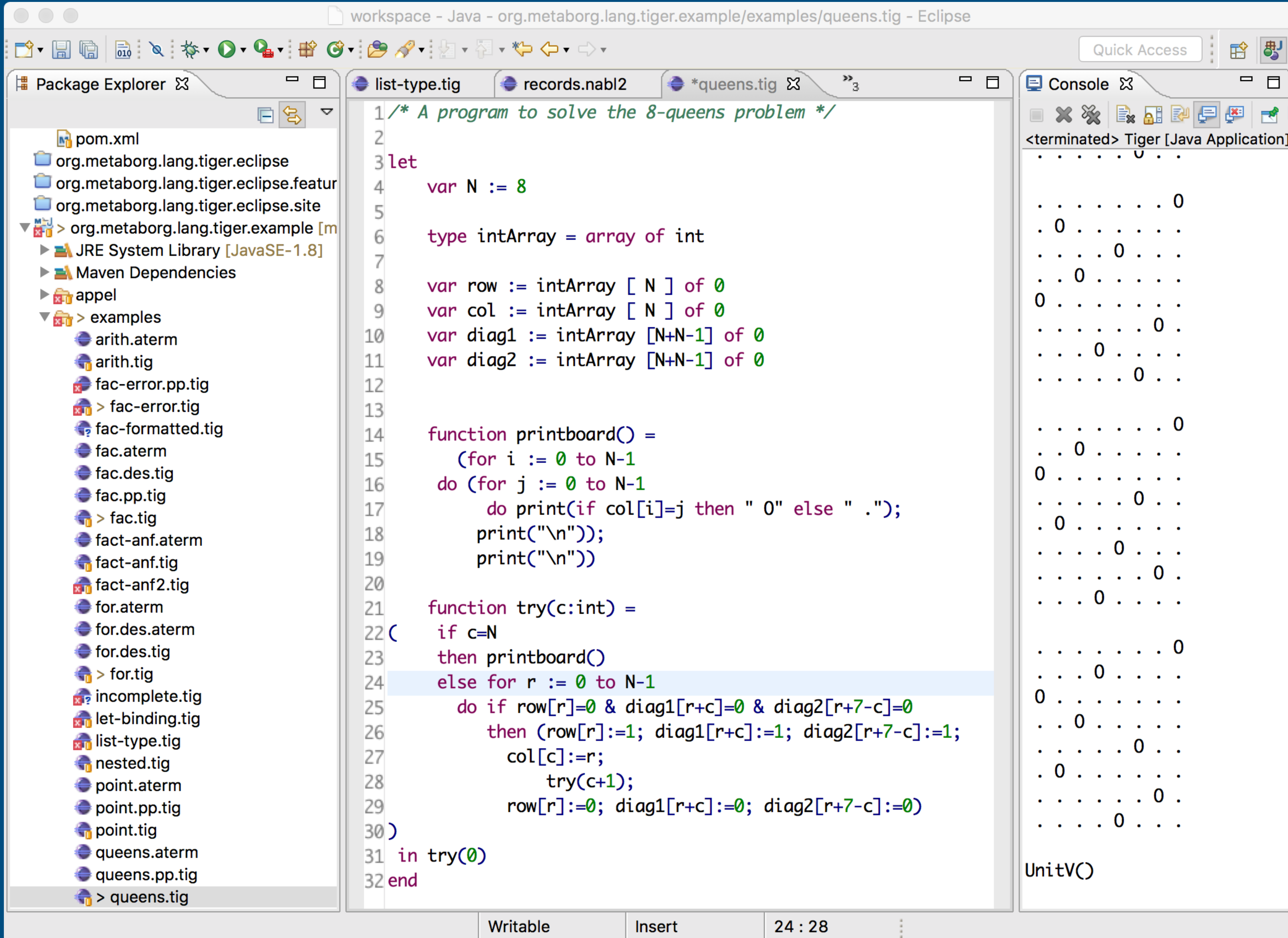
The Design of NaBL2: From Theory (of Name Resolution) to Practice

