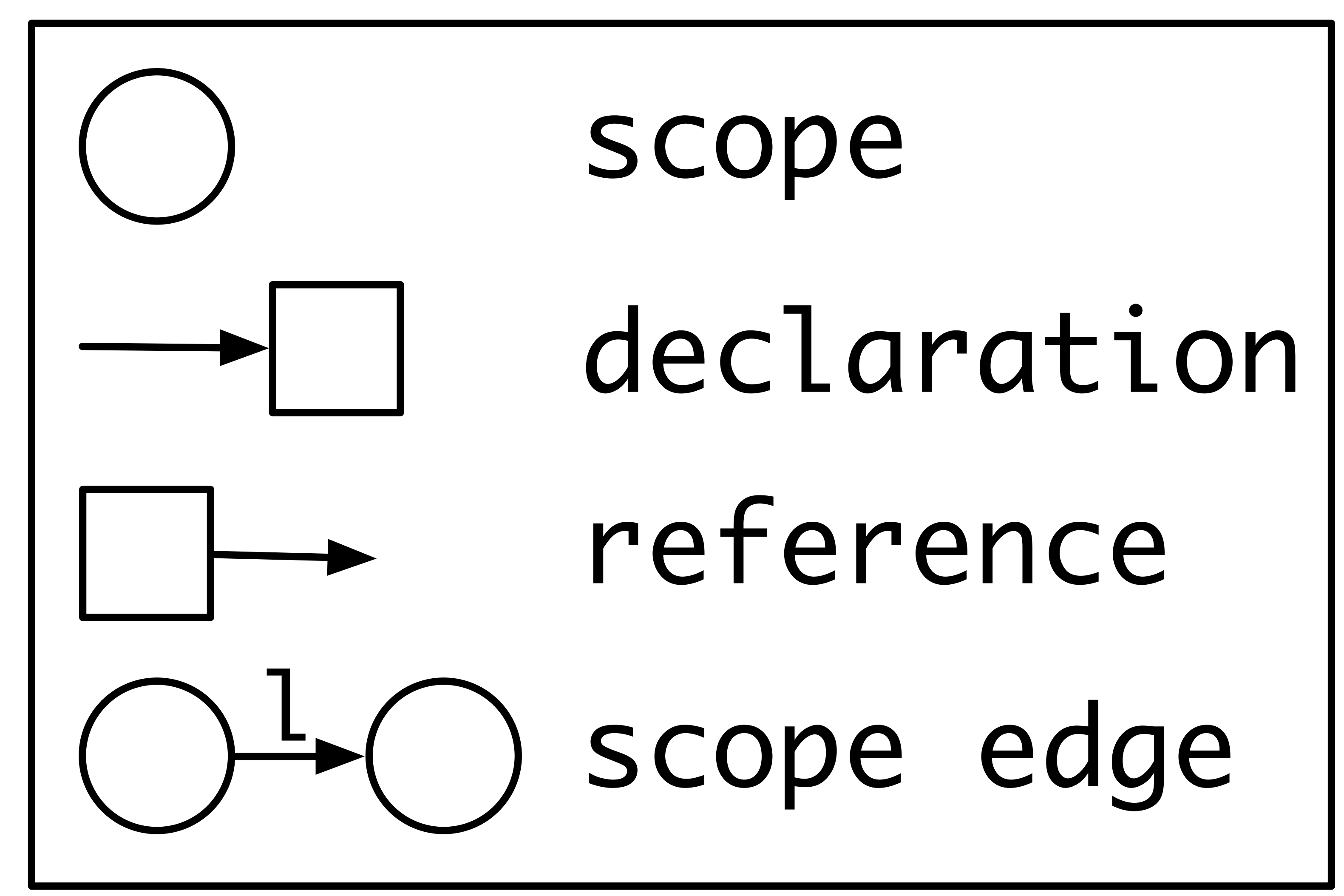
[ECOOP'16]

Lexical Scoping

```
val x = 31;
val y = x + 11;
```

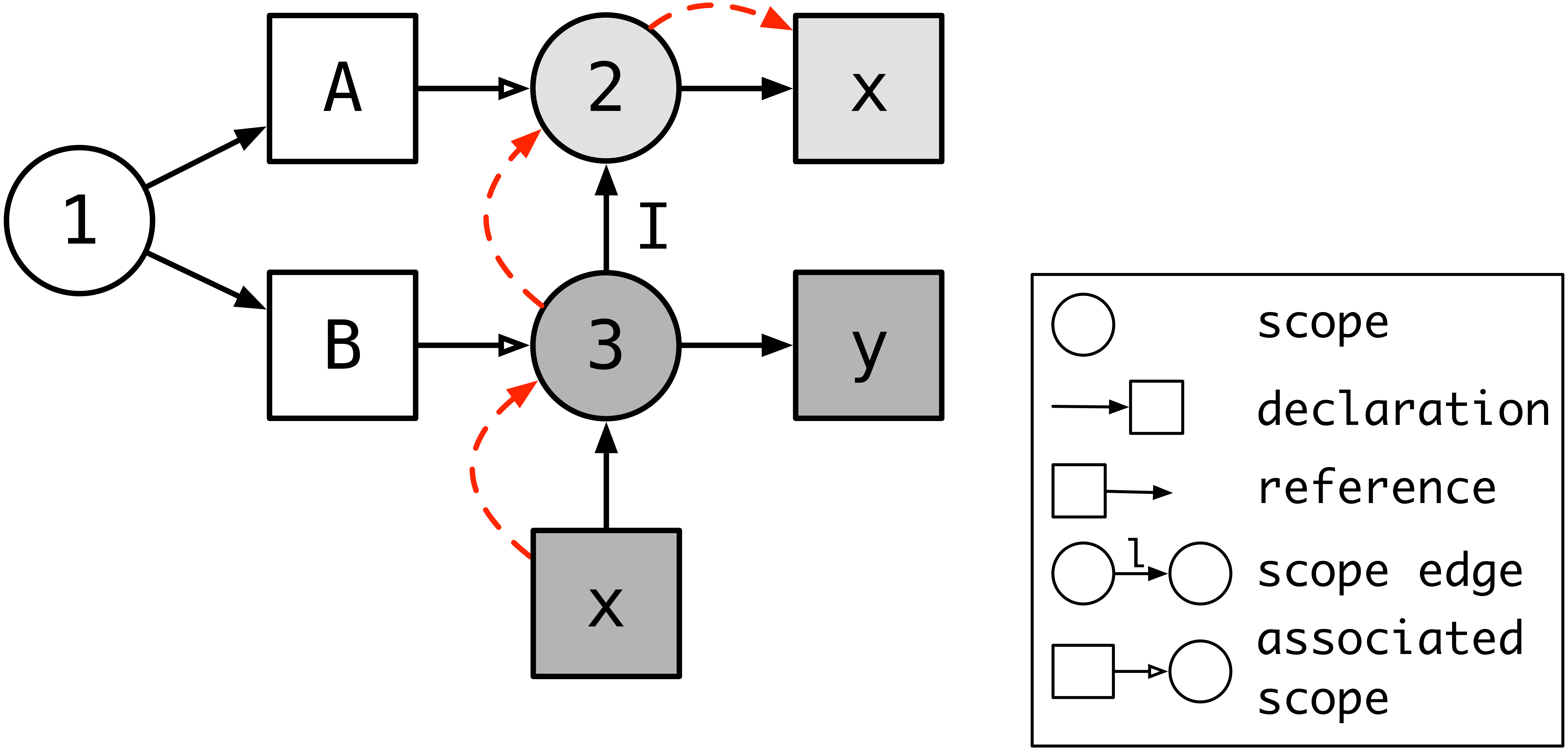


[ESOP'15; PEPM'16]

