Dynamic Semantics

Eelco Visser

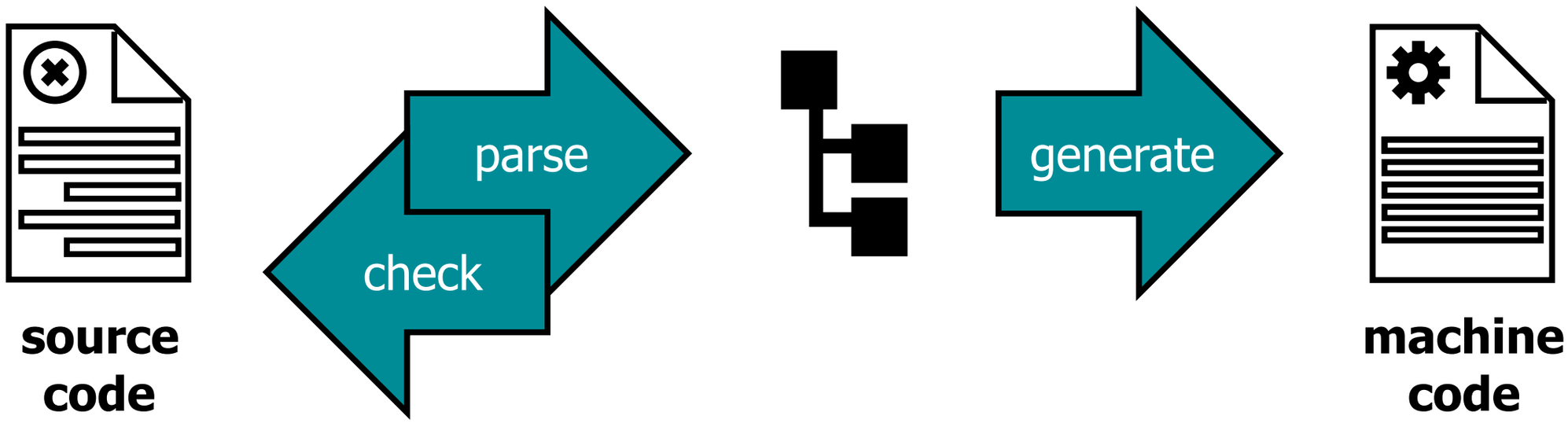
Outline

- The meaning of programs
- Operational semantics
- DynSem: A DSL for dynamic semantics specification
- Interpreter generation
- Scopes describe frames

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

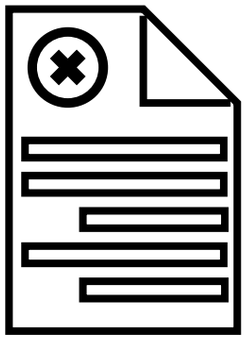
`semanticsL1(p) = semanticsL2(translate(p))`

Requires understanding `translate` and `semanticsL2`

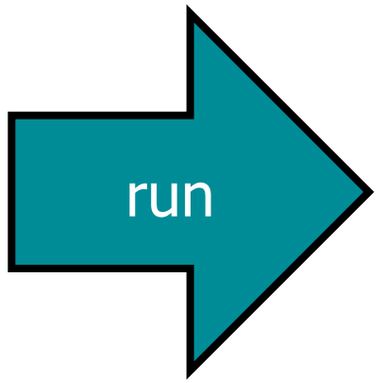
How do we know that `translate` is correct?

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

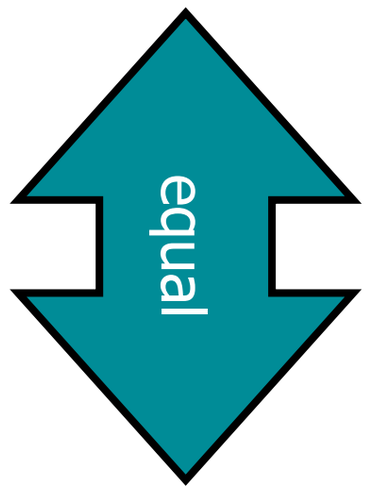
semanticsL1

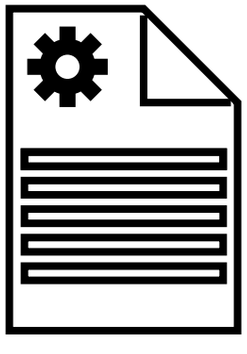


source code

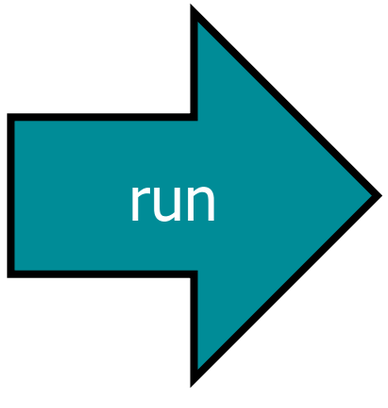


value





machine code



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

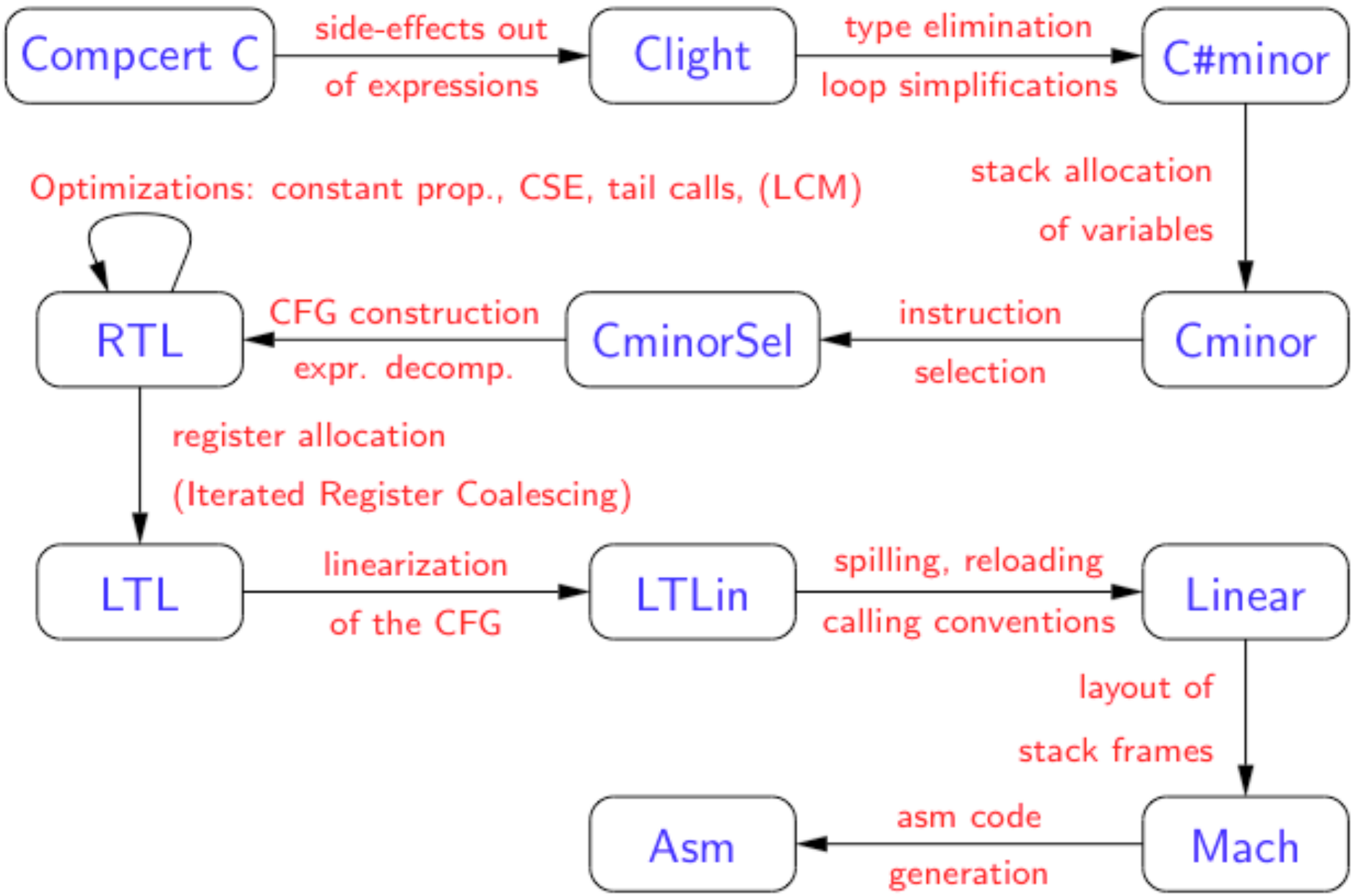
Is this the right semantics?

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 $\dfrac{e_1 \rightarrow e_1' \ \ldots \ e_n \rightarrow e_n'}{e \rightarrow e'}$

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
```

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

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

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

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

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

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

$e \rightarrow e$

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
```

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

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

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

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

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

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

$e \rightarrow e$

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?

Natural (Big-Step) Semantics

$$e = x \mid i \mid e + e \mid \backslash x.e \mid e\ e \mid \text{ifz}(e)\ e\ \text{else}\ e$$

$$\frac{}{E \vdash i \Rightarrow \text{NumV}(i)}$$

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

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

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

$$E \vdash e \Rightarrow v$$

reducing expressions to values

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

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

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

DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, Eelco Visser

RTA 2015

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

Portable

High-performance

Big-Step

Modular

Concise

Unsurprising

Statically Typed

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

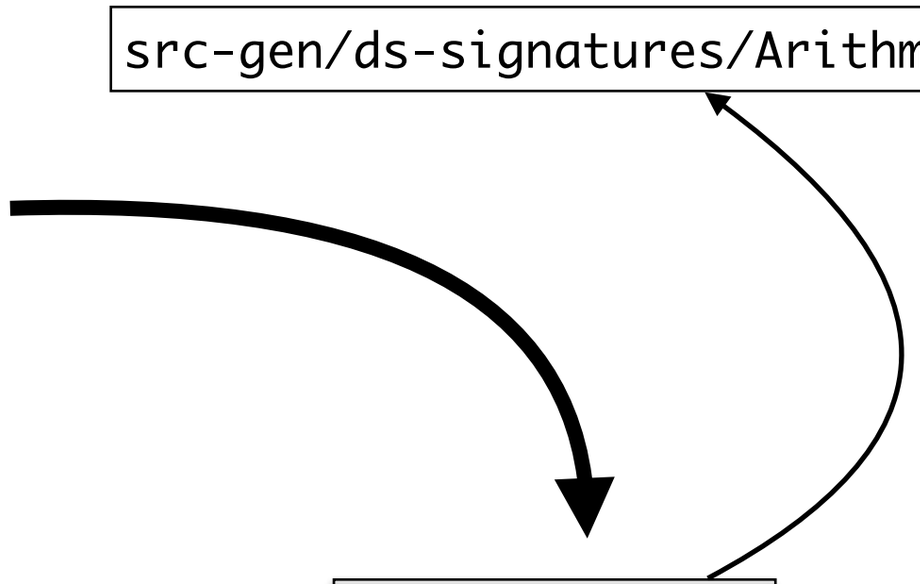
- Arithmetic
- Booleans
- Comparisons
- Mutable variables
- Functions
- Boxes

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 }
```




```
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

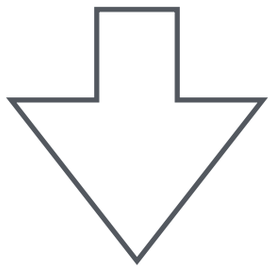
```
module primitives
signature
  native operators
    str2int : String -> Int
    plusI   : Int * Int -> Int
    minusI  : Int * Int -> Int
```

```
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
```

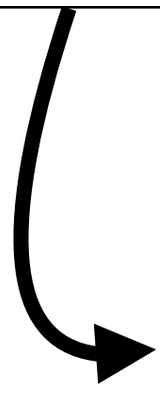


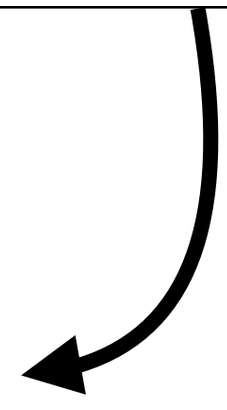
```
rules

  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.
```

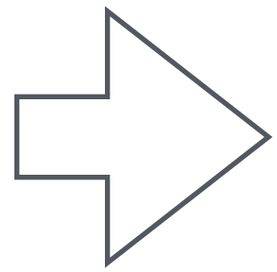




```
module comparison

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

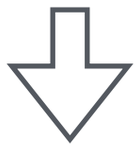
```
constructors
  Fun : ID * Expr -> Expr
  App : Expr * Expr -> Expr
```