Eelco Visser

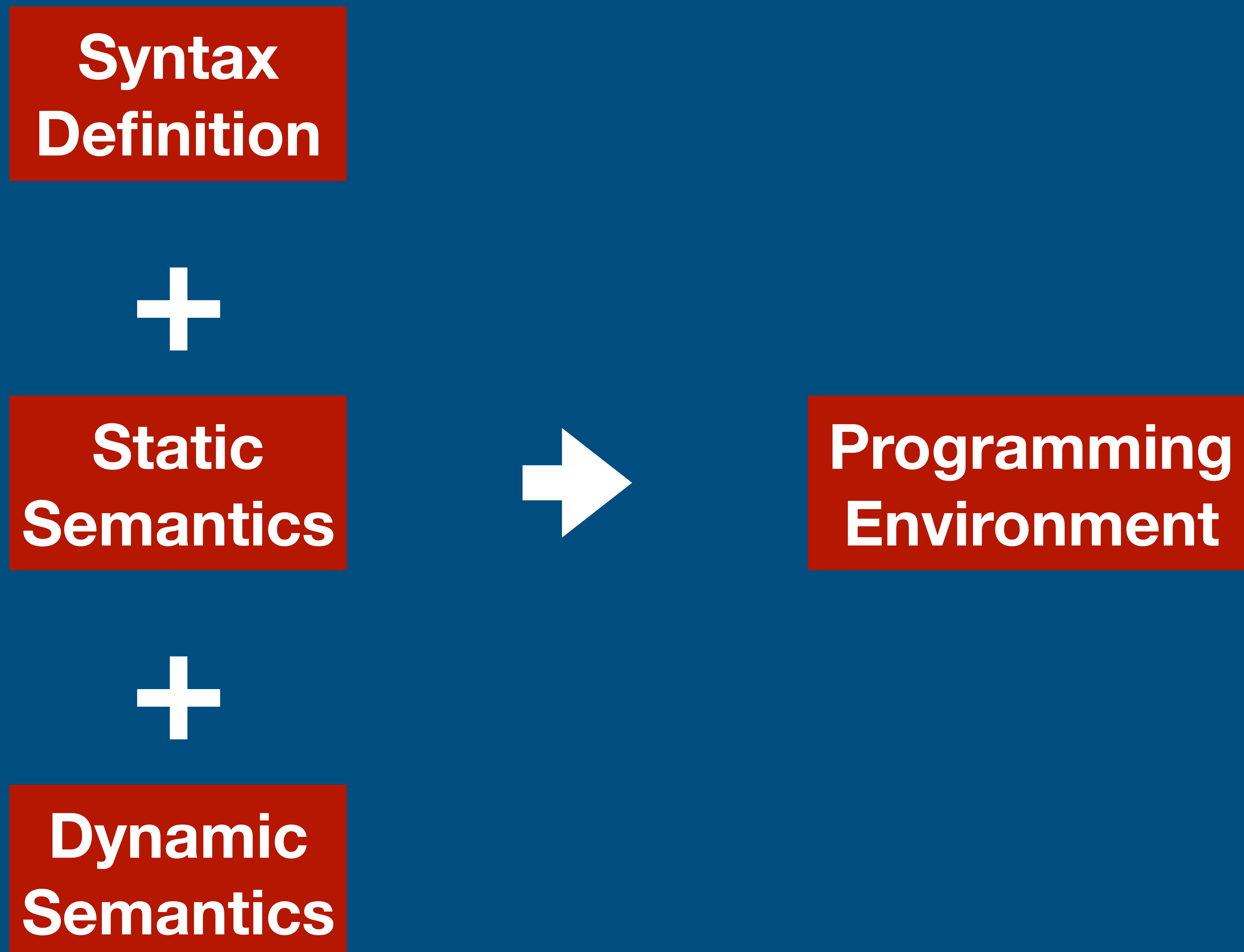
IFIP WG 2.16 Language Design
Park City, Utah
August 2017

TU Delft

Motivation


workspace - Java - org.metaborg.lang.tiger.example/examples/queens.tig - Eclipse
Quick Access
Package Explorer
pom.xml
org.metaborg.lang.tiger.eclipse
org.metaborg.lang.tiger.eclipse.featur
org.metaborg.lang.tiger.eclipse.site
> org.metaborg.lang.tiger.example [m
JRE System Library [JavaSE-1.8]
Maven Dependencies
appel
> examples
arith.aterm
arith.tig
fac-error.pp.tig
> fac-error.tig
fac-formatted.tig
fac.aterm
fac.des.tig
fac.pp.tig
> fac.tig
fact-anf.aterm
fact-anf.tig
fact-anf2.tig
for.aterm
for.des.aterm
for.des.tig
> for.tig
incomplete.tig
let-binding.tig
list-type.tig
nested.tig
point.aterm
point.pp.tig
point.tig
queens.aterm
queens.pp.tig
> queens.tig
list-type.tig
records.nabl2
*queens.tig
/* A program to solve the 8-queens problem */
let
var N := 8
type intArray = array of int
var row := intArray [N] of 0
var col := intArray [N] of 0
var diag1 := intArray [N+N-1] of 0
var diag2 := intArray [N+N-1] of 0
function printboard() =
(for i := 0 to N-1
do (for j := 0 to N-1
do print(if col[i]=j then " O" else " .");
print("\n"));
print("\n"))
function try(c:int) =
(if c=N
then printboard()
else for r := 0 to N-1
do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
col[c]:=r;
try(c+1);
row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)
)
in try(0)
end
Console
<terminated> Tiger [Java Application]
UnitV()
Writable
Insert
24 : 28