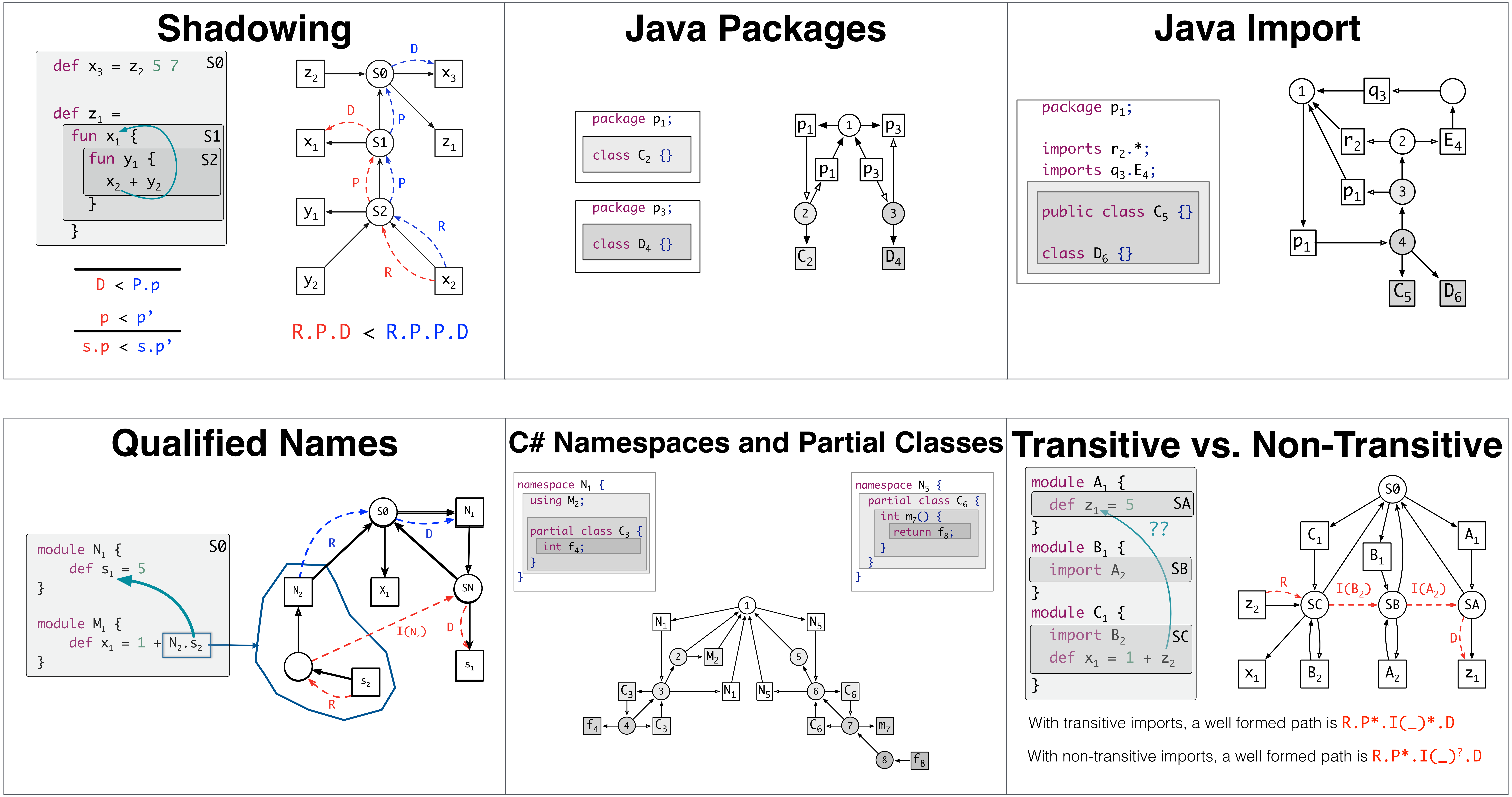


Inheritance

```
class A { var x = 42; }
class B extends A { var y = x; }
```



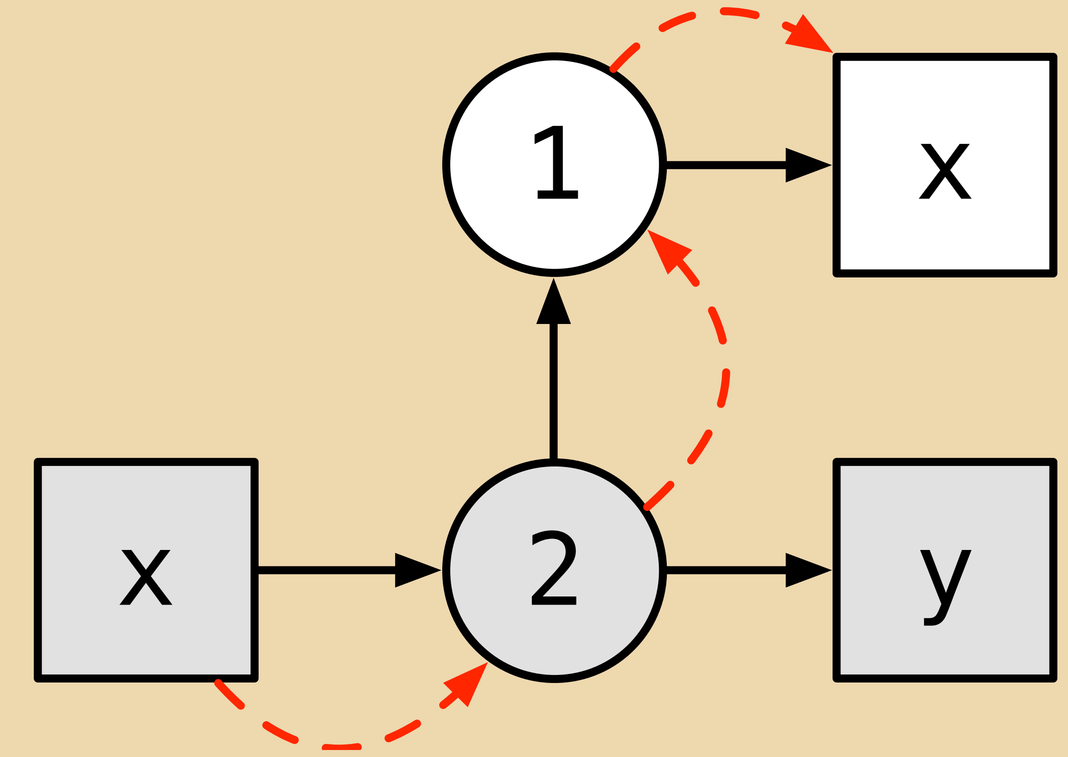
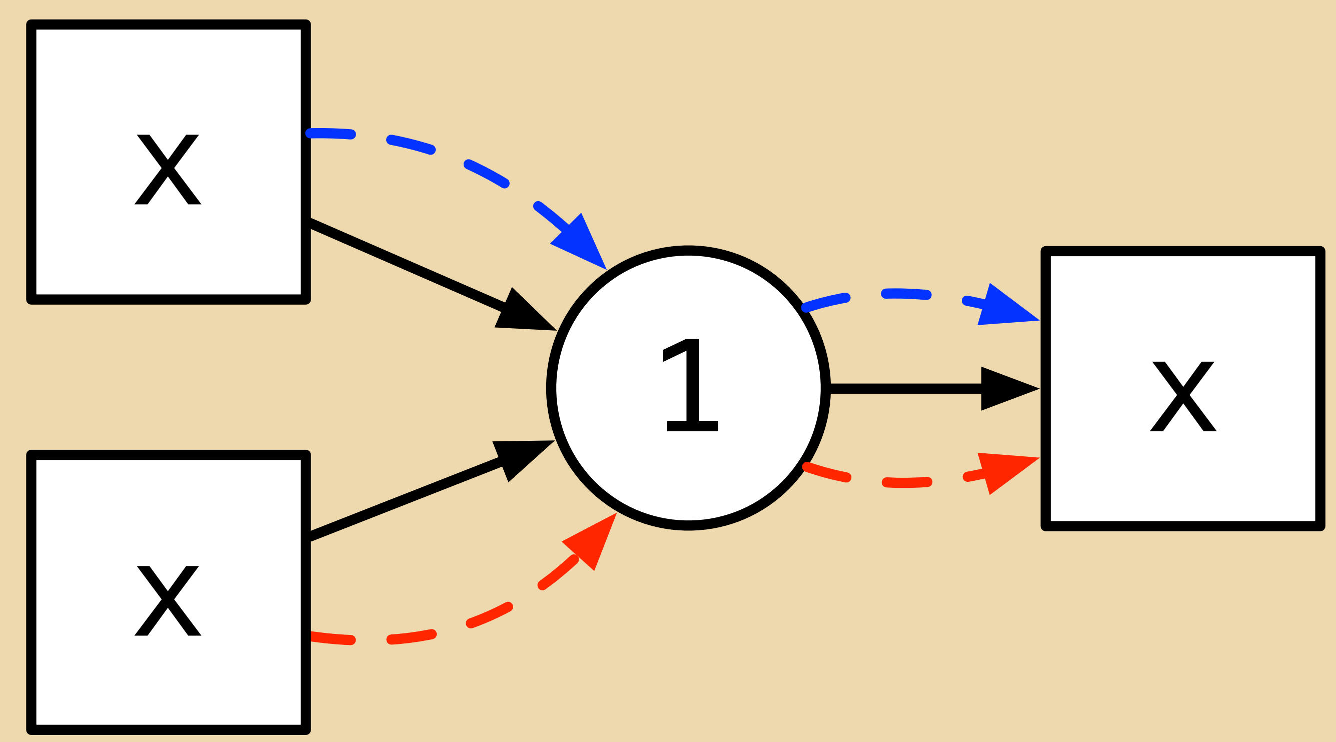
More Binding Patterns