```
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.
```

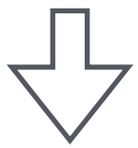
```
module environment

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 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}.
```

```
module store

imports values

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

Mutable Variables: Environment + Store

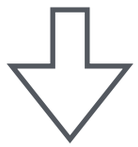
```
constructors
  Let : ID * Expr * Expr -> Expr
  Var : ID -> Expr
  Set : String * Expr -> Expr
```

```
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.
```

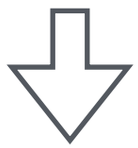
```
module store

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);
```

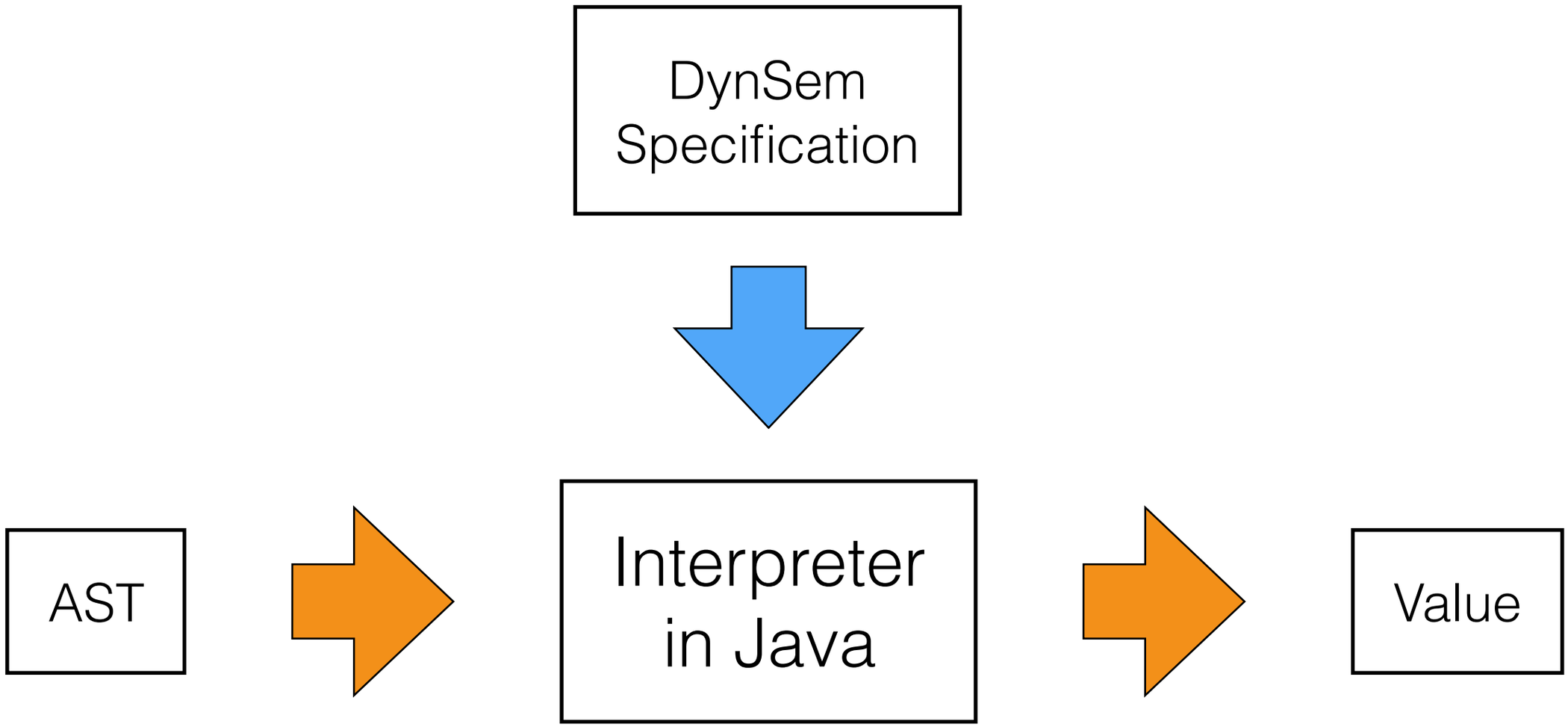
prettyprint.aterm

```
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"))]
```

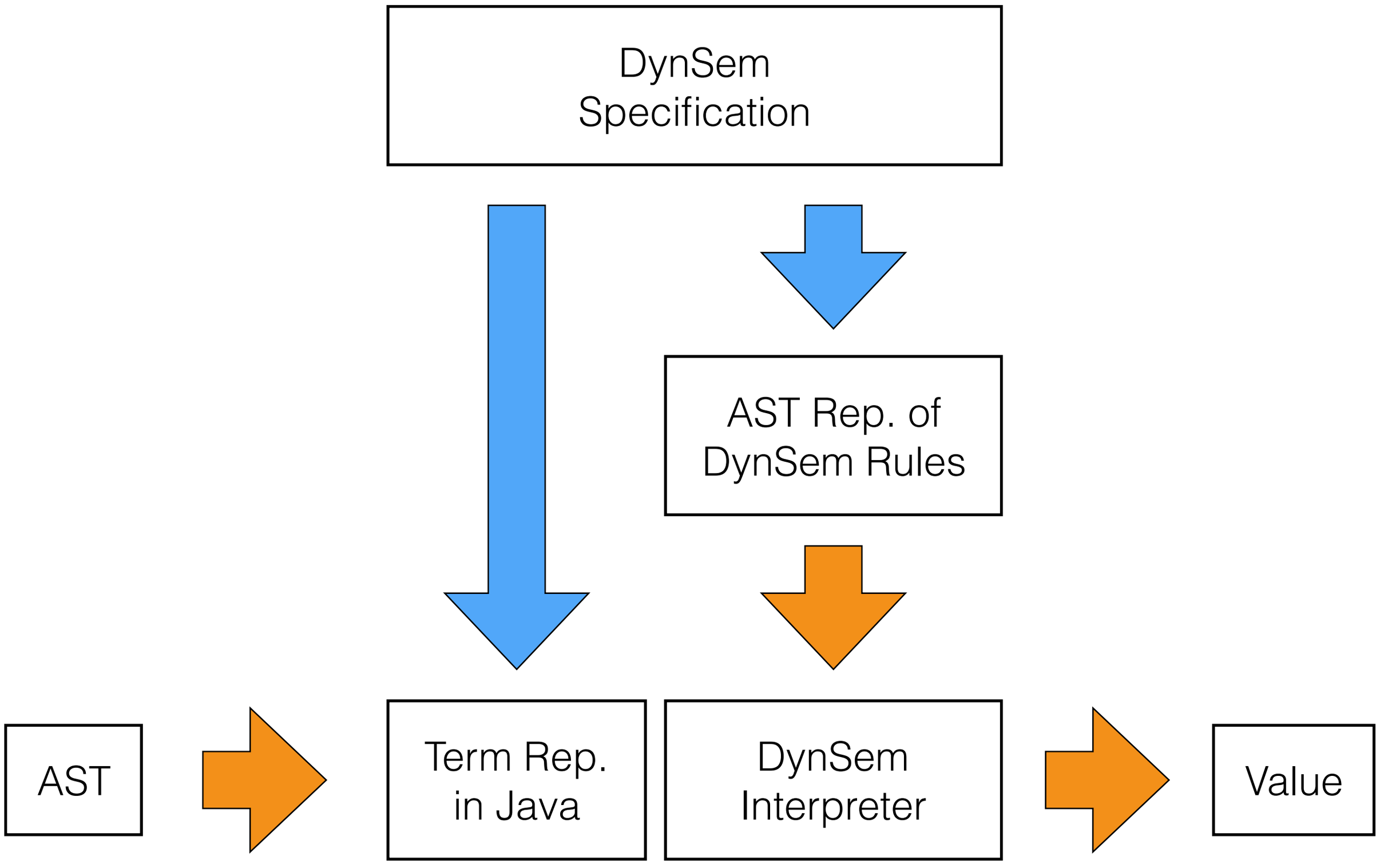
https://github.com/MetaBorgCube/metaborg-tiger

Interpreter Generation

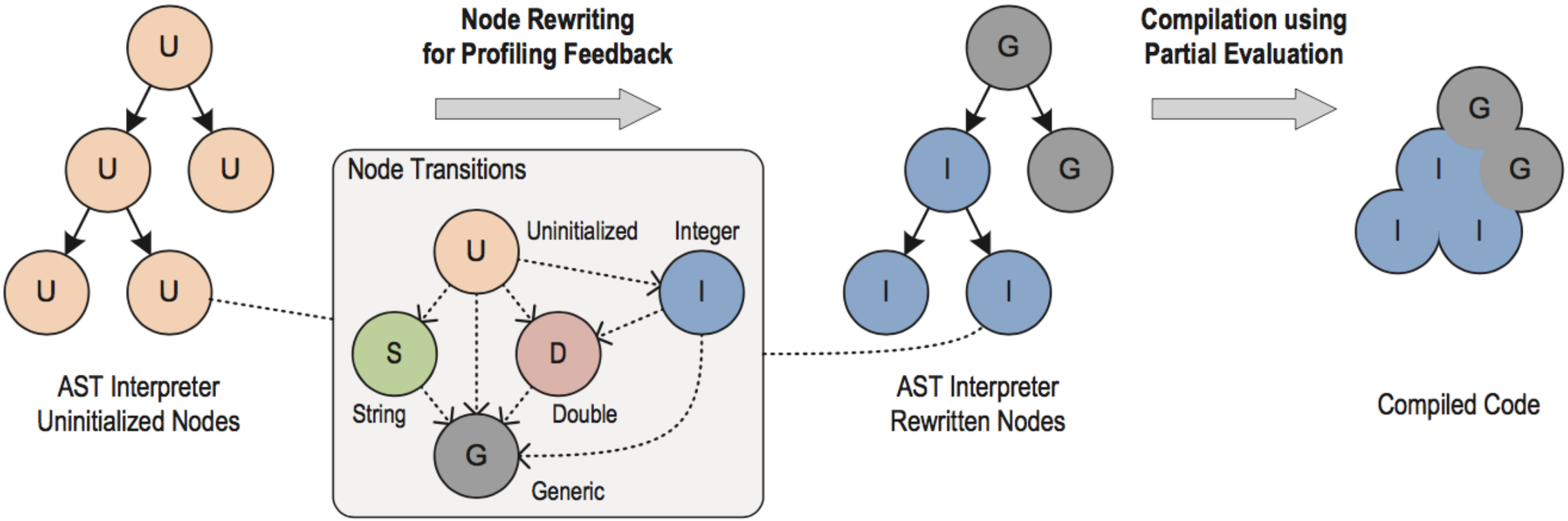
Generating AST Interpreter from Dynamic Semantics