Language Workbench



Spoofax Language Workbench

SDF3: Syntax Definition + **NaBL2: Static Semantics** + **DynSem: Dynamic Semantics** → **Programming Environment**

Separation of Concerns

Representation

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

Specification Formalism

- Language-specific declarative rules
- Abstract from implementation concerns

Language-Independent Interpretation

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

Separation of Concerns in Syntax Definition

Representation: (Abstract Syntax) Trees

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

Formalism: Syntax Definition

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

A meta-language for talking about syntax

Language-Independent Interpretation

- Well-formedness of abstract syntax trees
  - provides declarative correctness criterion for parsing
- Parsing algorithm
  - No need to understand parsing algorithm
  - Debugging in terms of representation
- Formatting based on layout hints in grammar
- Syntactic completion

Separation of Concerns in Name Binding

Representation

- To conduct and represent the results of name resolution

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

NaBL: Name Binding Language

Konat, Kats, Wachsmuth, Visser. Declarative Name Binding and Scope Rules. SLE 2012

Interaction with Type System (1)

```
FieldAccess(e, f):
  refers to Field f in c
  where e has type ClassType(c)
```

```
class C {
  int i;
}

class D {
  C c;
  int i;
  void init() {
    i = c.i;
  }
}
```

Name Binding

`defines`	class partial class type inheritance
`refers`	
`namespaces`	namespace using
`scopes`	method field variable parameter block
`imports`	

Theory: Scope Graphs

Néron, Tolmach, Visser, Wachsmuth. A Theory of Name Resolution. ESOP 2015

Separation of Concerns in Name Binding

Representation

- **Scope Graphs**

Declarative Rules

- To define name binding rules of a language

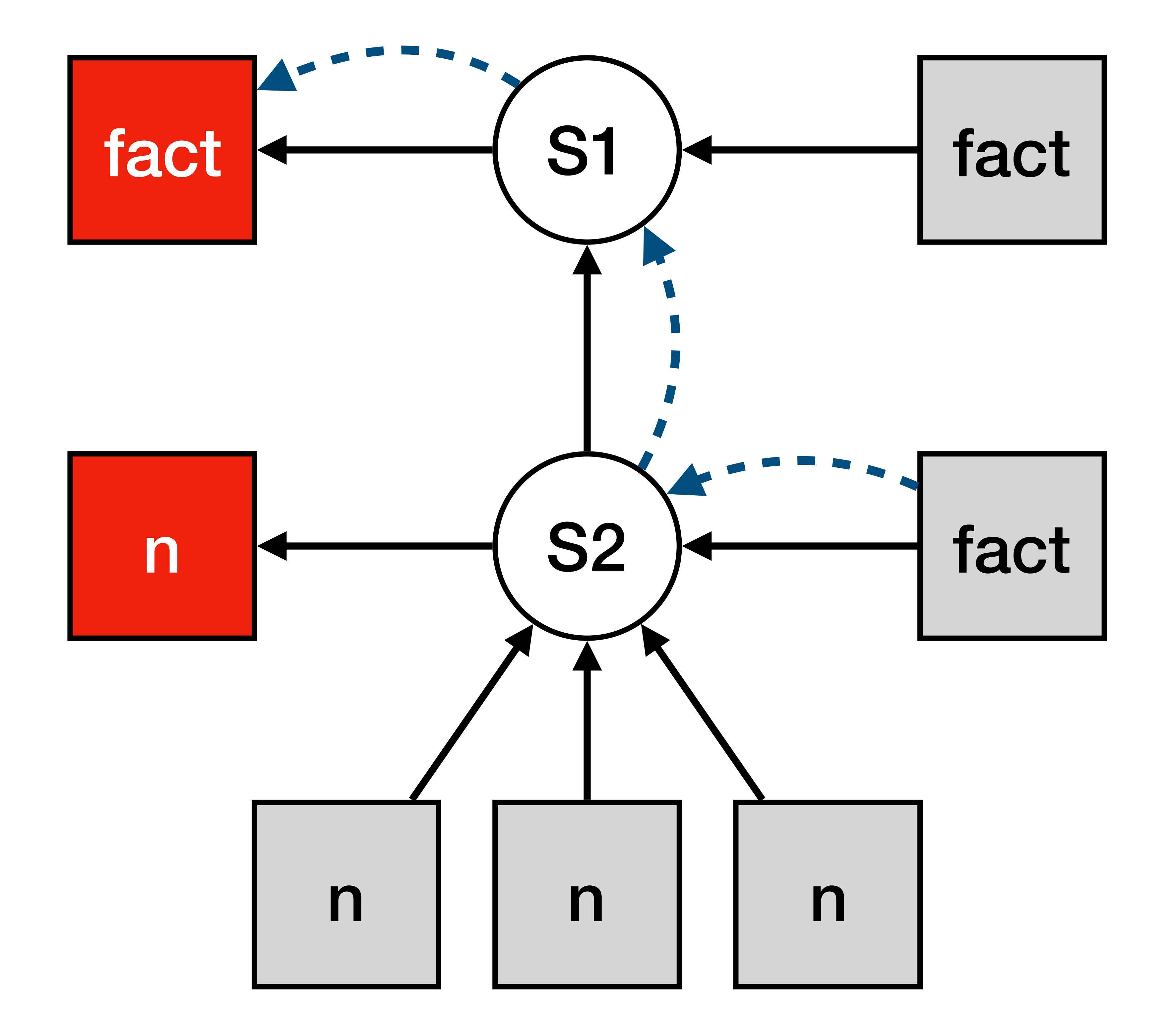
Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Program