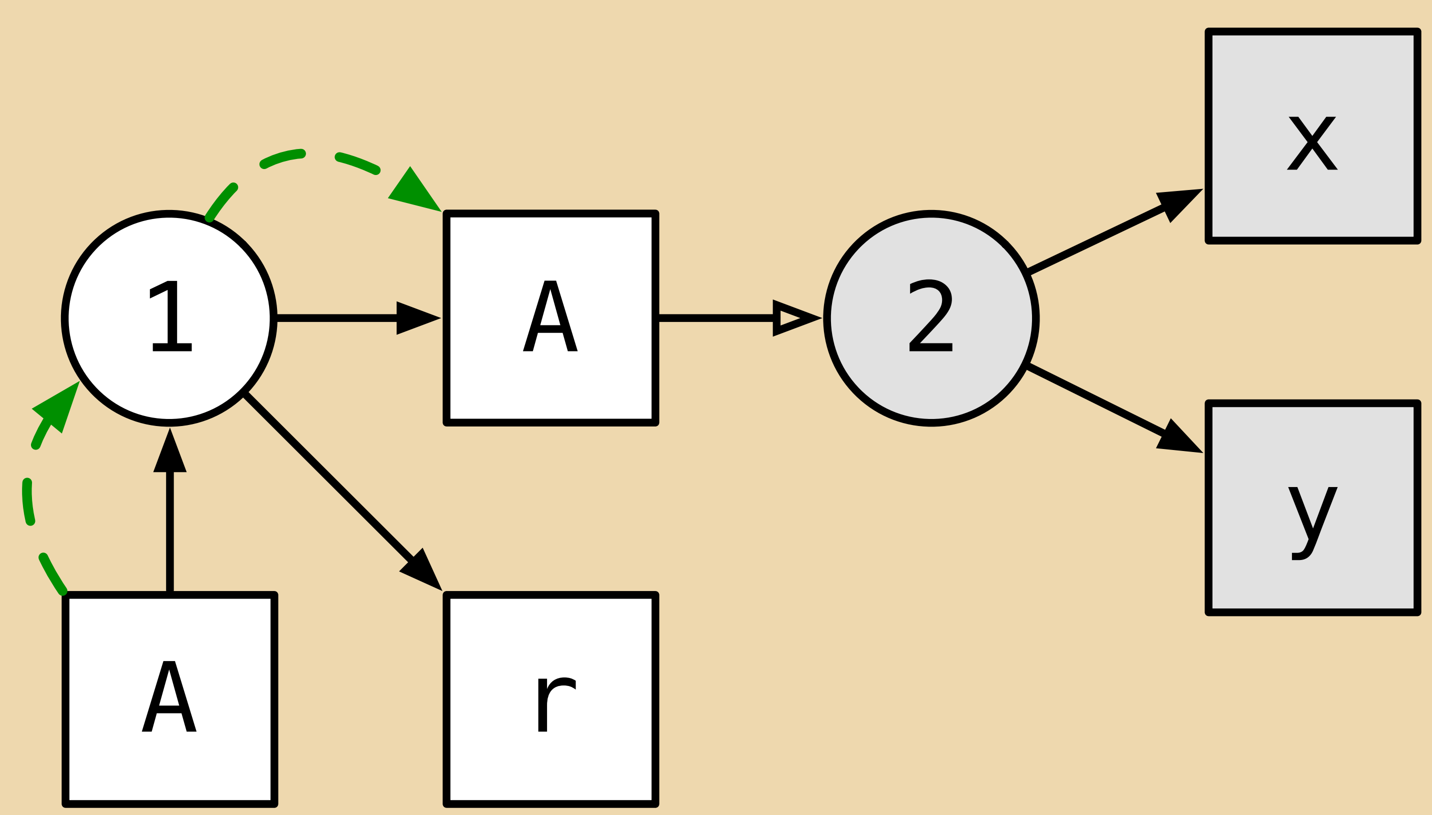


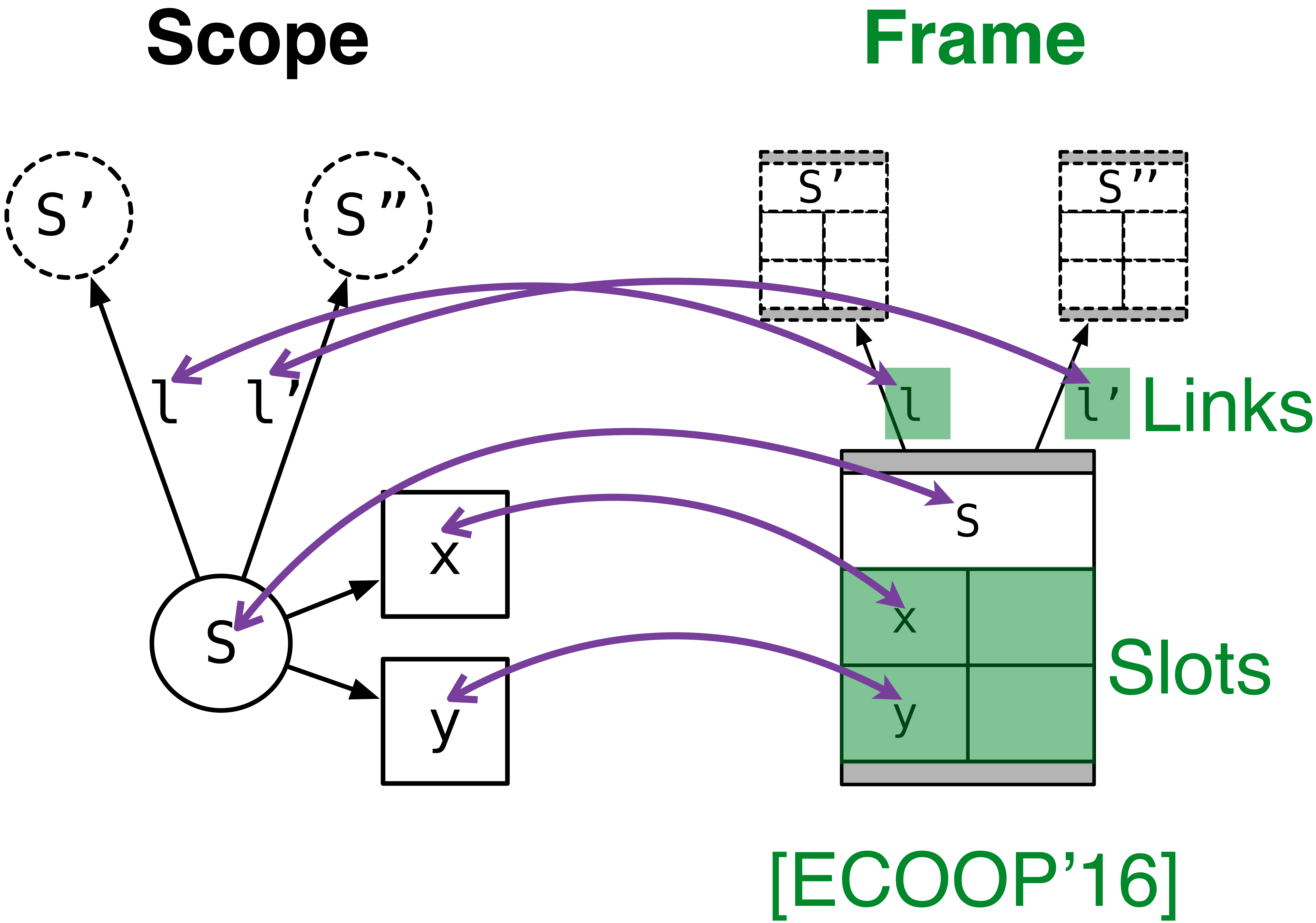
[ESOP'15]

		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`	Typing Contexts Type Substitution	Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`	Typing Contexts Store Typing	Stores/Heaps

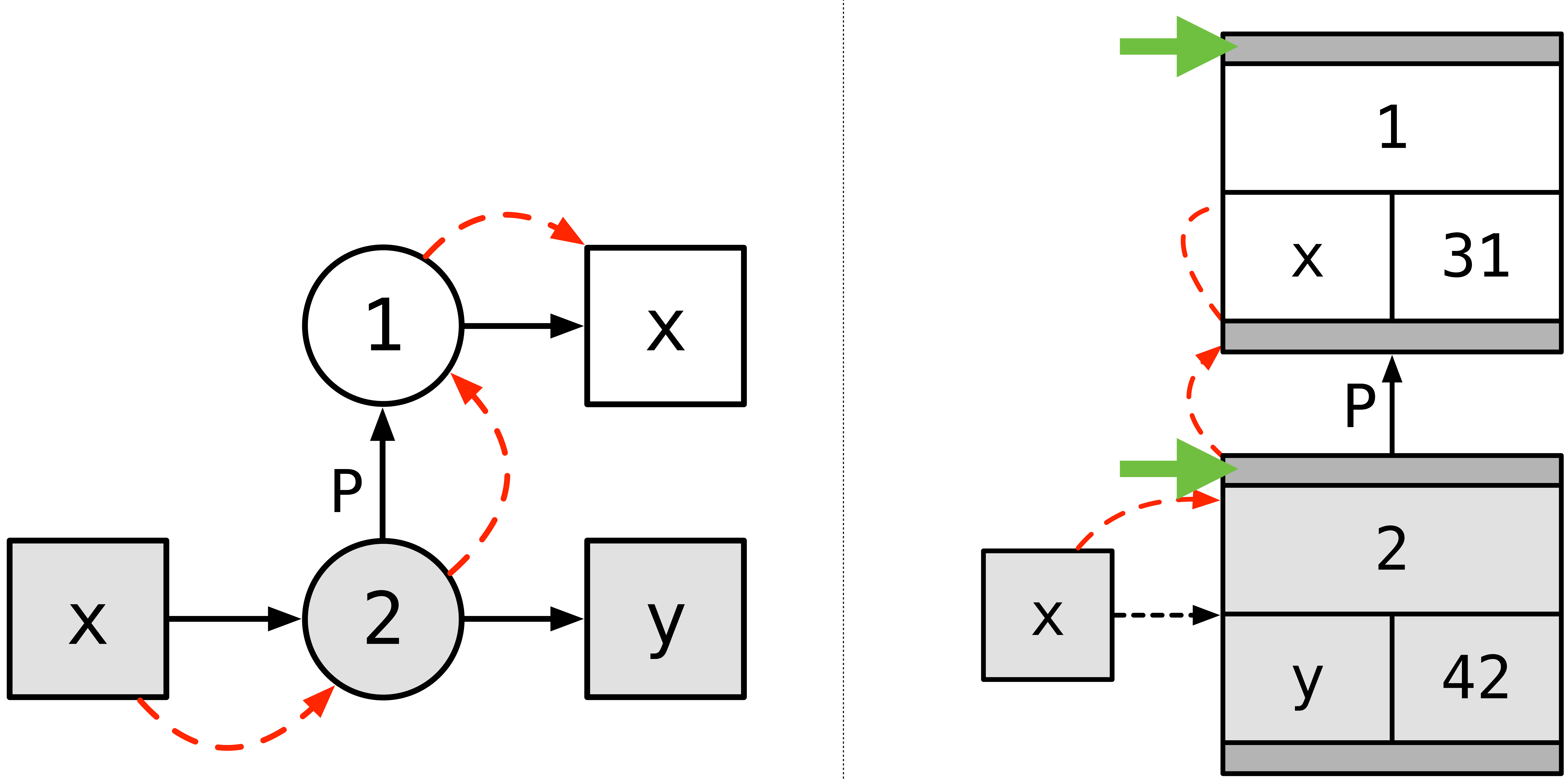
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
		Class Tables	Mutable Objects
			Stores/Heaps

		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`		Stores/Heaps

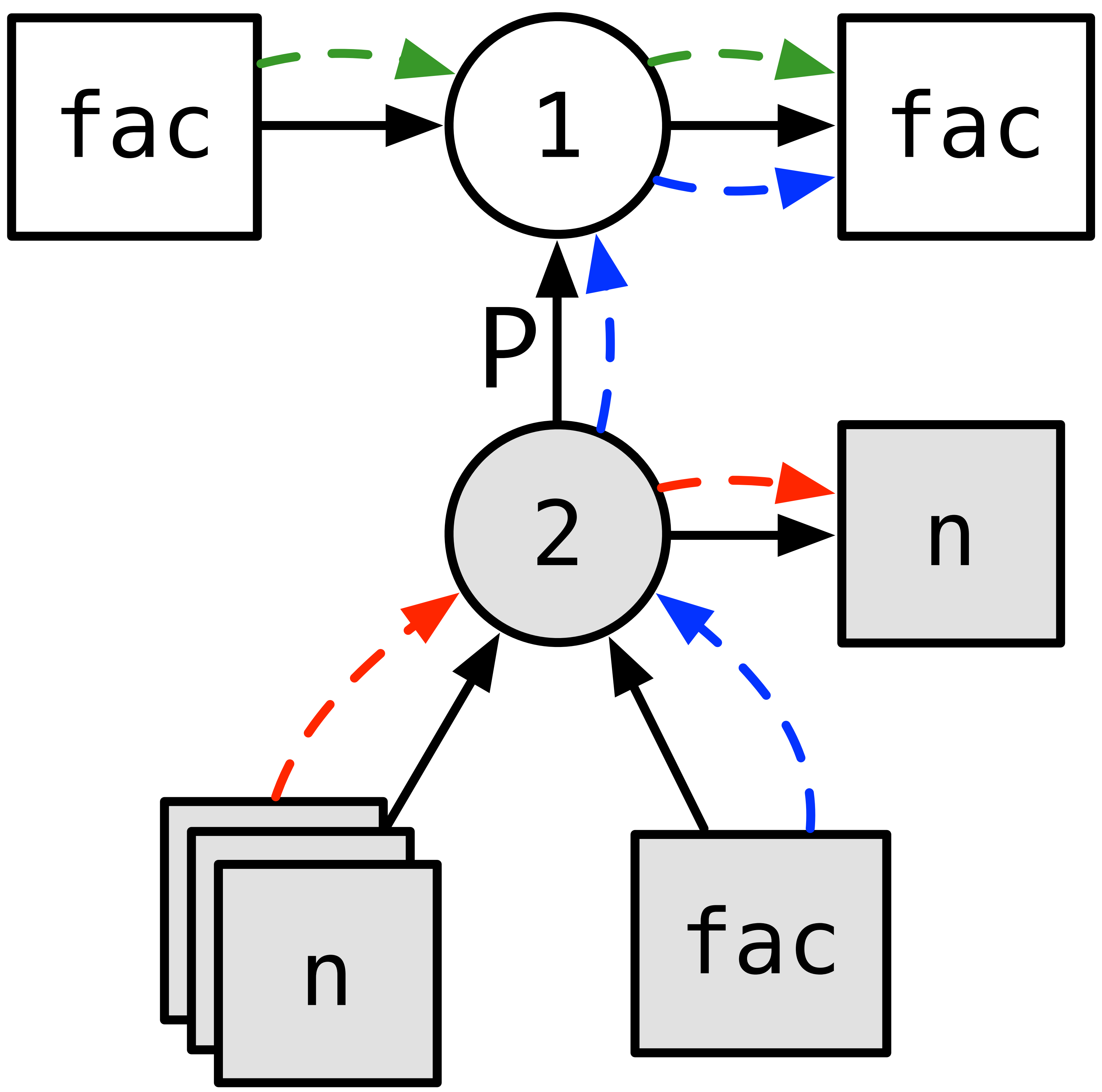
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
			Mutable Objects
			Stores/Heaps