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

Semantic Specification

Static Semantics

Binding

Dynamic Semantics

Binding

Garbage Collector

Type Soundness

Tools

IDEs

Type Checkers

Language Run Time

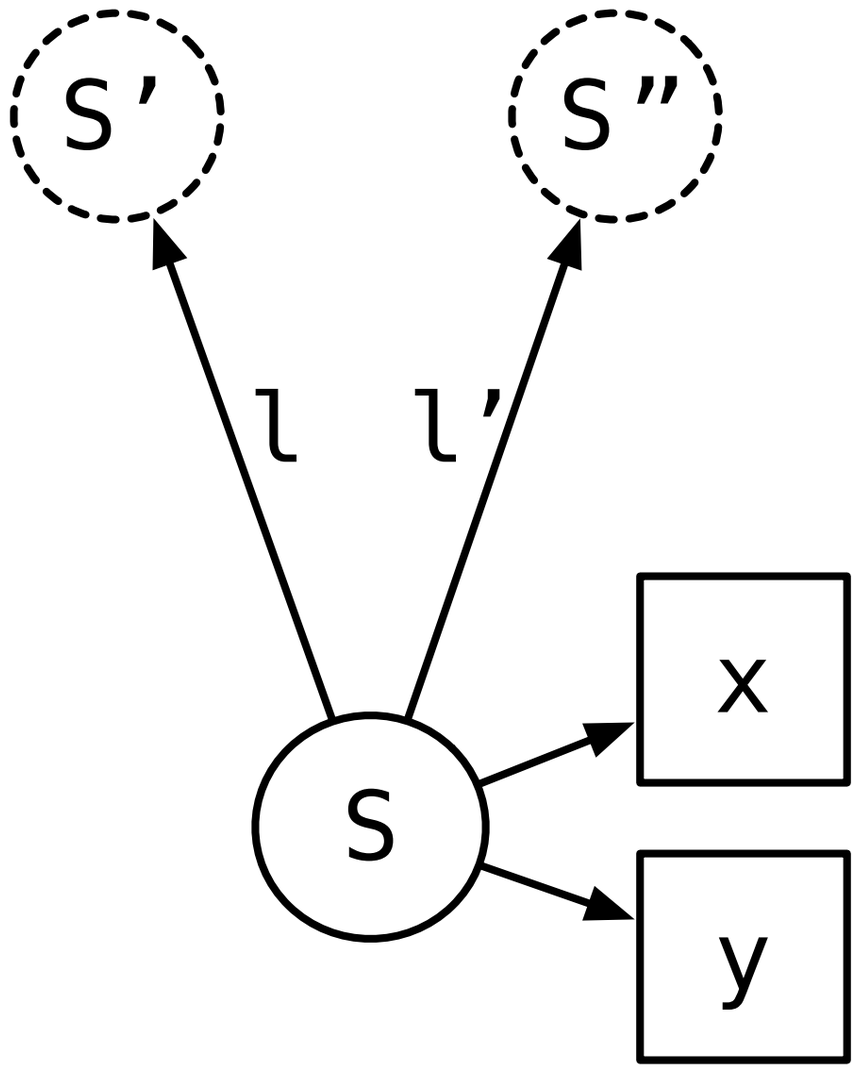
Garbage Collector

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

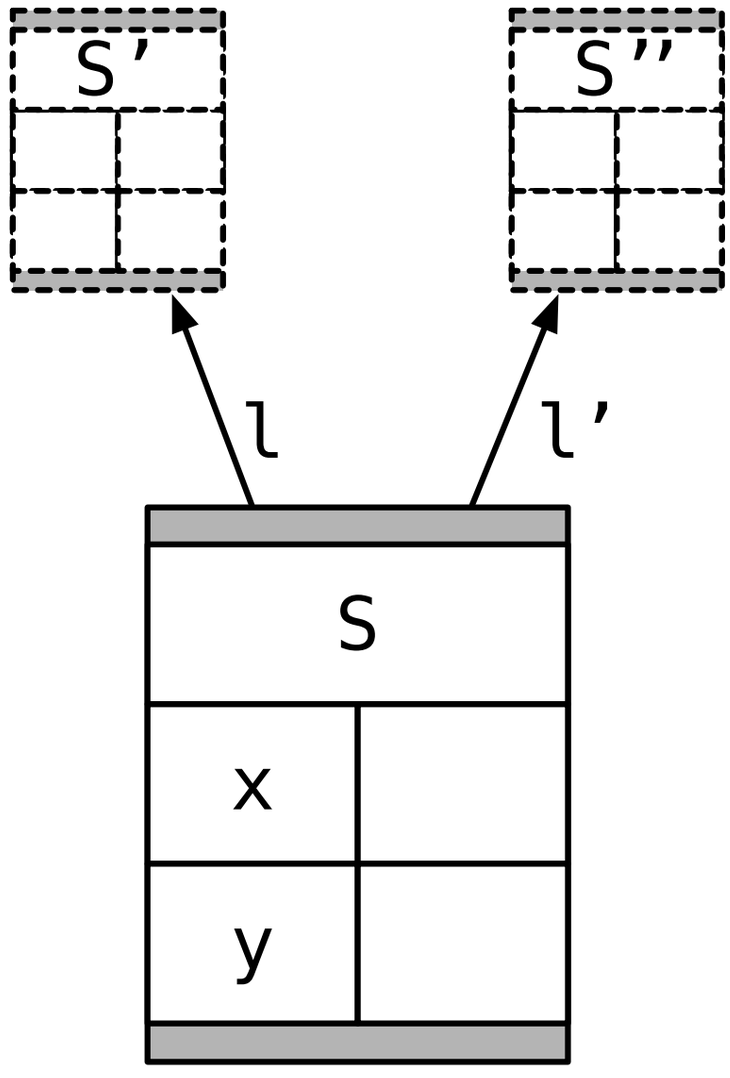
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
		Class Tables	Mutable Objects
			Stores/Heaps

Scope



[ESOP’15]

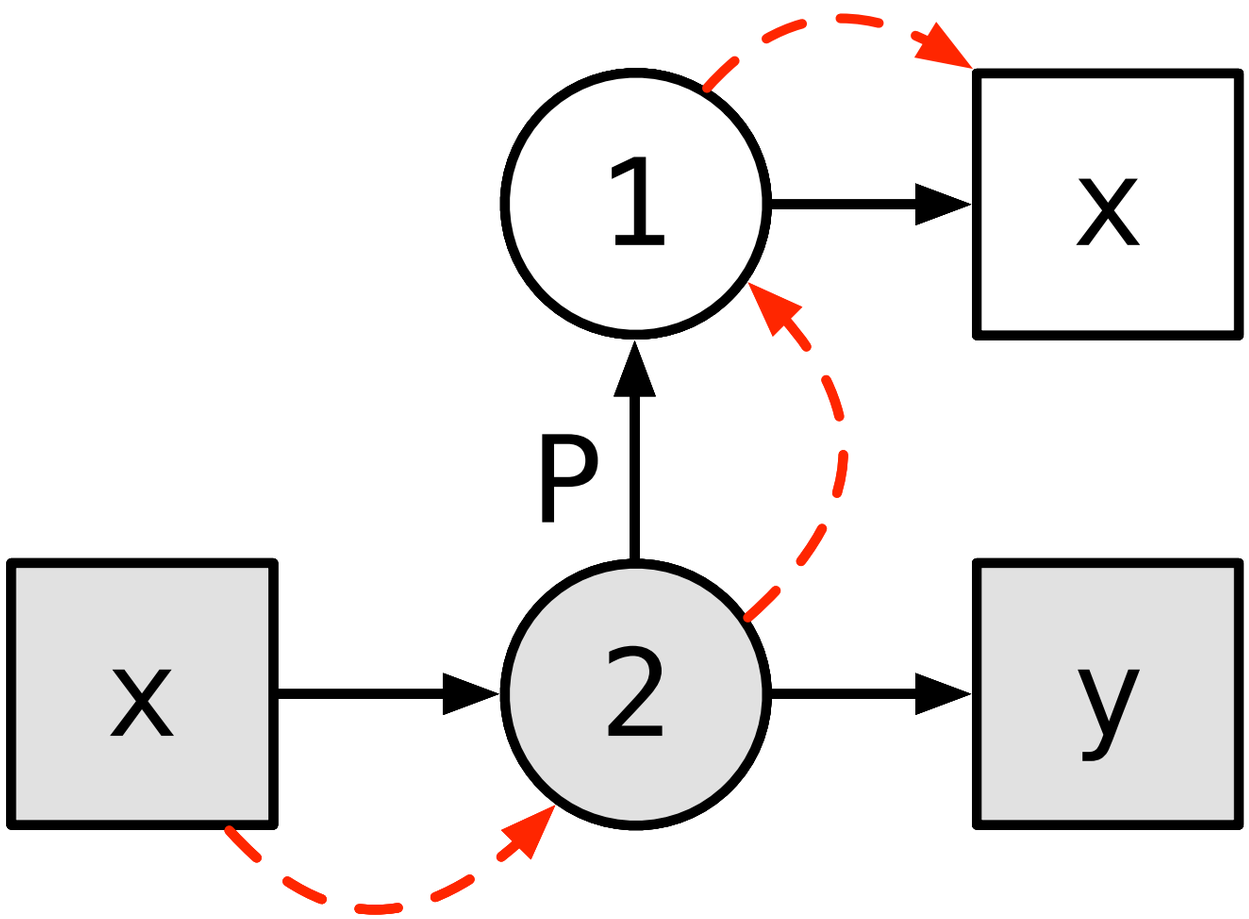
Frame



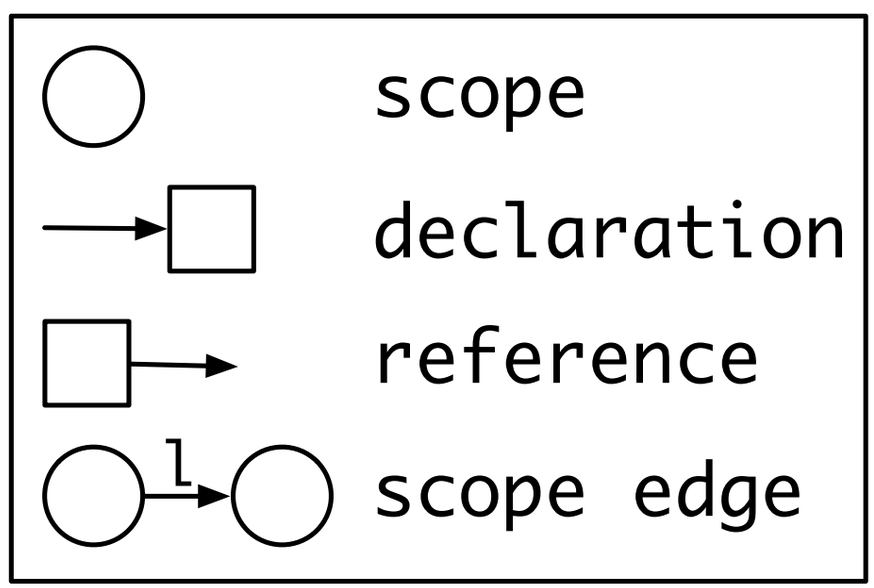
[ECOOP’16]