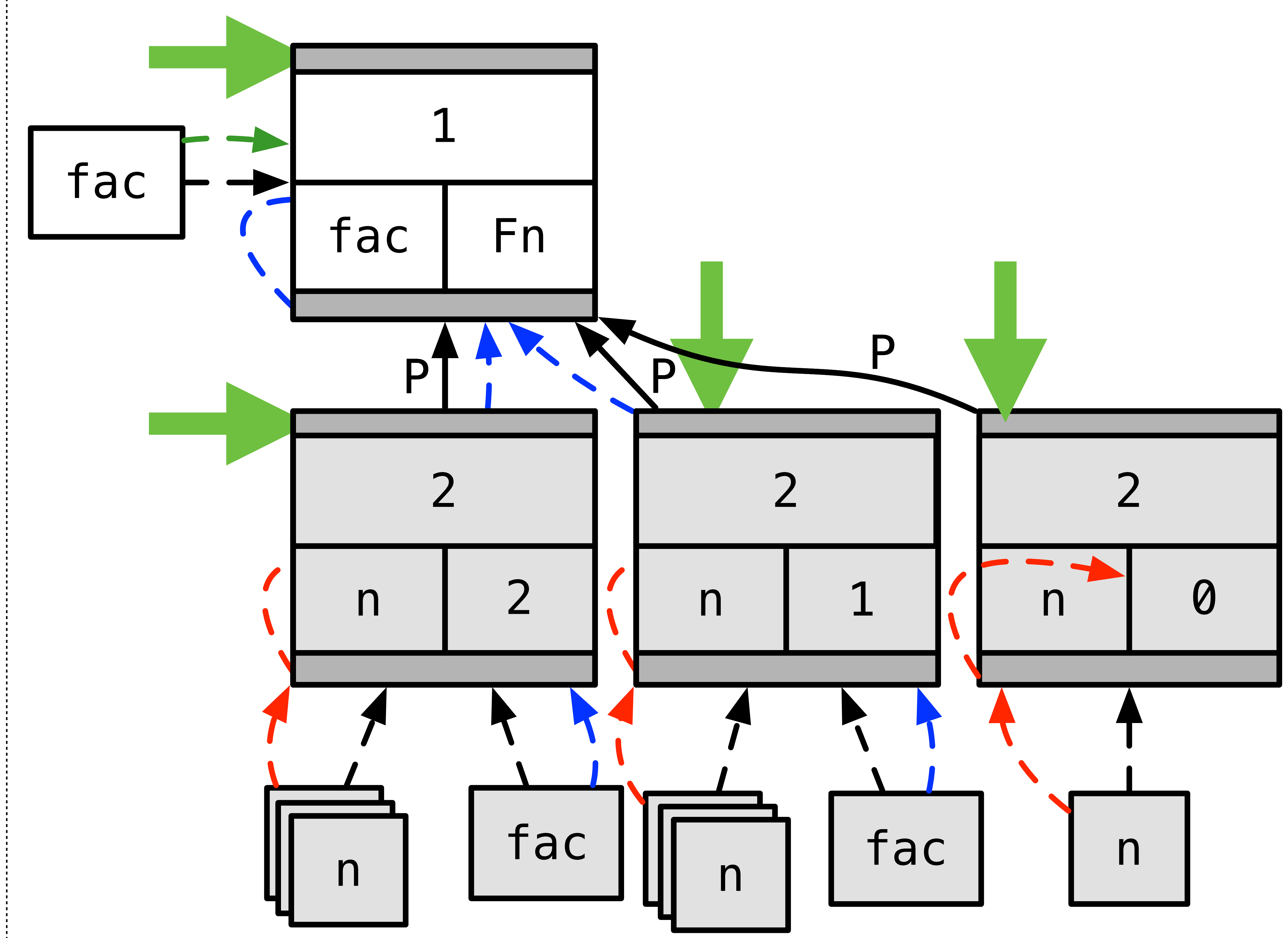

Scope
Frame
S'
S''
l
l'
s
x
y
S'
S''
l
l'
Links
s
x
y
Slots
[ECOOP'16]

```
val x = 31;
val y = x + 11;
```

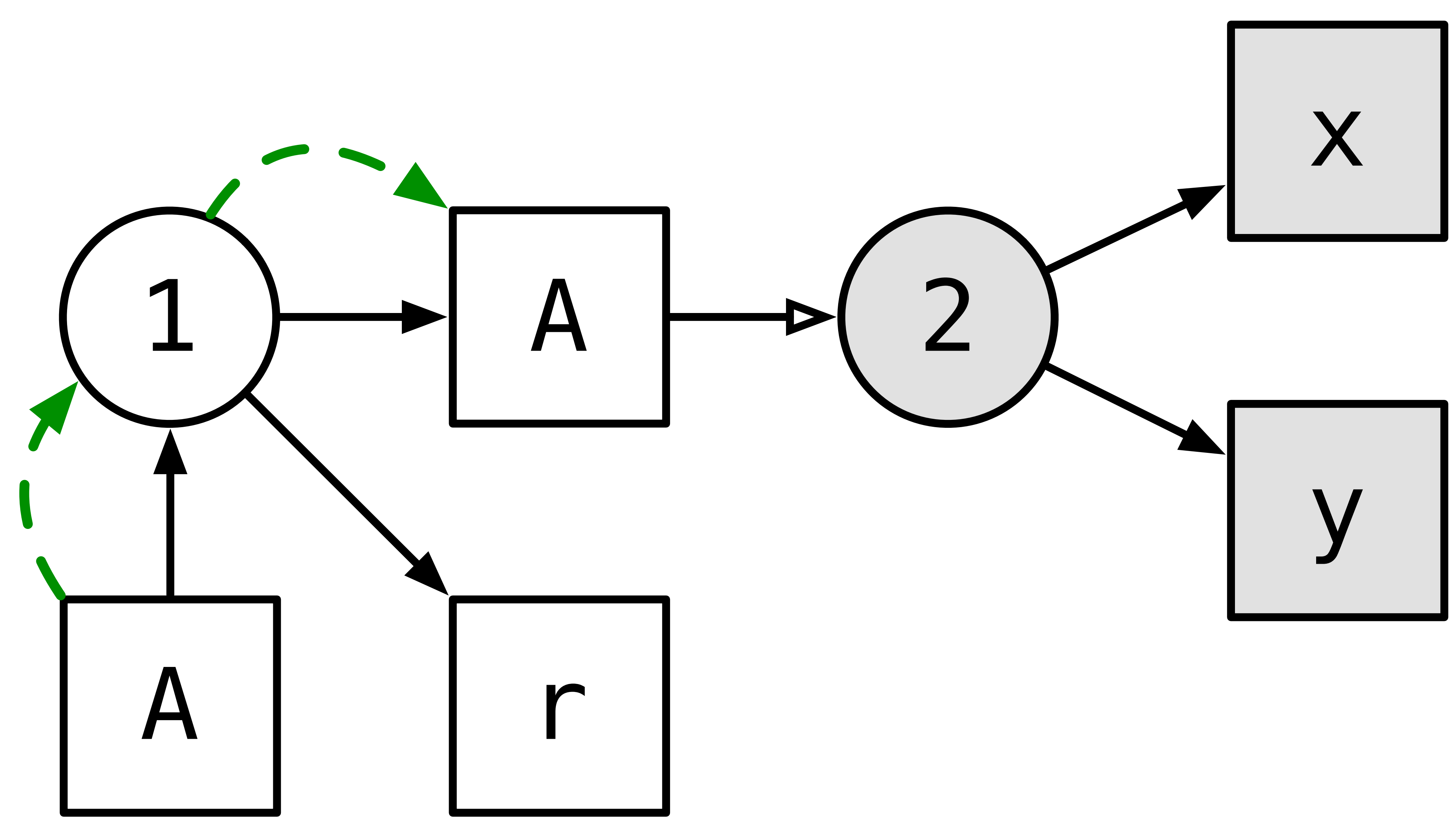


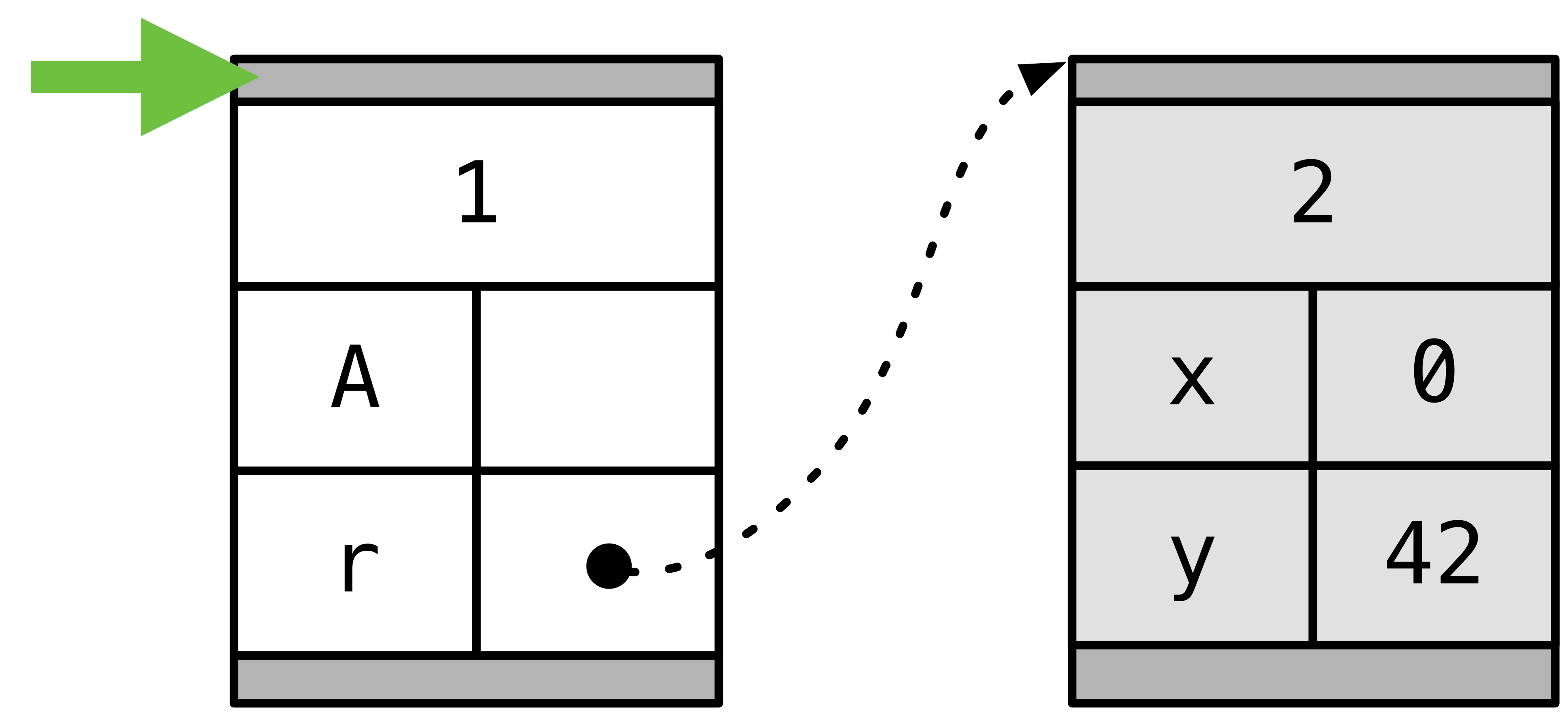
```
def fac(n : Int) : Int = {
  if (n == 0) 1
  else n * fac(n - 1)
};
fac(2);
```

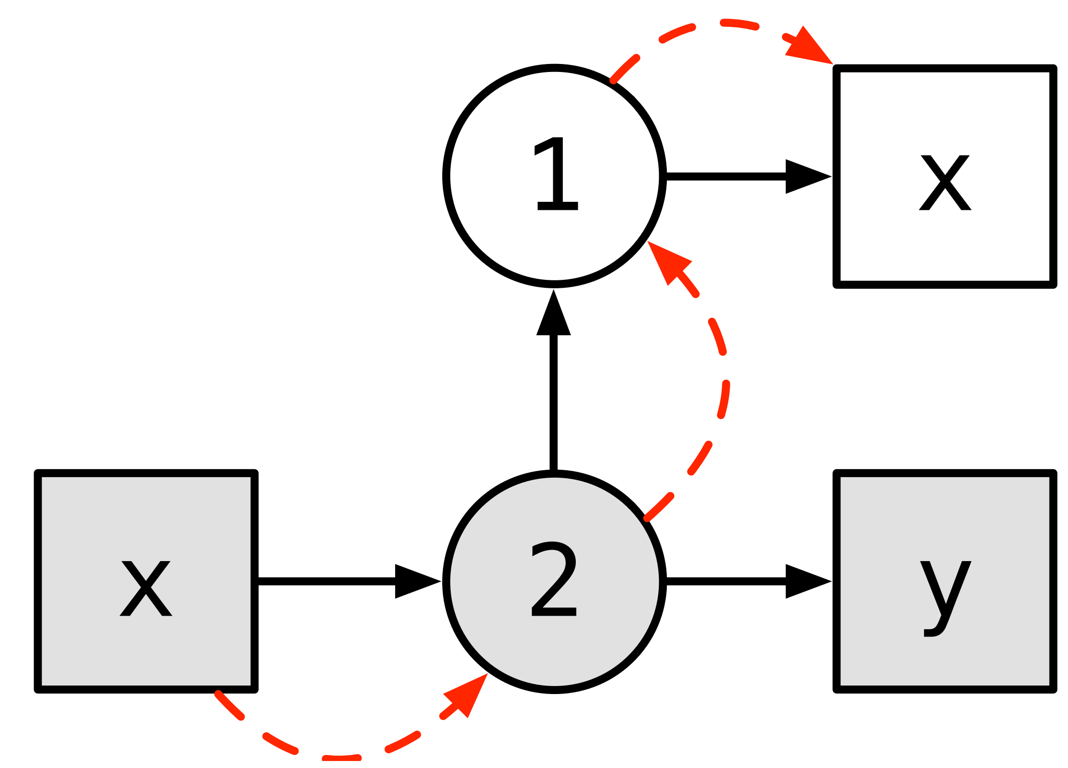
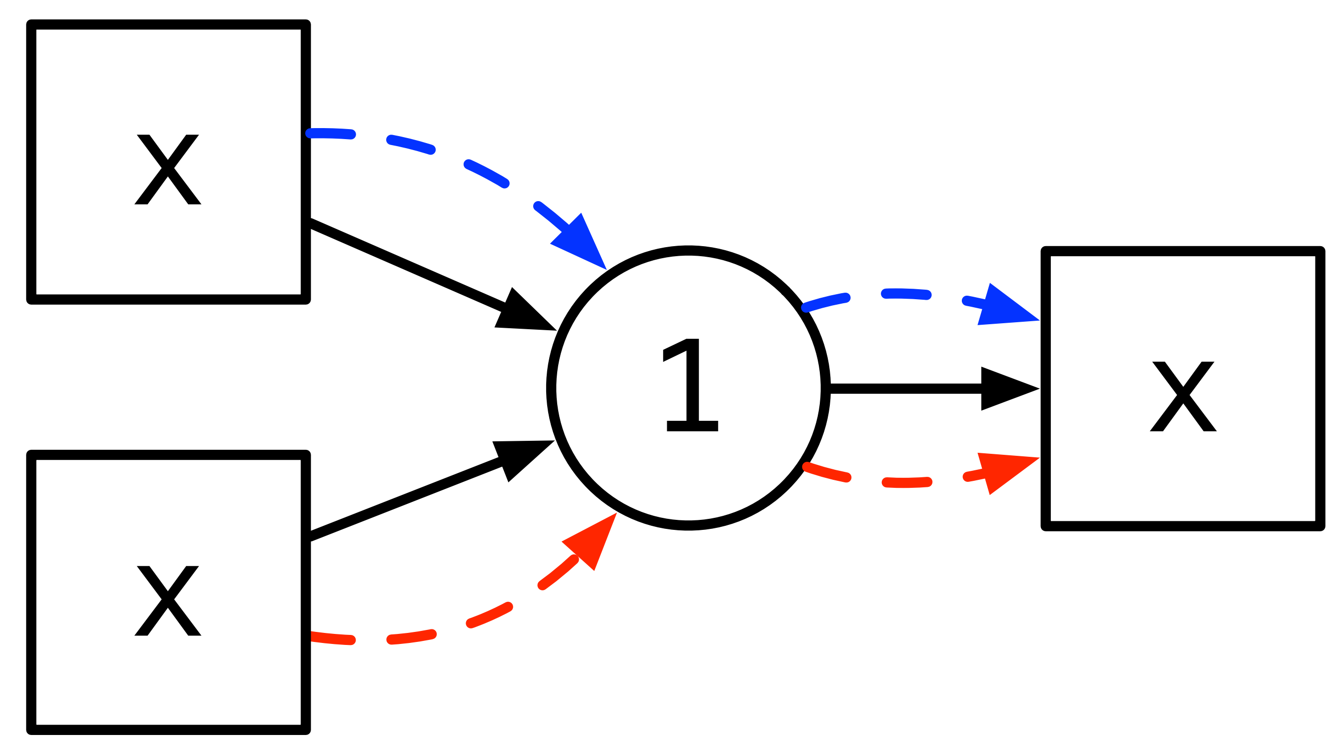