Lexical Scoping

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

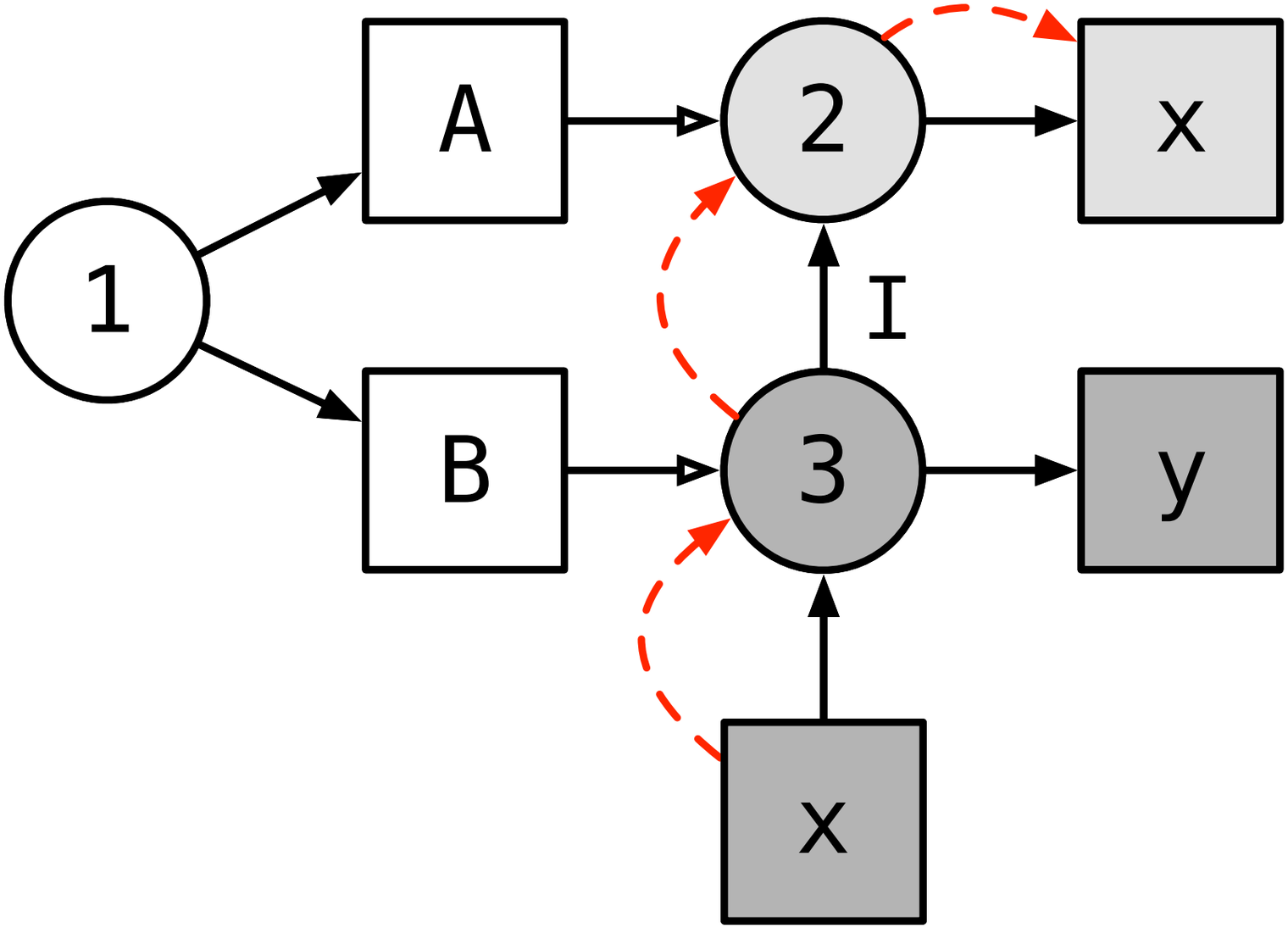


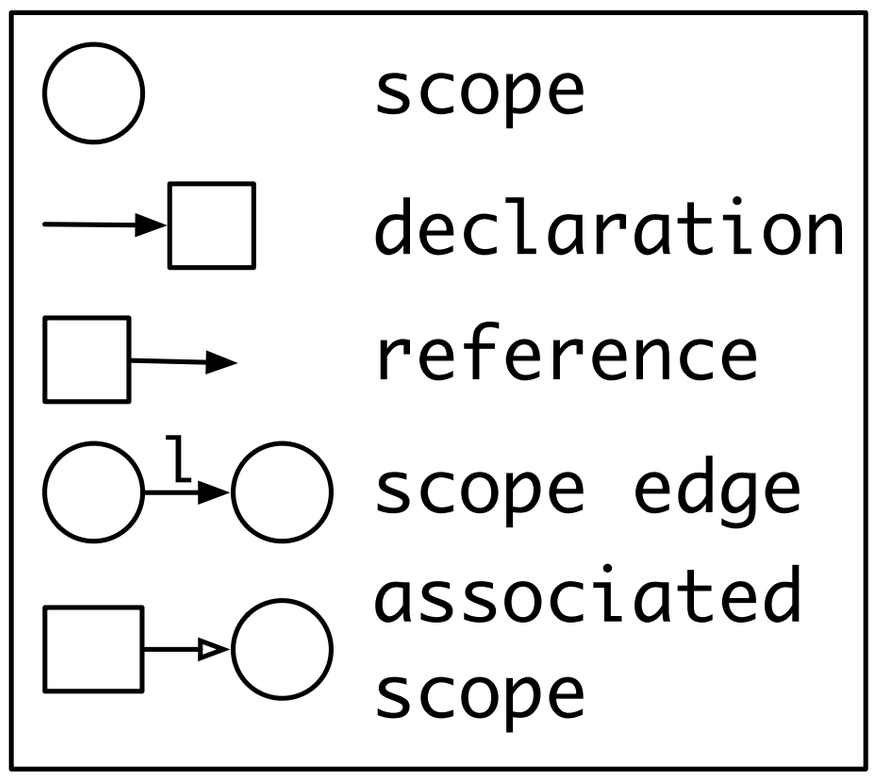
[ESOP'15; PEPM'16]



Inheritance

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

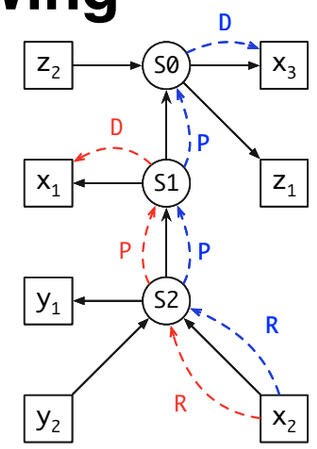




More Binding Patterns

Shadowing

```
def x3 = z2 5 7          S0

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



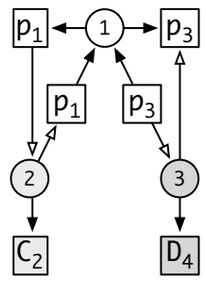
$$\frac{}{D < P.p} \qquad \frac{p < p'}{s.p < s.p'}$$

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

Java Packages

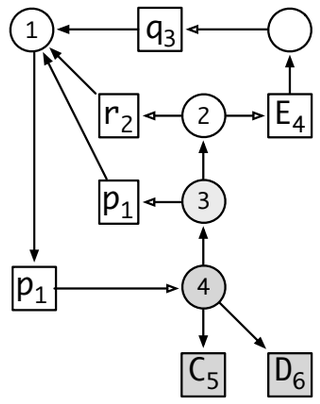
```
package p1;

class C2 {}
```

```
package p3;

class D4 {}
```



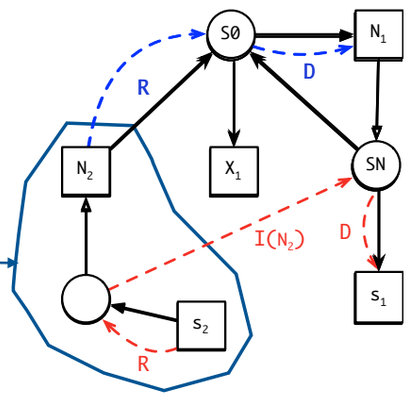
Java Import

```
package p1;

imports r2.*;
imports q3.E4;

public class C5 {}

class D6 {}
```



Qualified Names

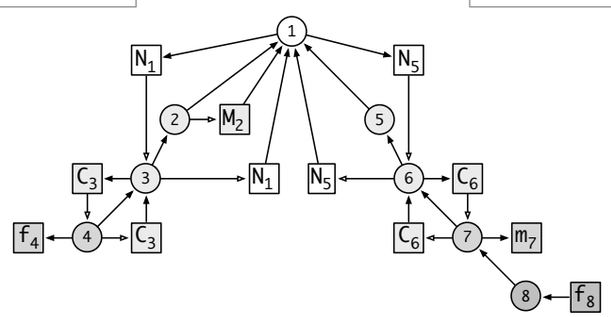
```
module N1 {                S0
    def s1 = 5
}

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



C# Namespaces and Partial Classes

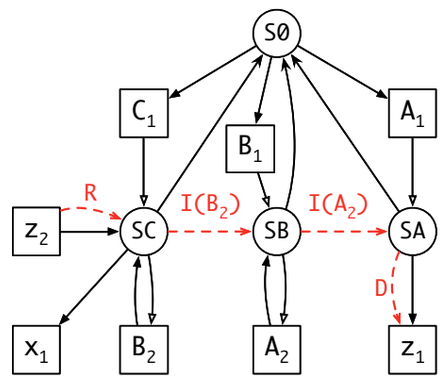
```
namespace N1 {
  using M2;

  partial class C3 {
    int f4;
  }
}
```

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



Transitive vs. Non-Transitive

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



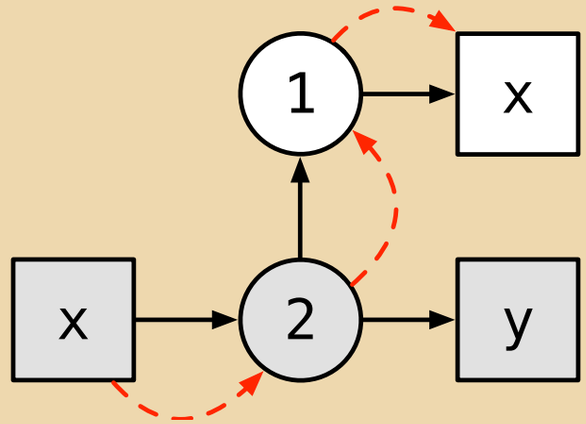
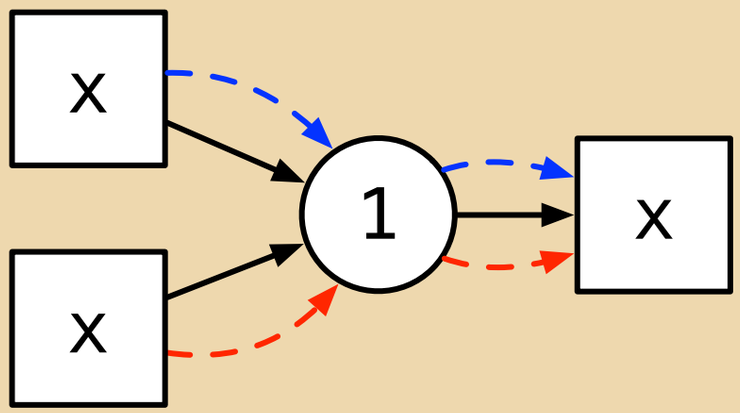
With transitive imports, a well formed path is R.P*.I(_)*.D

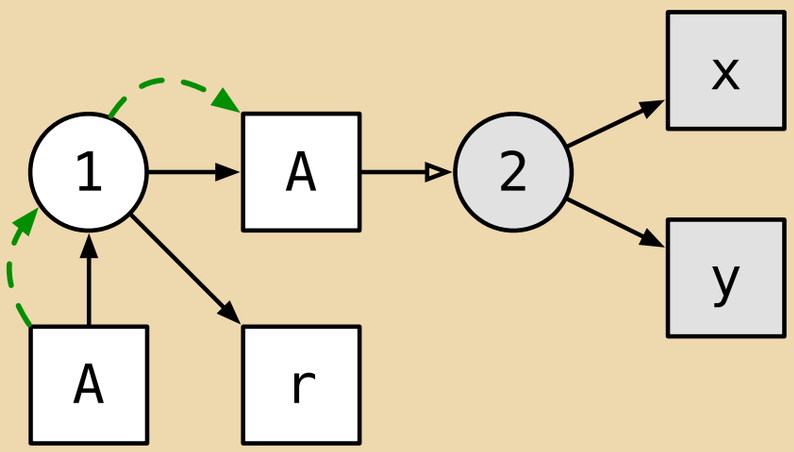
With non-transitive imports, a well formed path is R.P*.I(_)?.D

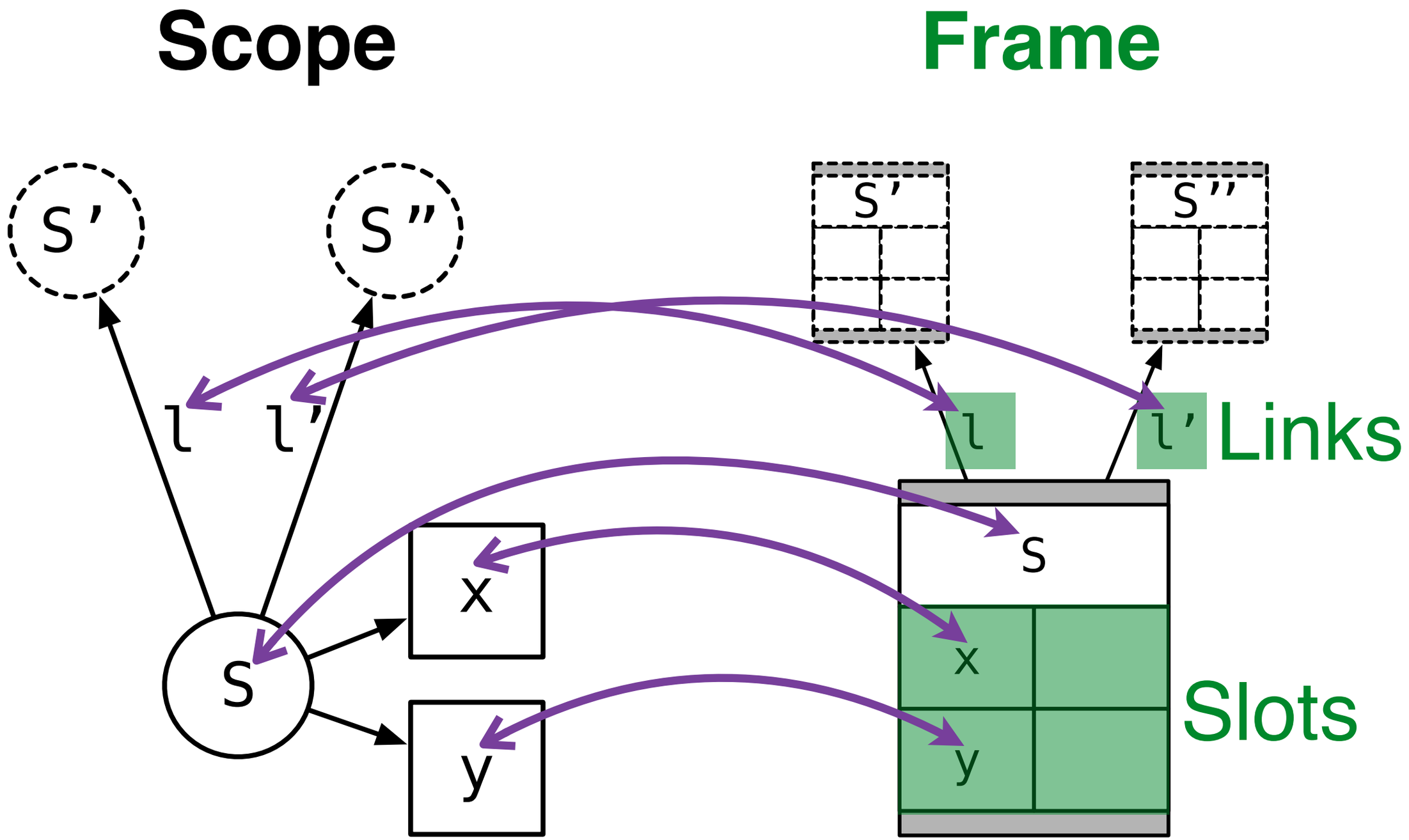
[ESOP'15]