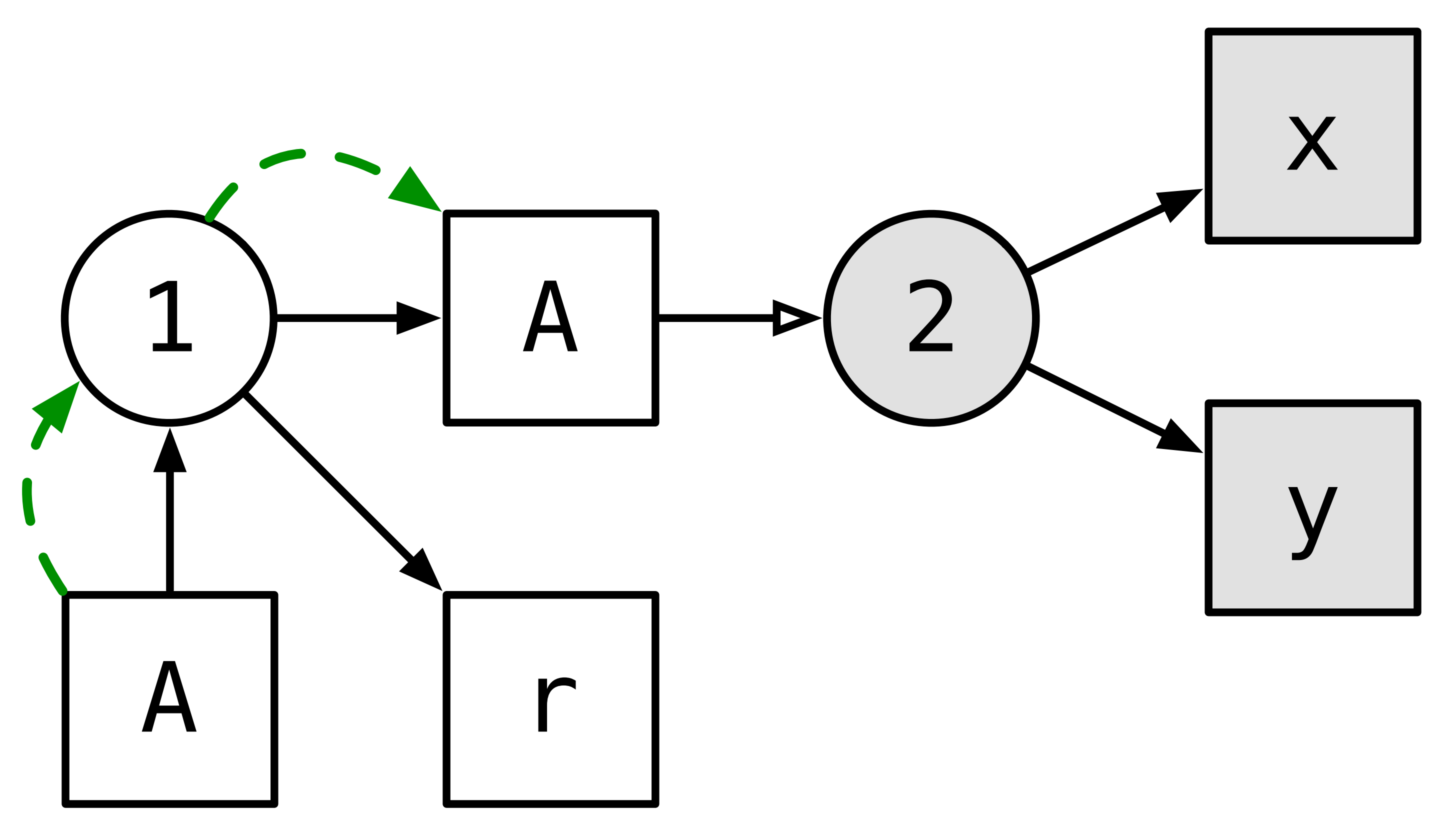


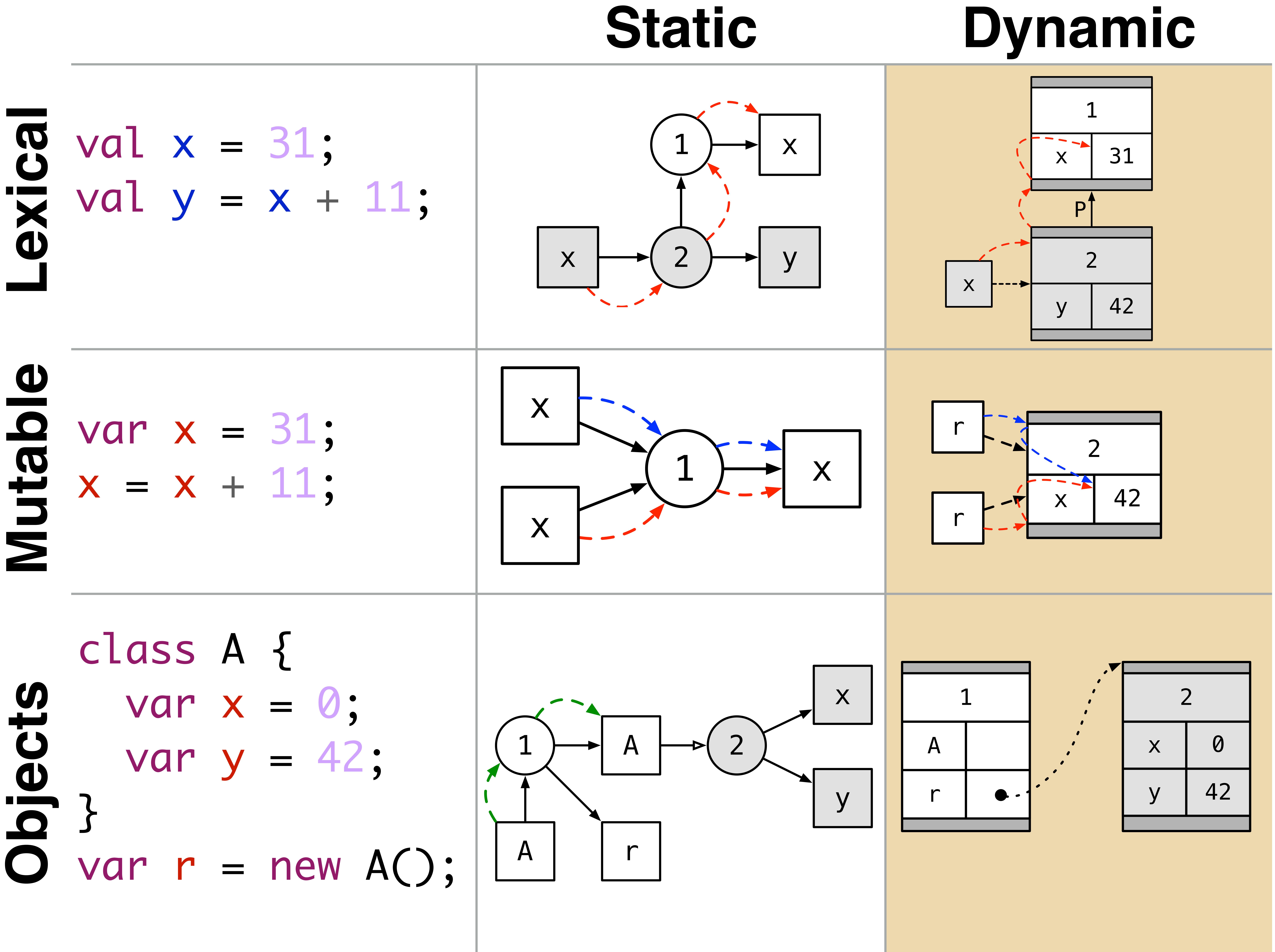
```
class A { var x = 0;
          var y = 42; }
var r = new A();
```



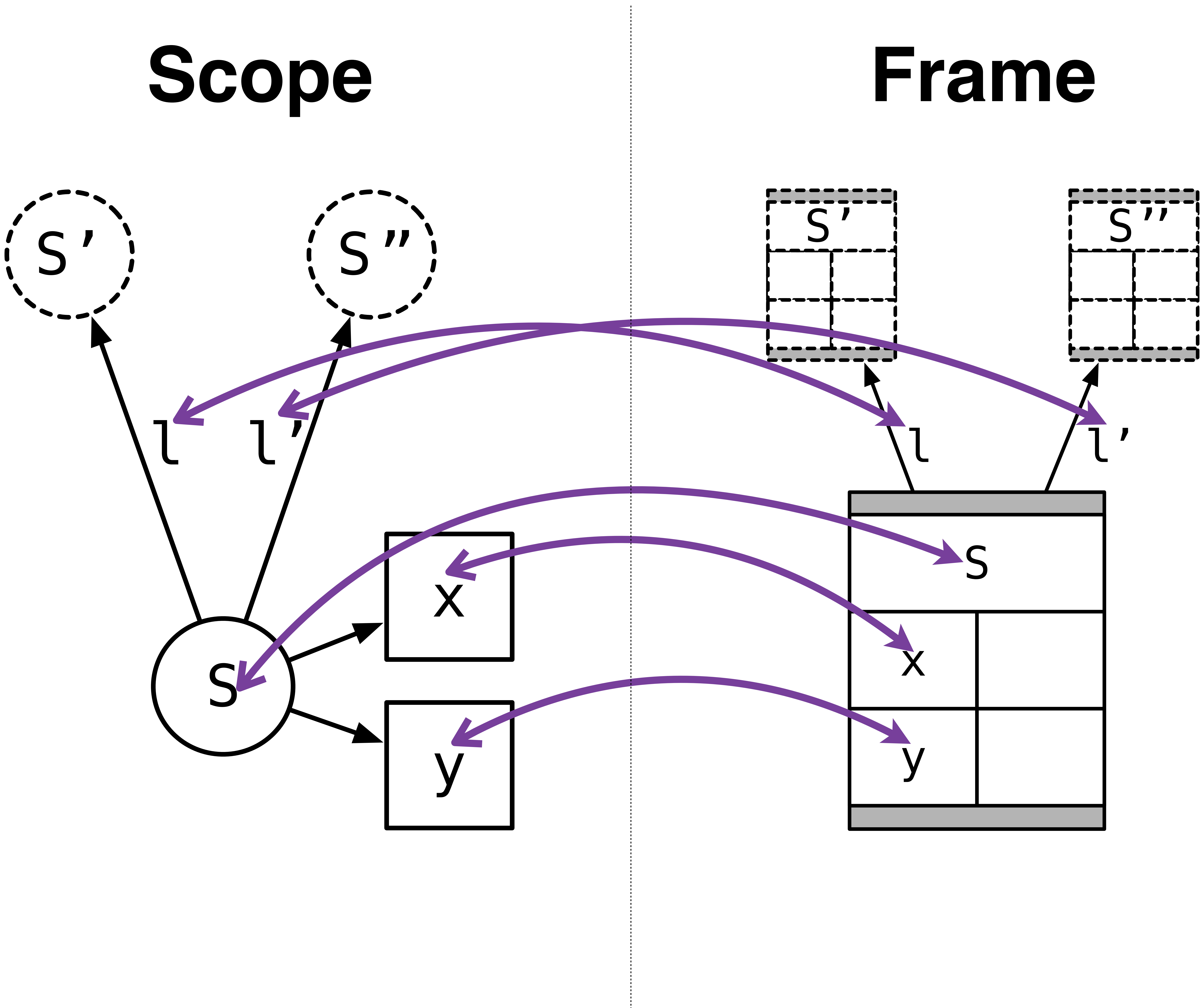


		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`		Stores/Heaps

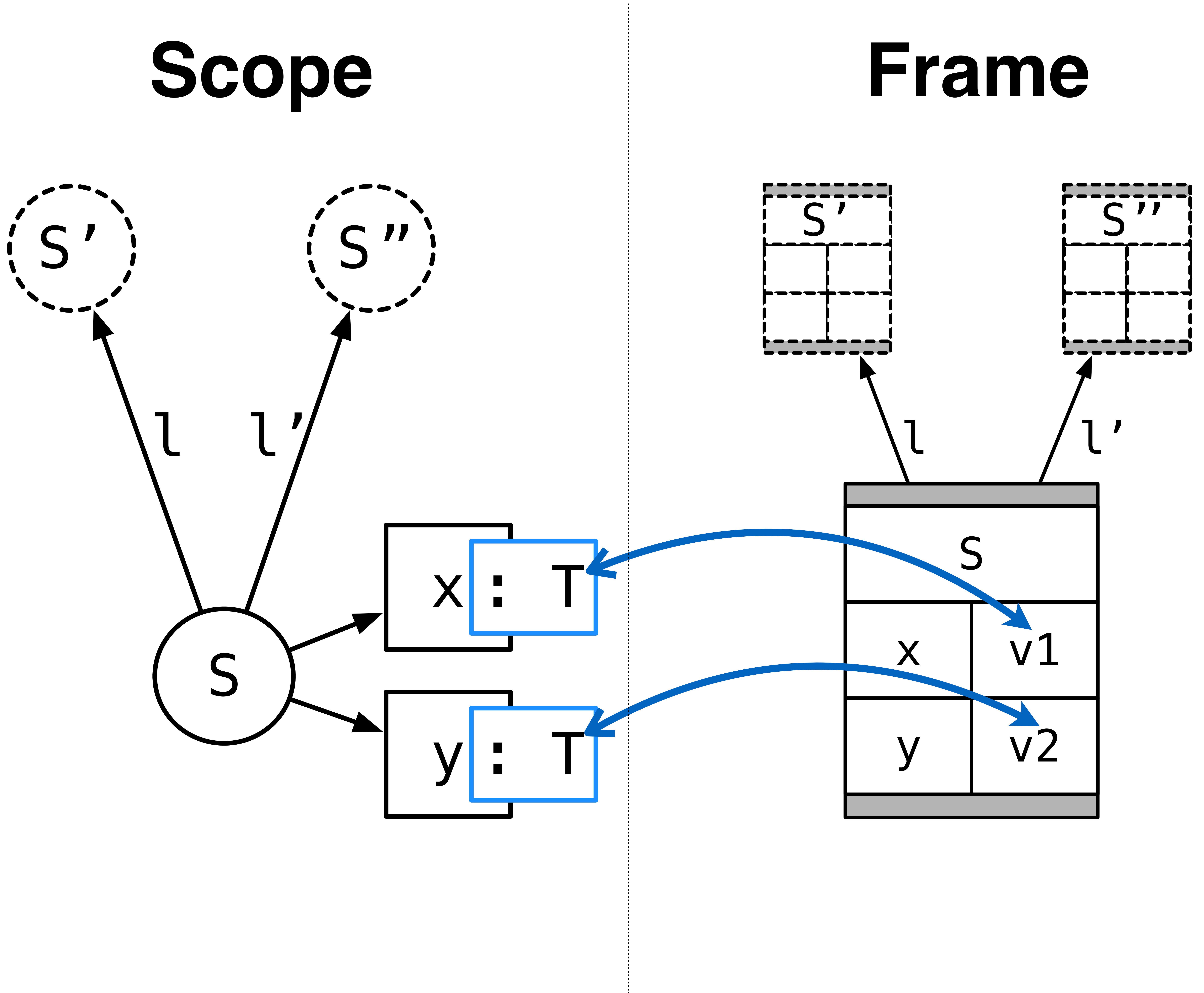
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
			Mutable Objects
			Stores/Heaps


Static
Dynamic
Lexical
val x = 31;
val y = x + 11;
Mutable
var x = 31;
x = x + 11;
Objects
class A {
var x = 0;
var y = 42;
}
var r = new A();
1
2
x
y
31
42
P
r
A
0