		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

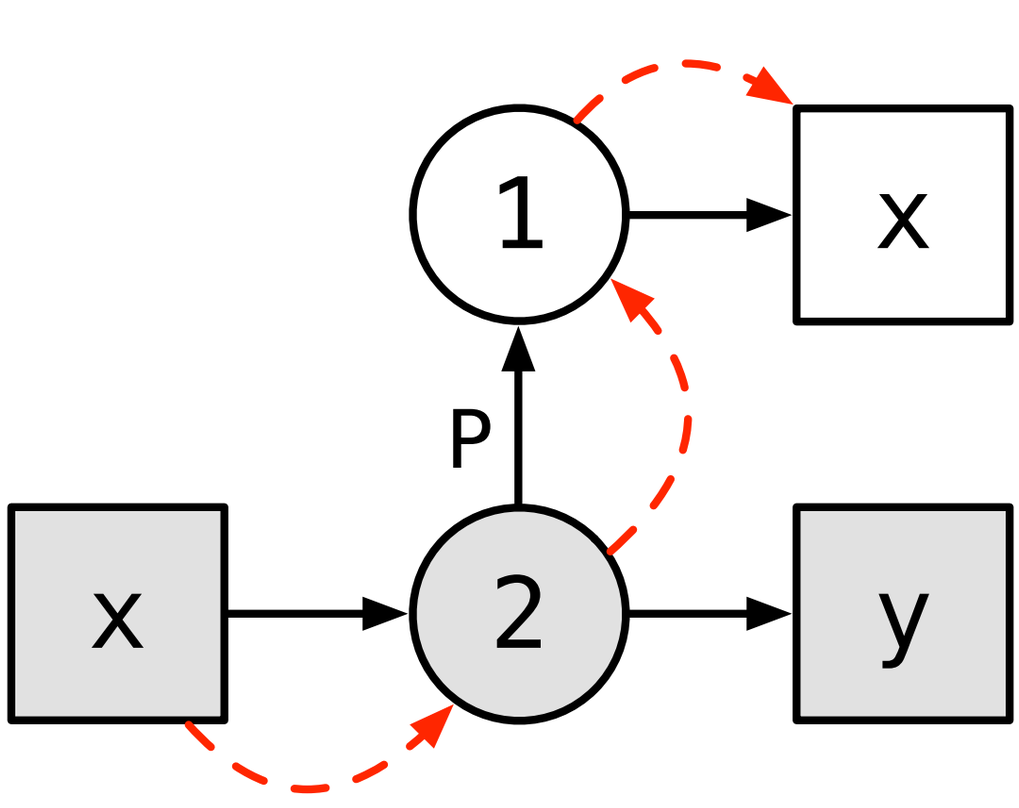
		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`		Stores/Heaps

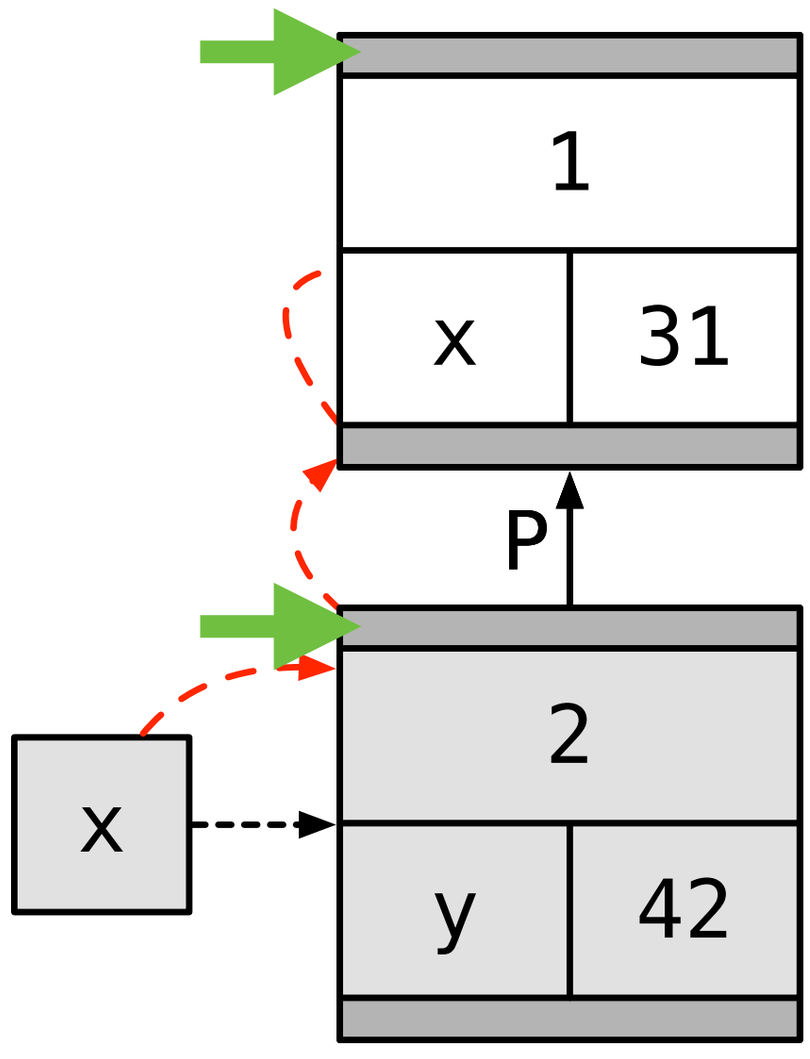
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
			Mutable Objects
			Stores/Heaps



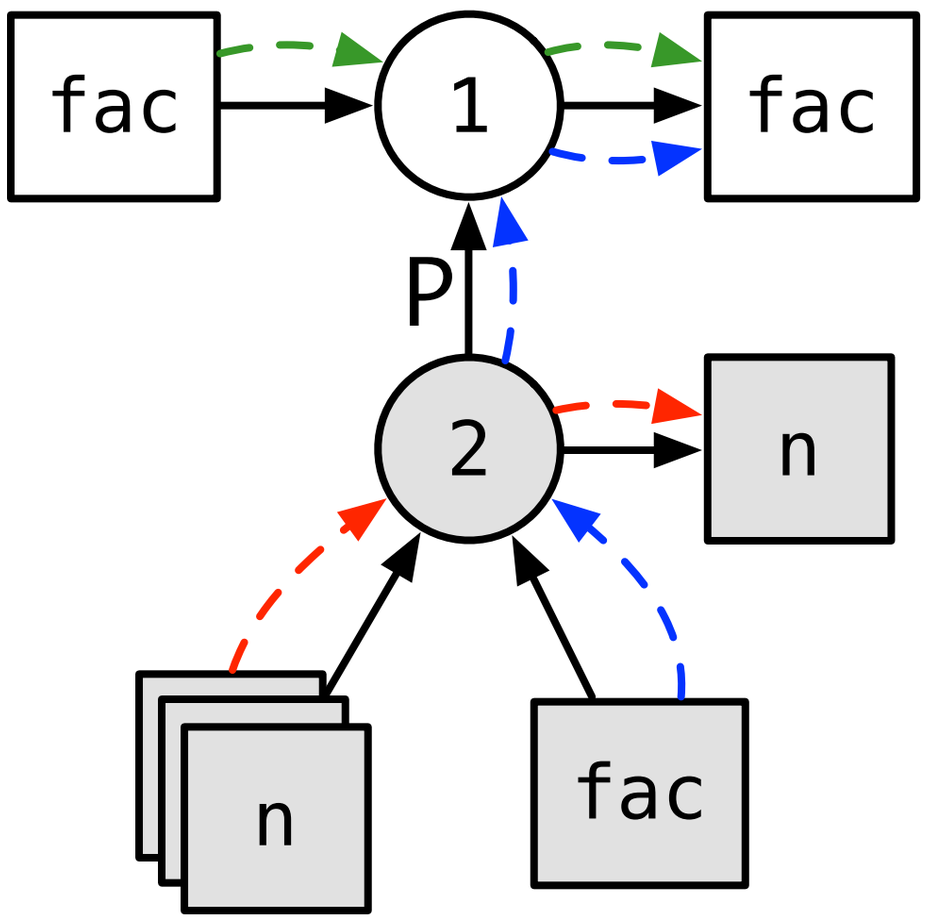
[ECOOP'16]