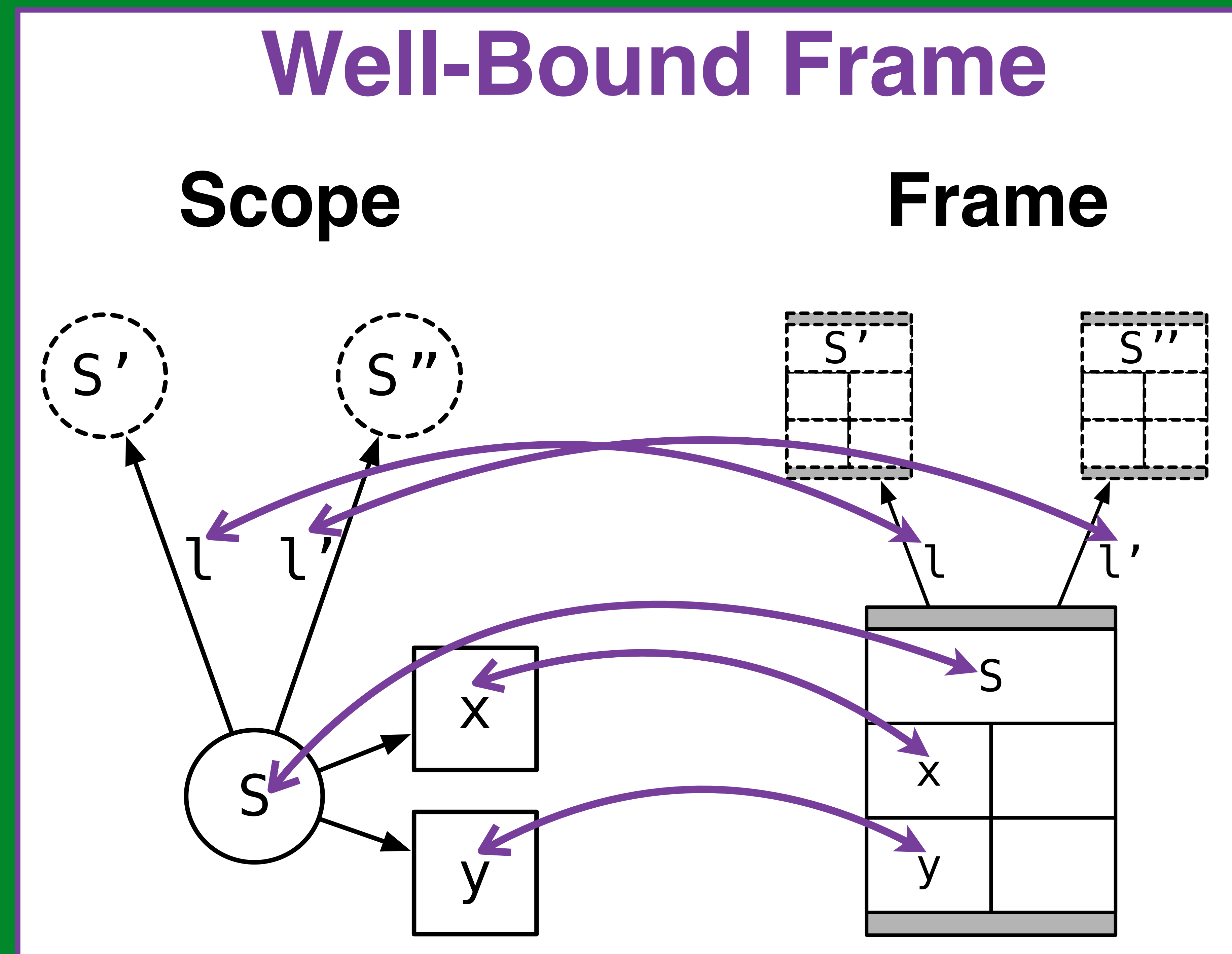
Well-Bound Frame

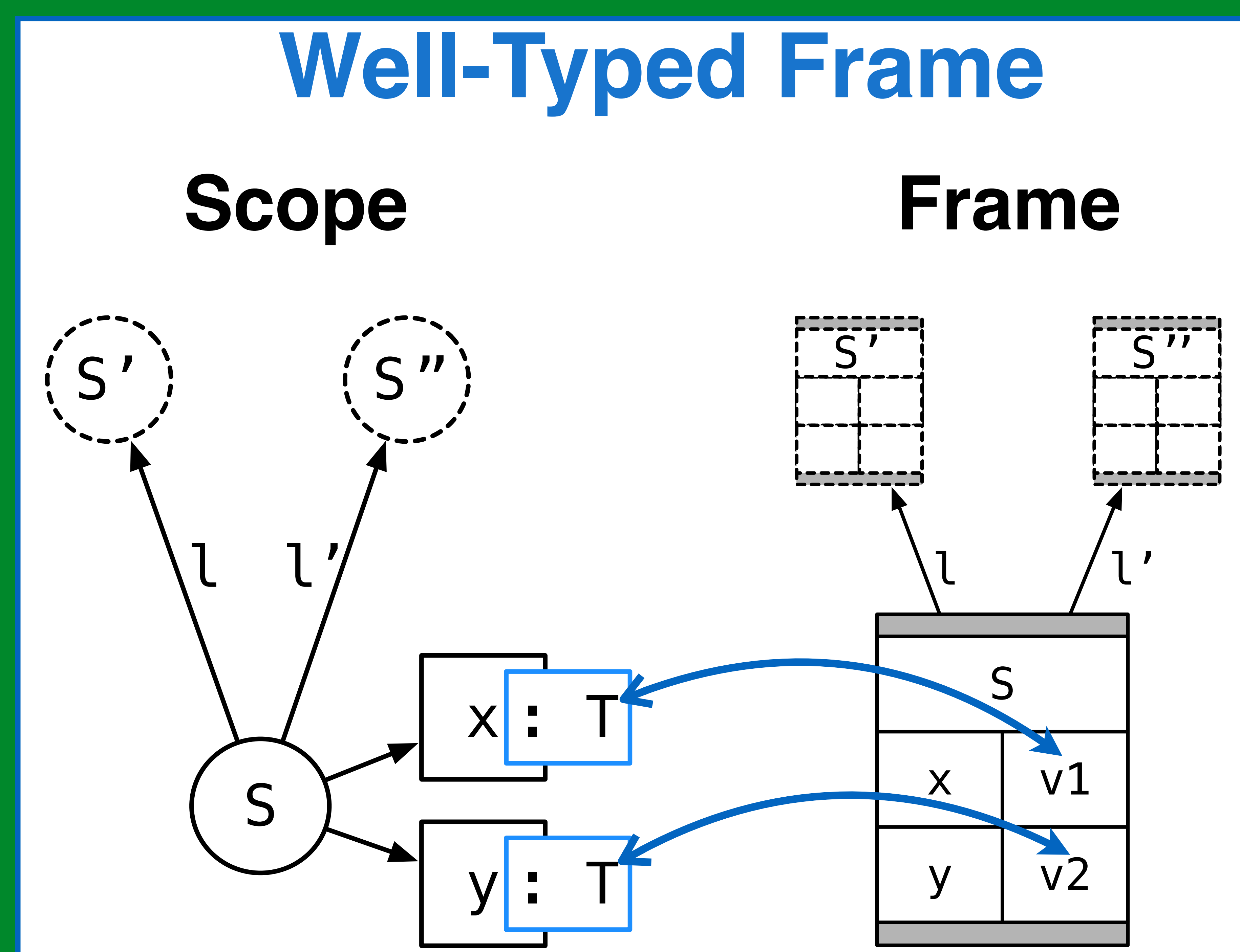


Well-Typed Frame



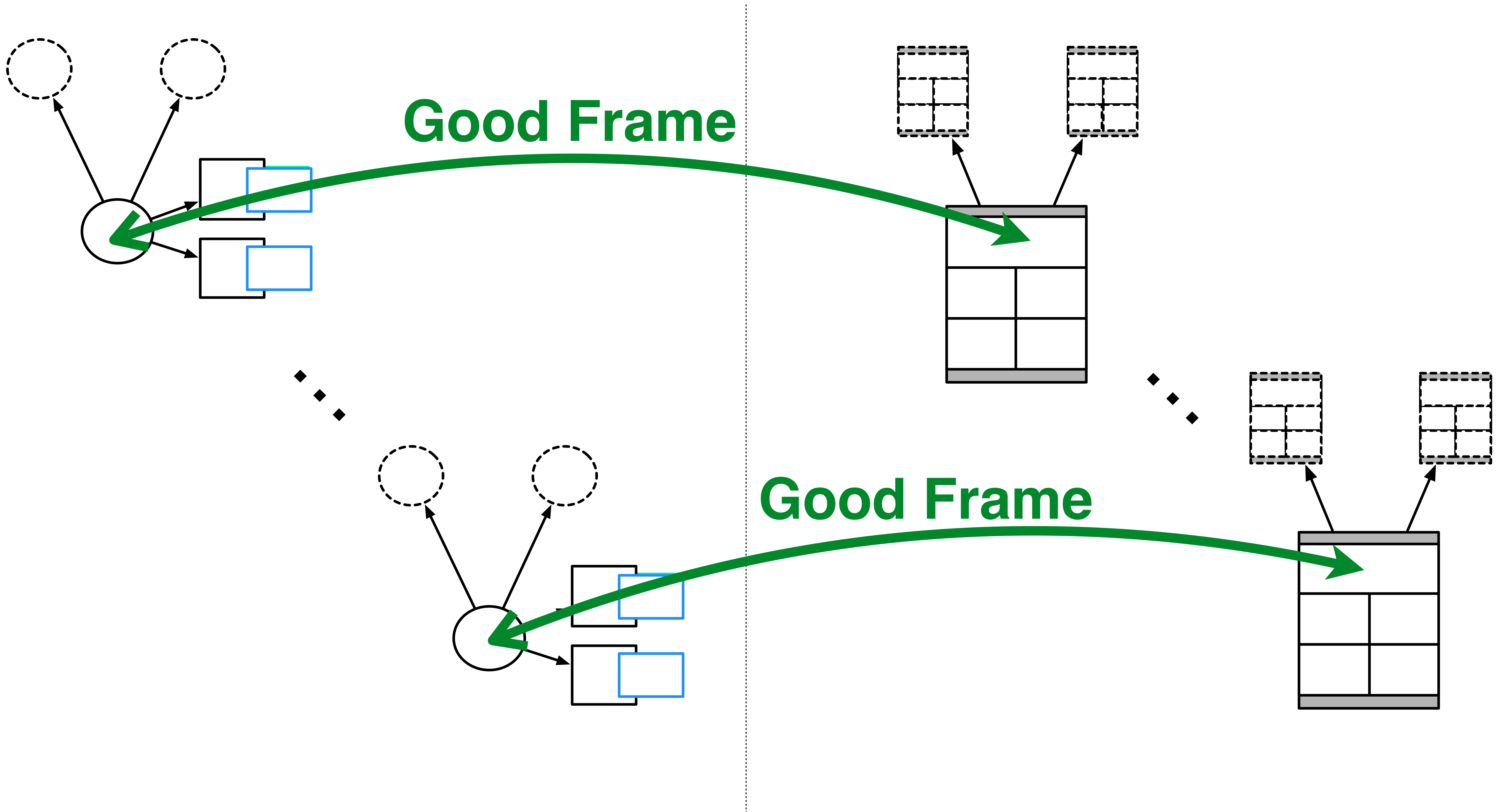
Good Frame Invariant



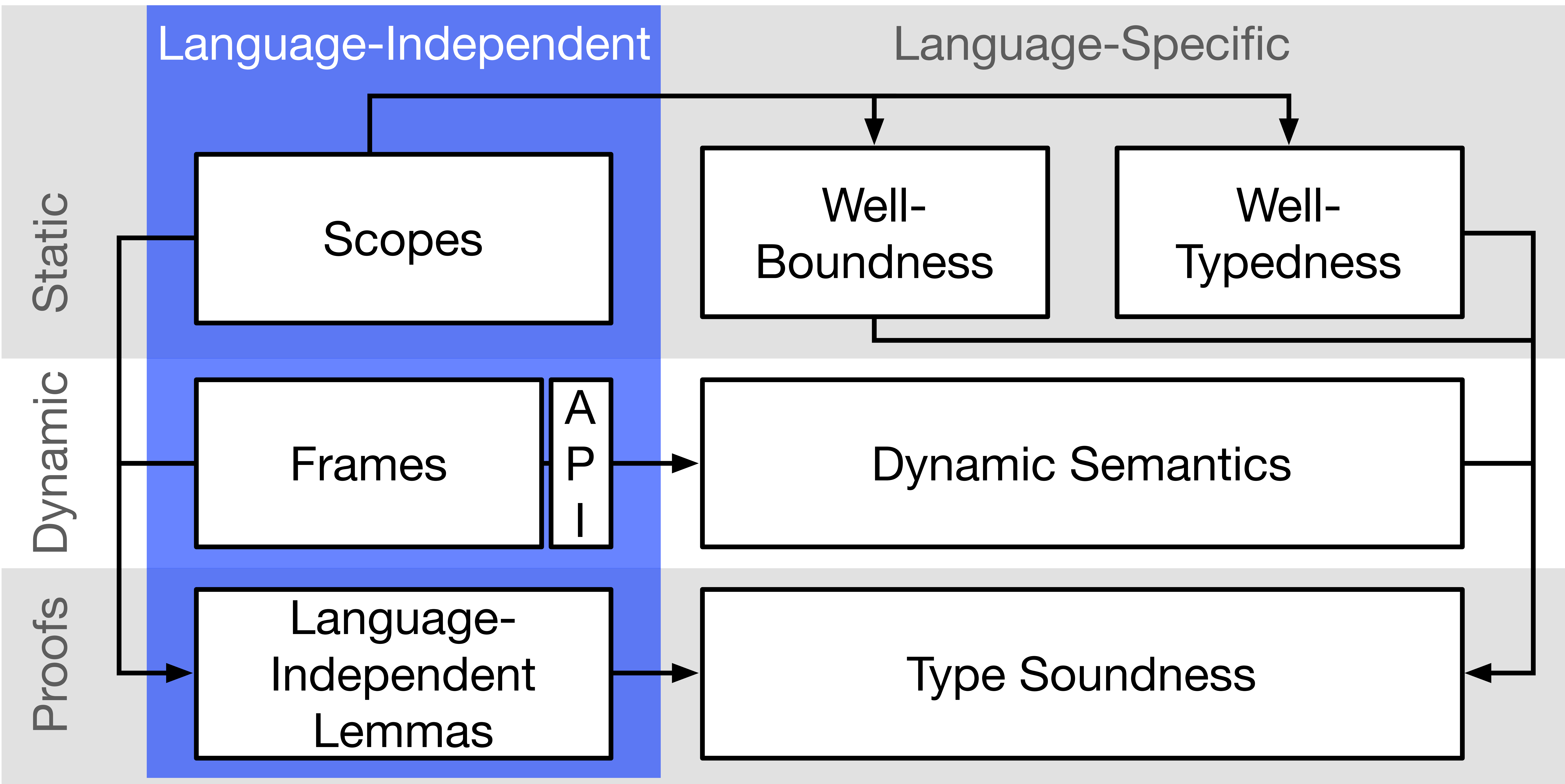


Good Heap Invariant

Every Frame is Well-Bound and Well-Typed

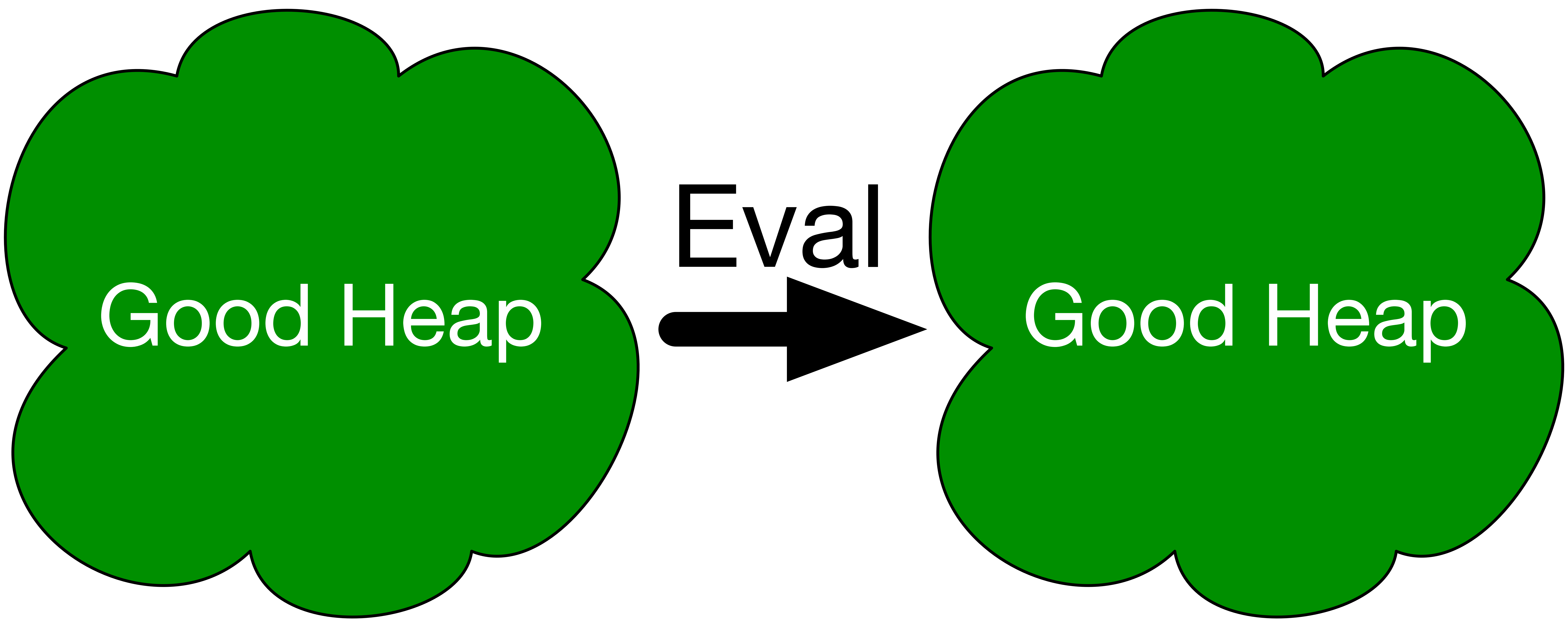


Architecture of a Specification



Type Soundness Principle

Evaluation Preserves
Good Heap Invariant



Summary

Summary

Compilers provide de-facto semantics to programming languages

=> often unclear

Formal specification of source language

is essential to pin down design

Hard requirement for future programming languages

Formal semantics should be live (connected to implementation)

and understandable (through readable meta-DSL)

Research Agenda

Abrupt termination?

Concurrency?

More case studies

Interpreter Generation

Optimization

Targeting (Graal+Truffle)/PyPy?

Type Soundness Verification