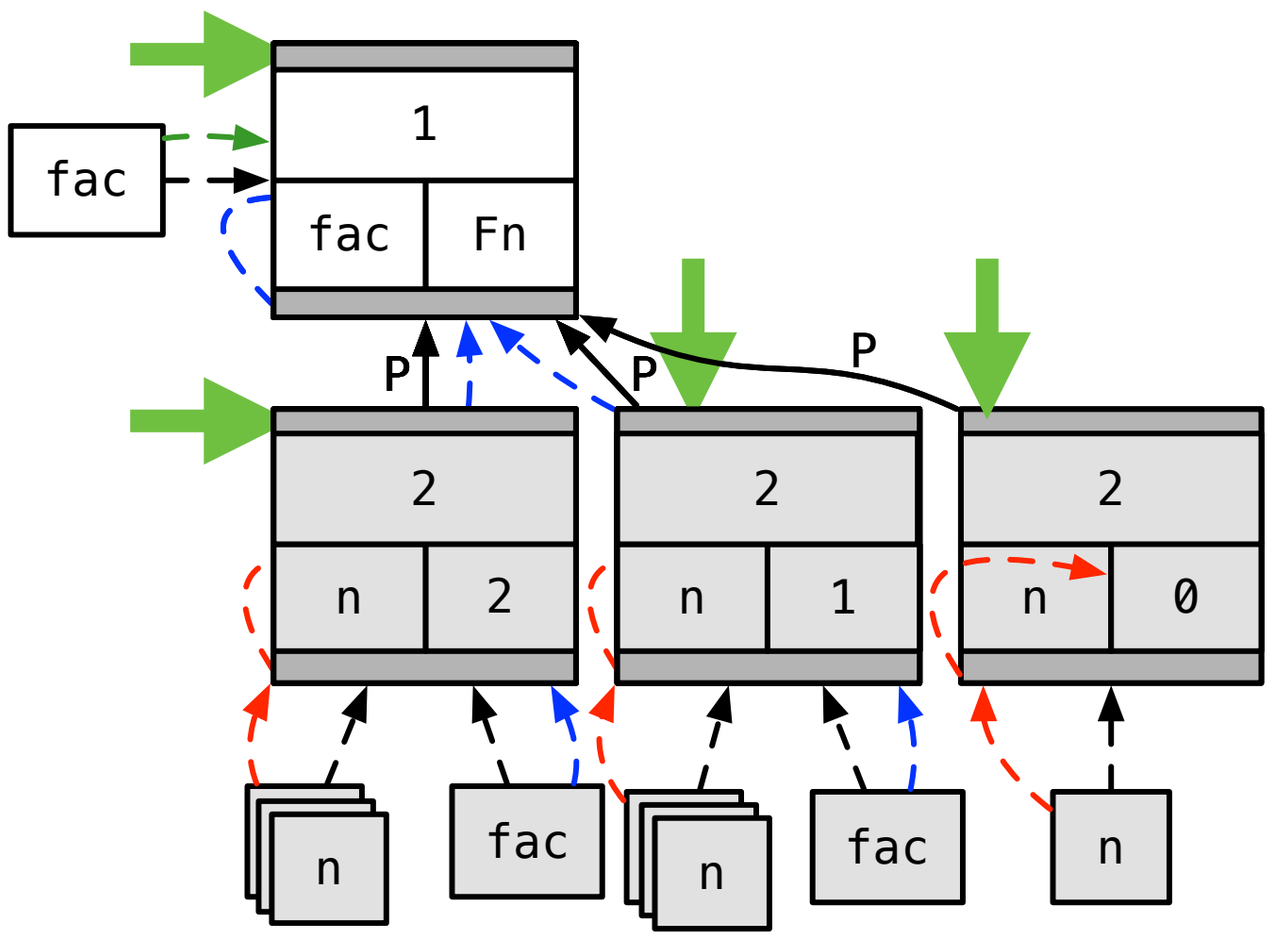
```
val x = 31;
val y = x + 11;
```



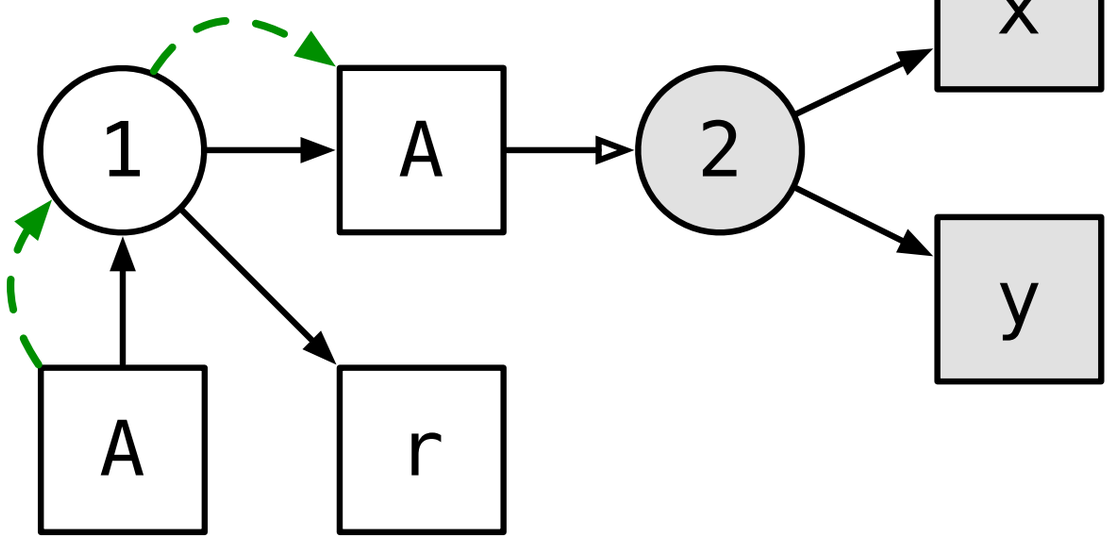


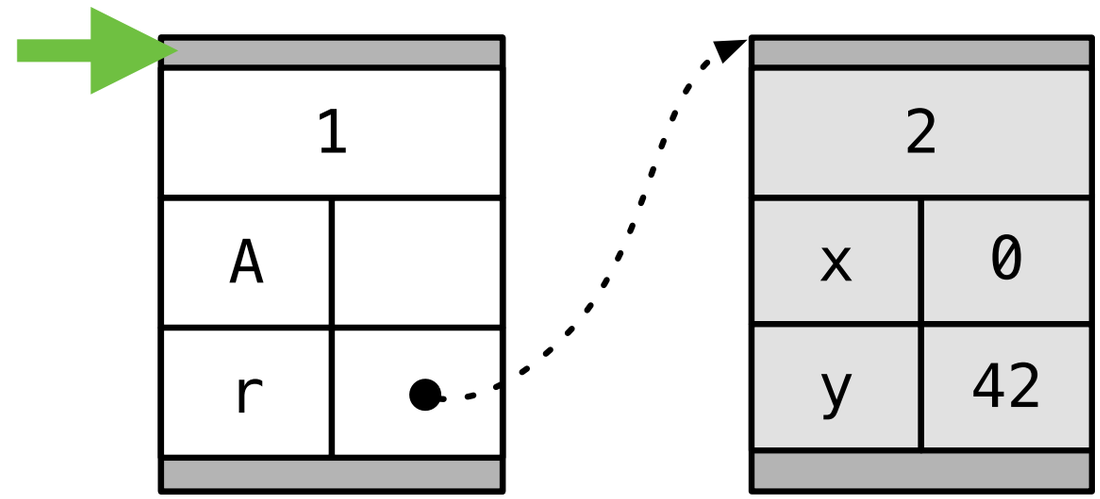
```
def fac(n : Int) : Int = {
  if (n == 0) 1
  else n * fac(n - 1)
};
fac(2);
```

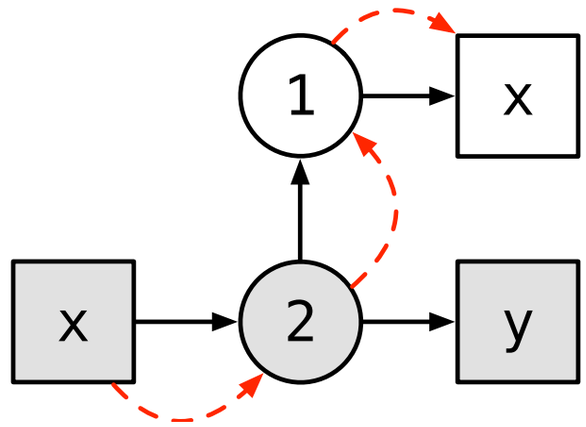
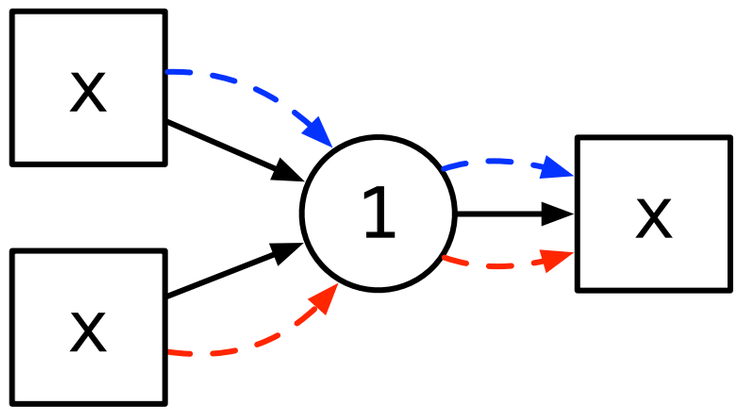


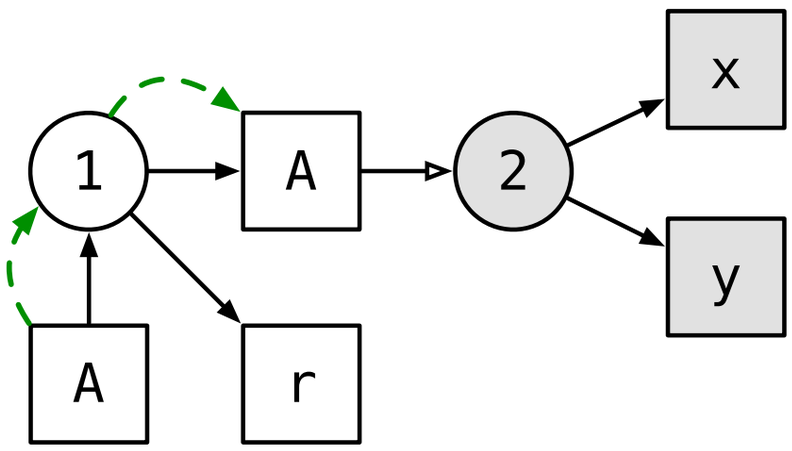


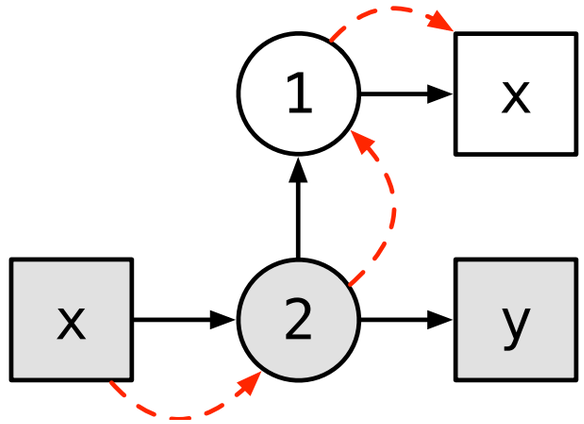
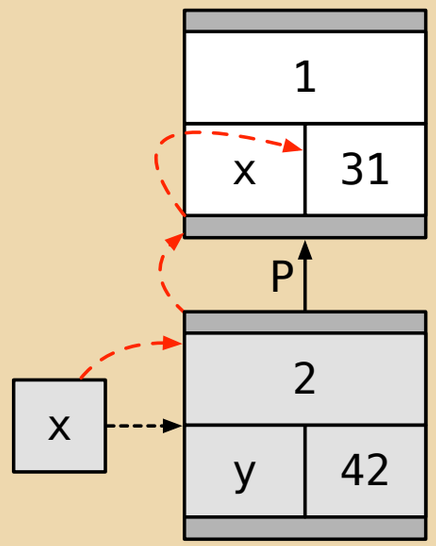
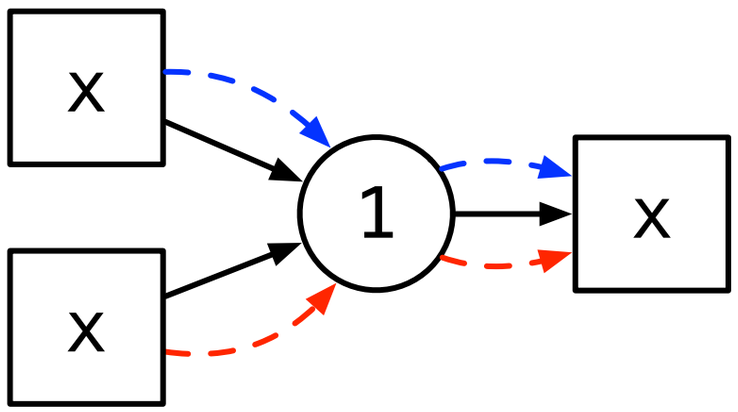
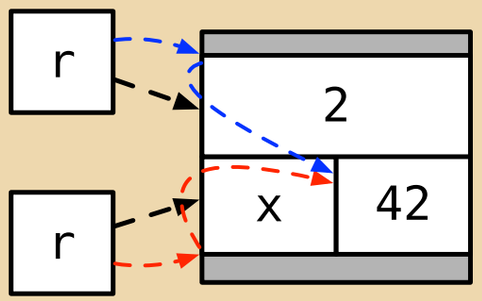
```
class A {
  var x = 0;
  var y = 42;
}
var r = new A();
```

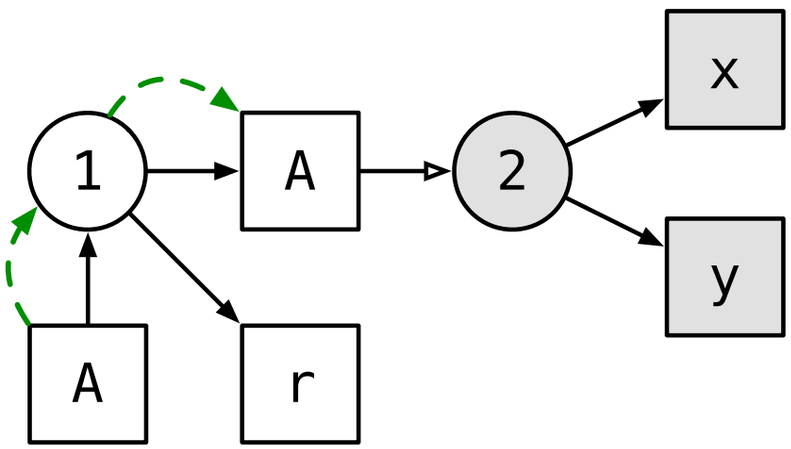
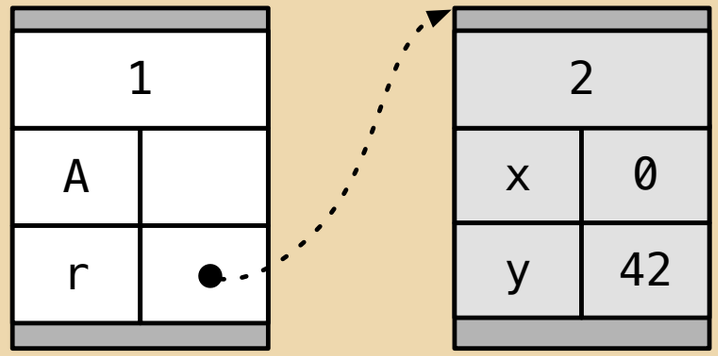




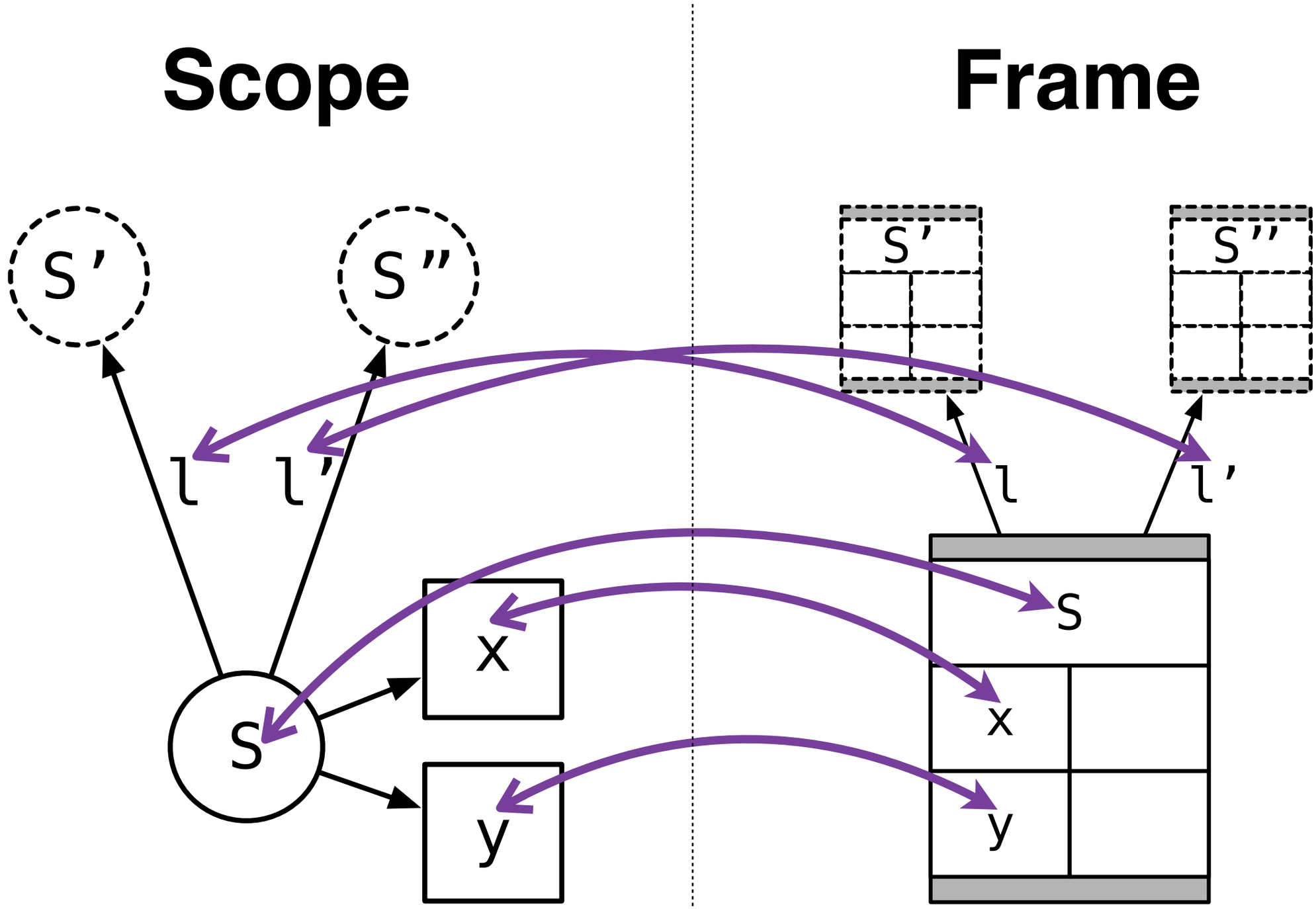
	Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`		Stores/Heaps

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;`	 

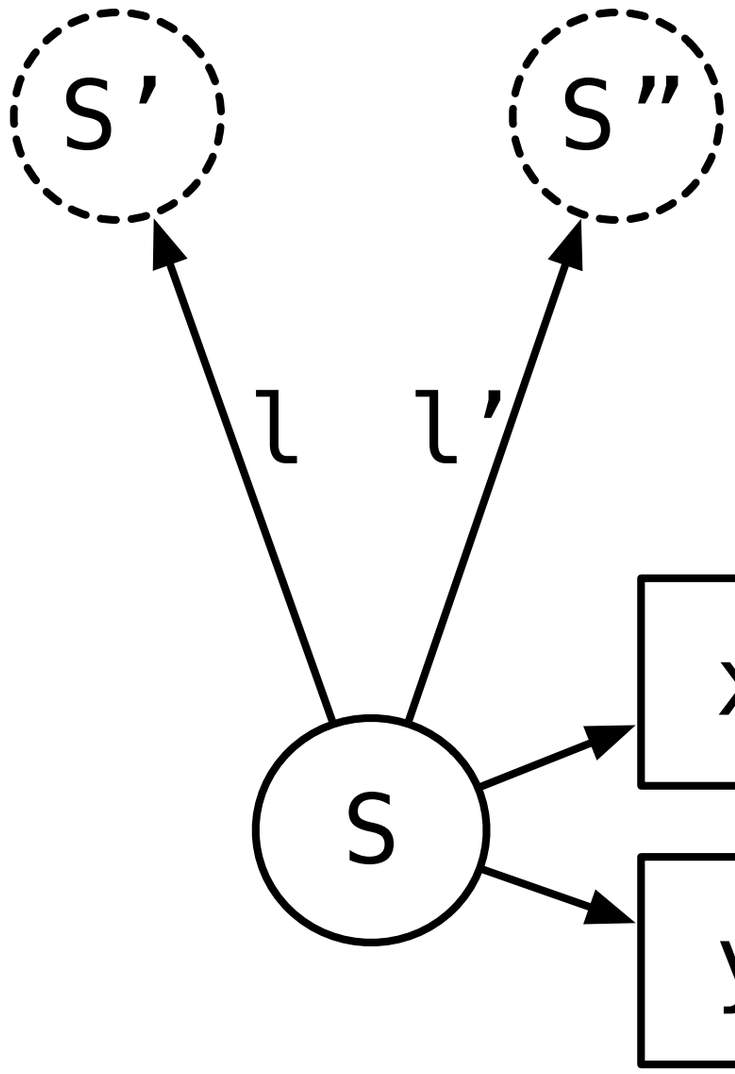
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
		 

Well-Bound Frame

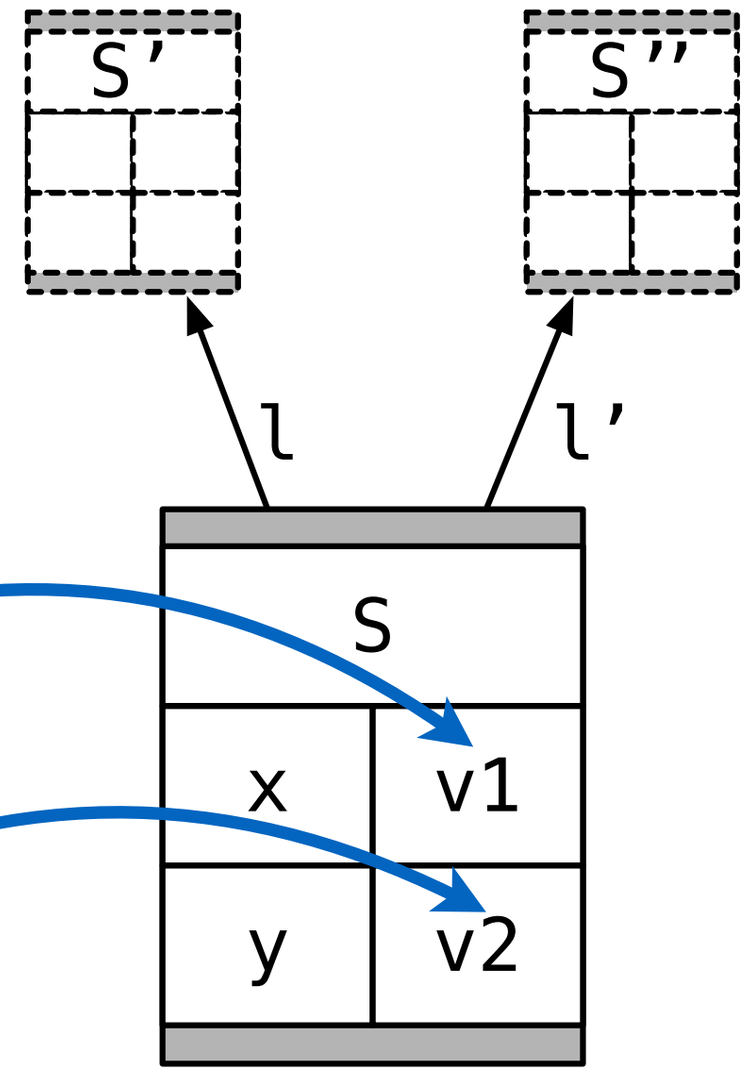


Well-Typed Frame

Scope

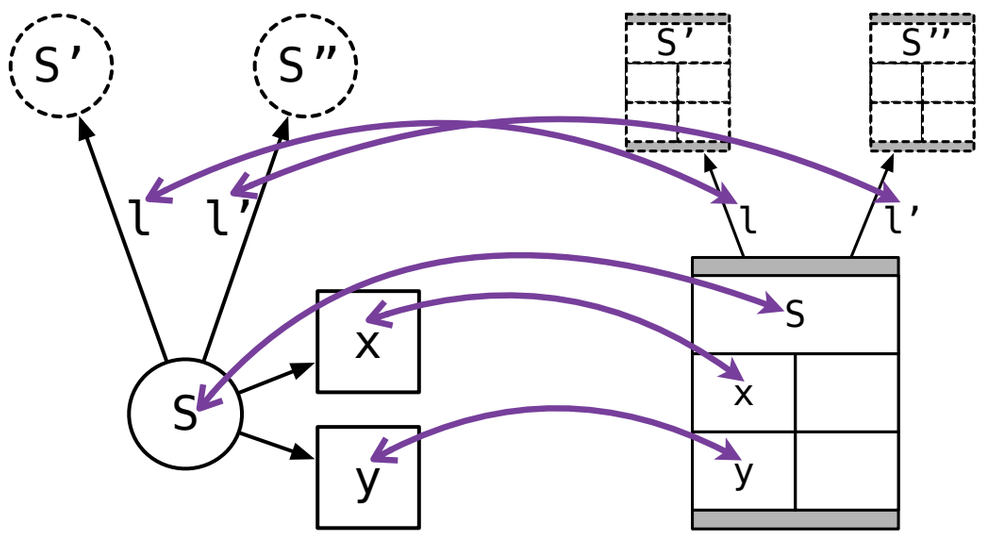


Frame

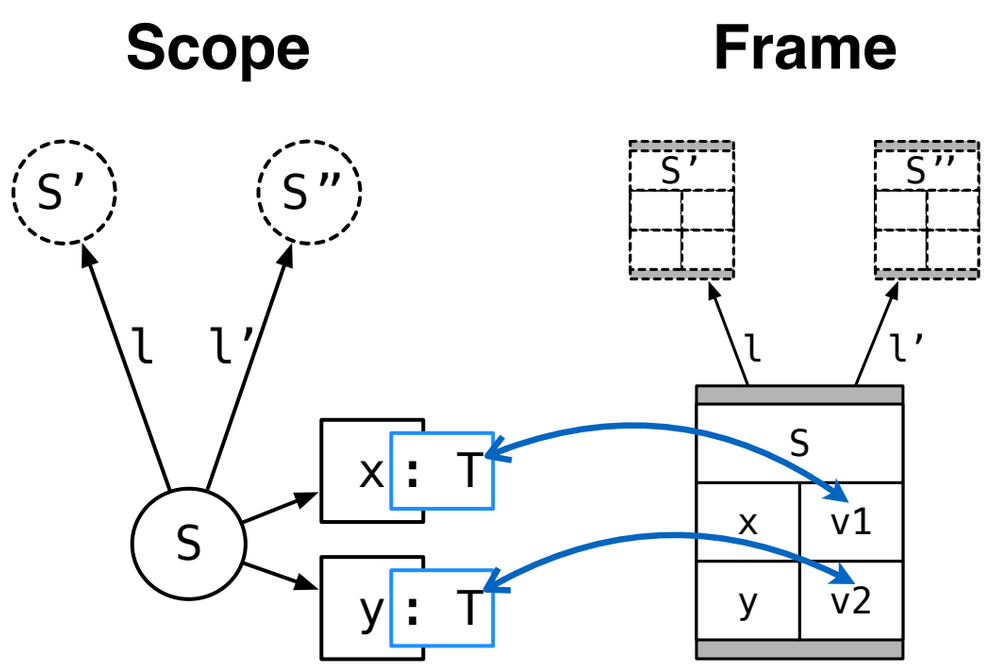


Good Frame Invariant

Well-Bound Frame

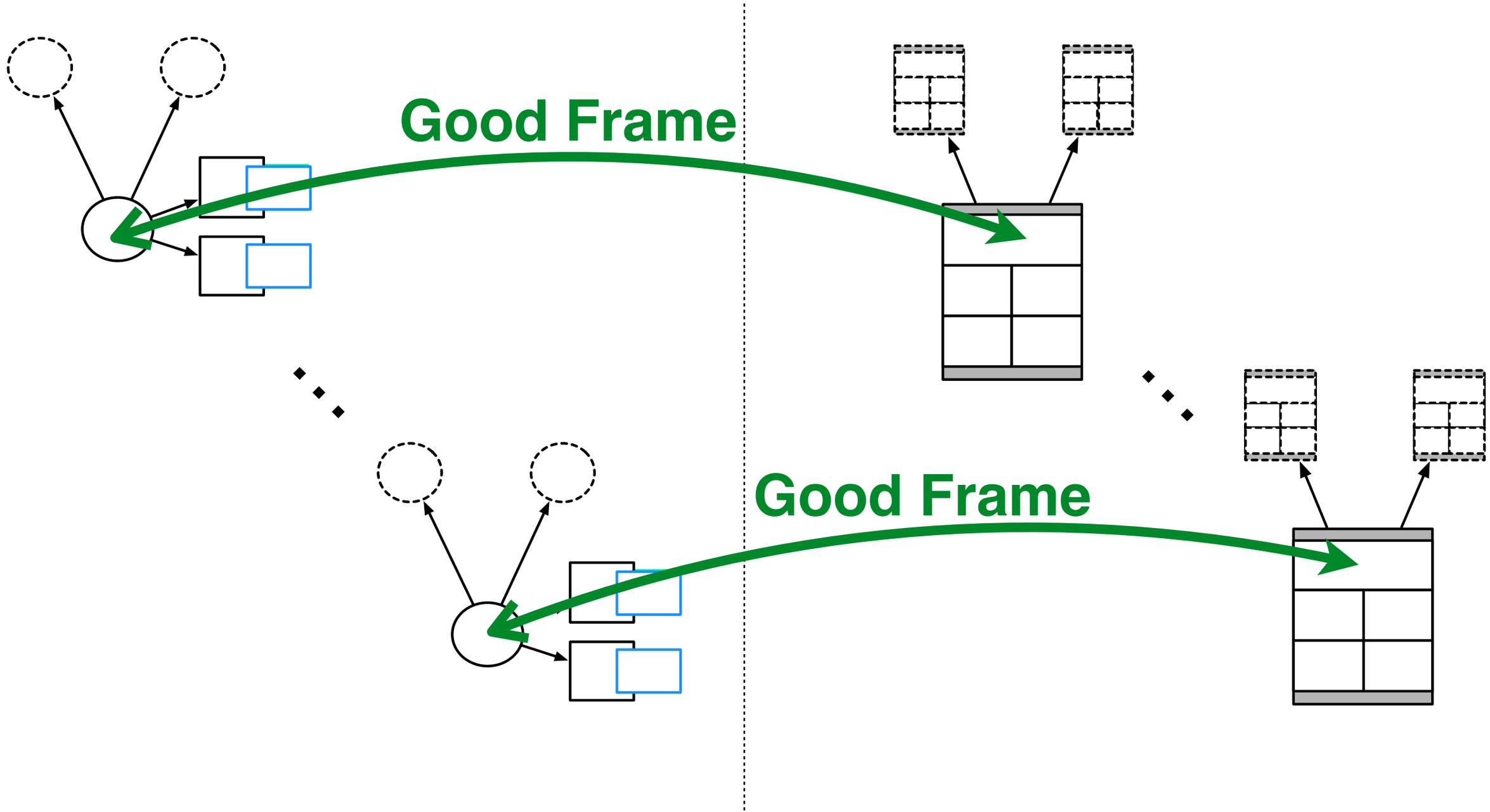


Well-Typed Frame

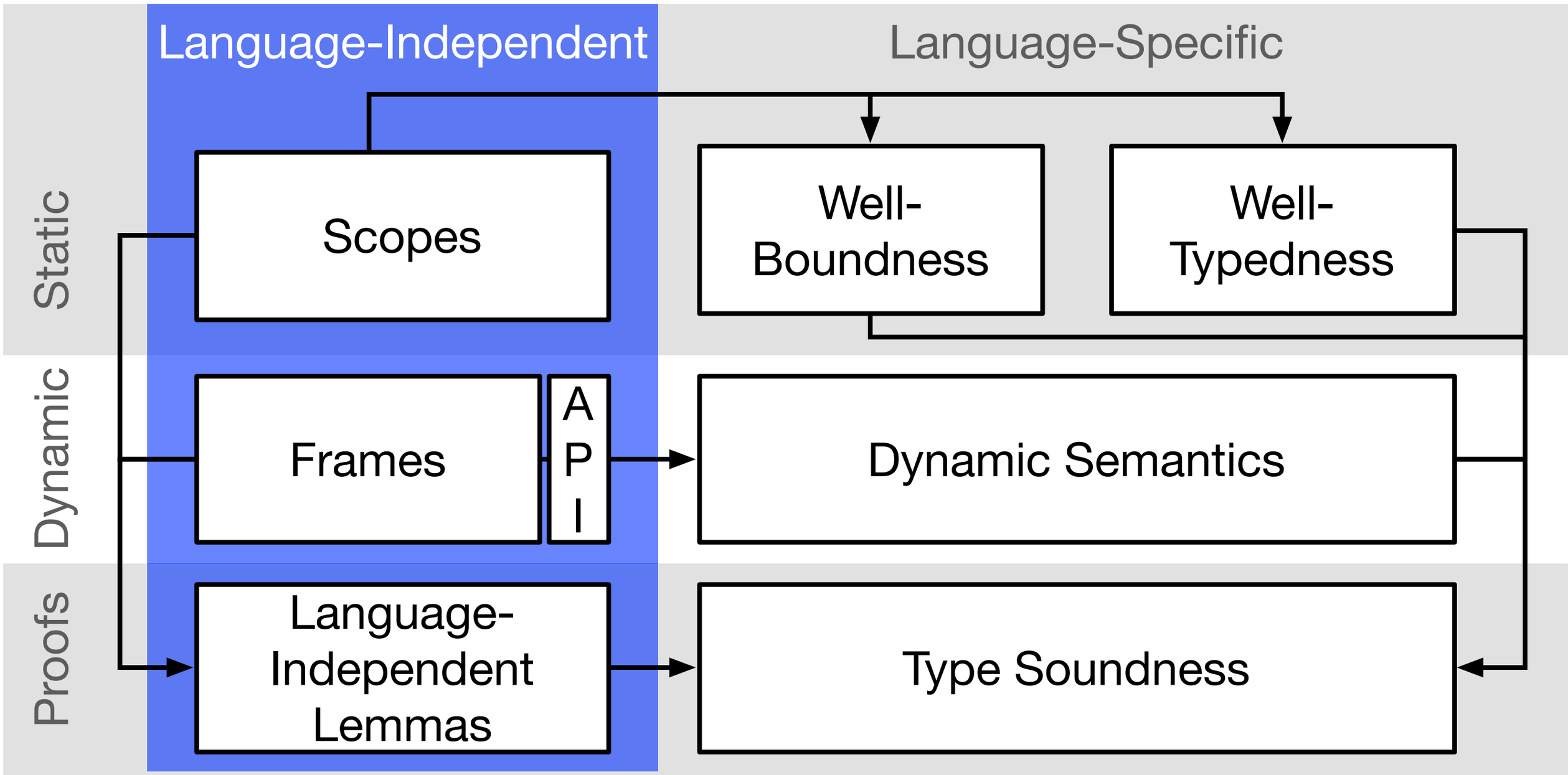


Good Heap Invariant

Every Frame is
Well-Bound and Well-Typed

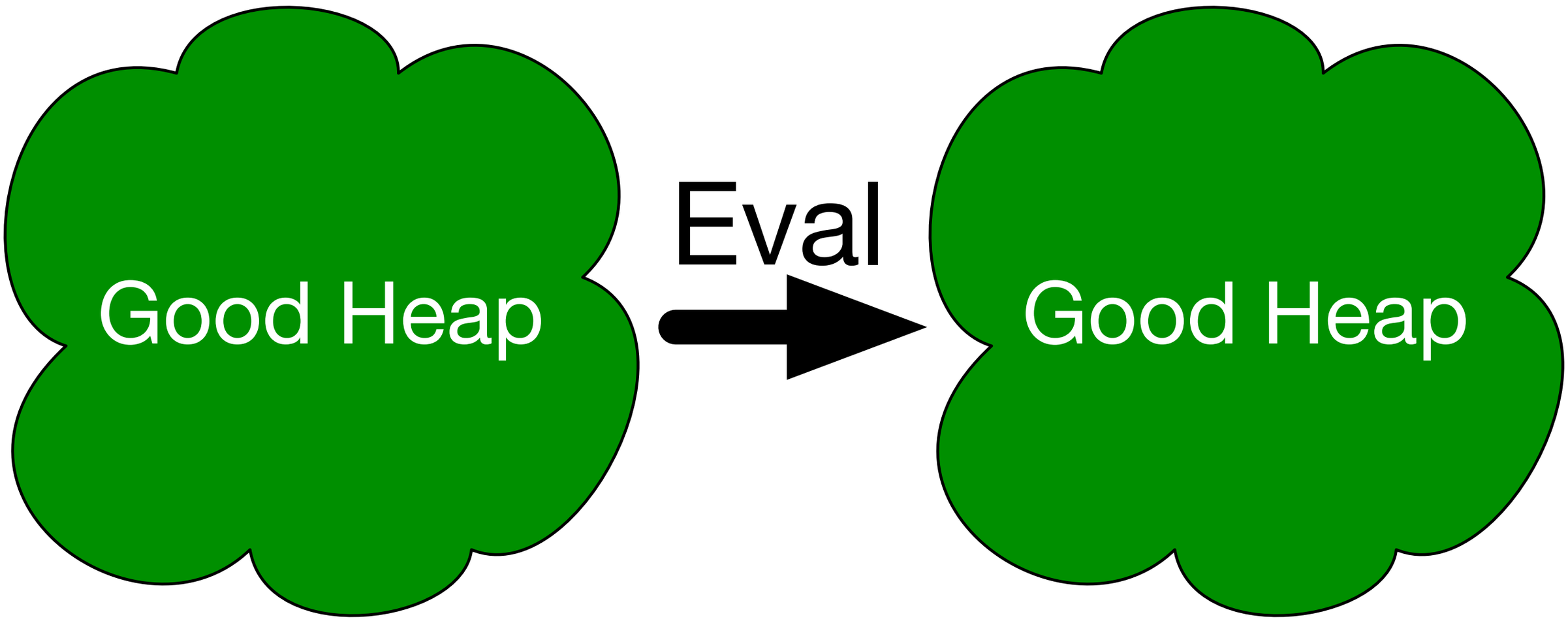


Architecture of a Specification



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

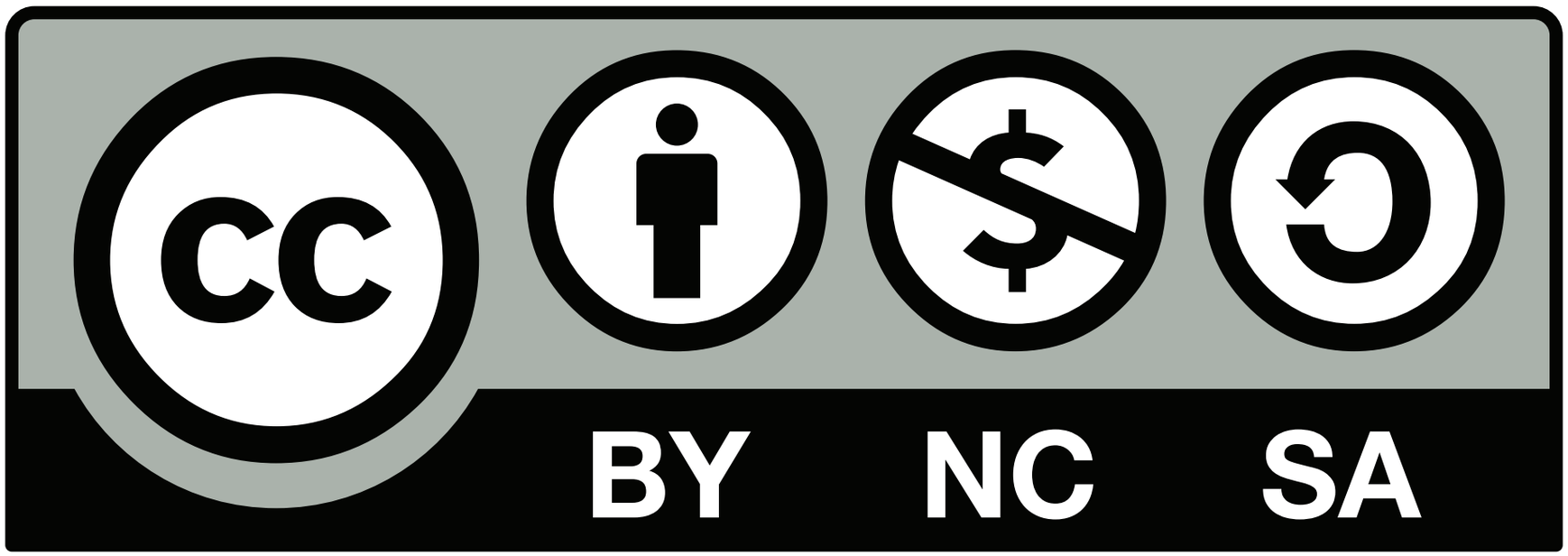
Interpreter Generation

Optimization

Targeting (Graal+Truffle)/PyPy?

Type Soundness Verification

copyrights


BY NC SA

Pictures

copyrights

Slide 1:
arrows by Dean Hochman, some rights reserved