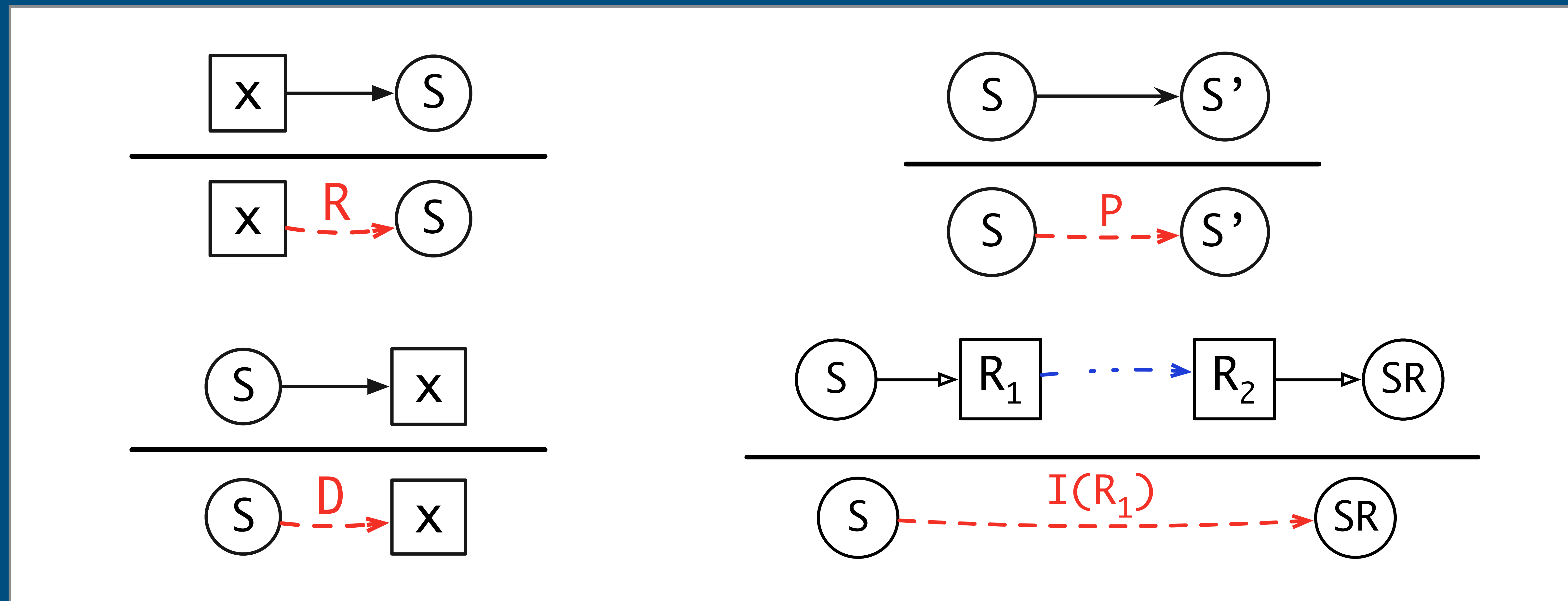
```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph



Name Resolution

A Calculus for Name Resolution



Reachability

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

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

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

Visibility

Visibility Policies

Lexical scope

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

Non-transitive imports

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

Transitive imports

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

Transitive Includes

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

Transitive includes and imports, and non-transitive imports

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

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

Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- To define name binding rules of a language

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- Scope (& Type) Constraint Rules

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

Theory: Constraints

Van Antwerpen, Néron, Tolmach, Visser, Wachsmuth.
A constraint language for static semantic analysis
based on scope graphs. PEPM 2016

NaBL2

Architecture

Scope Graph Constraints

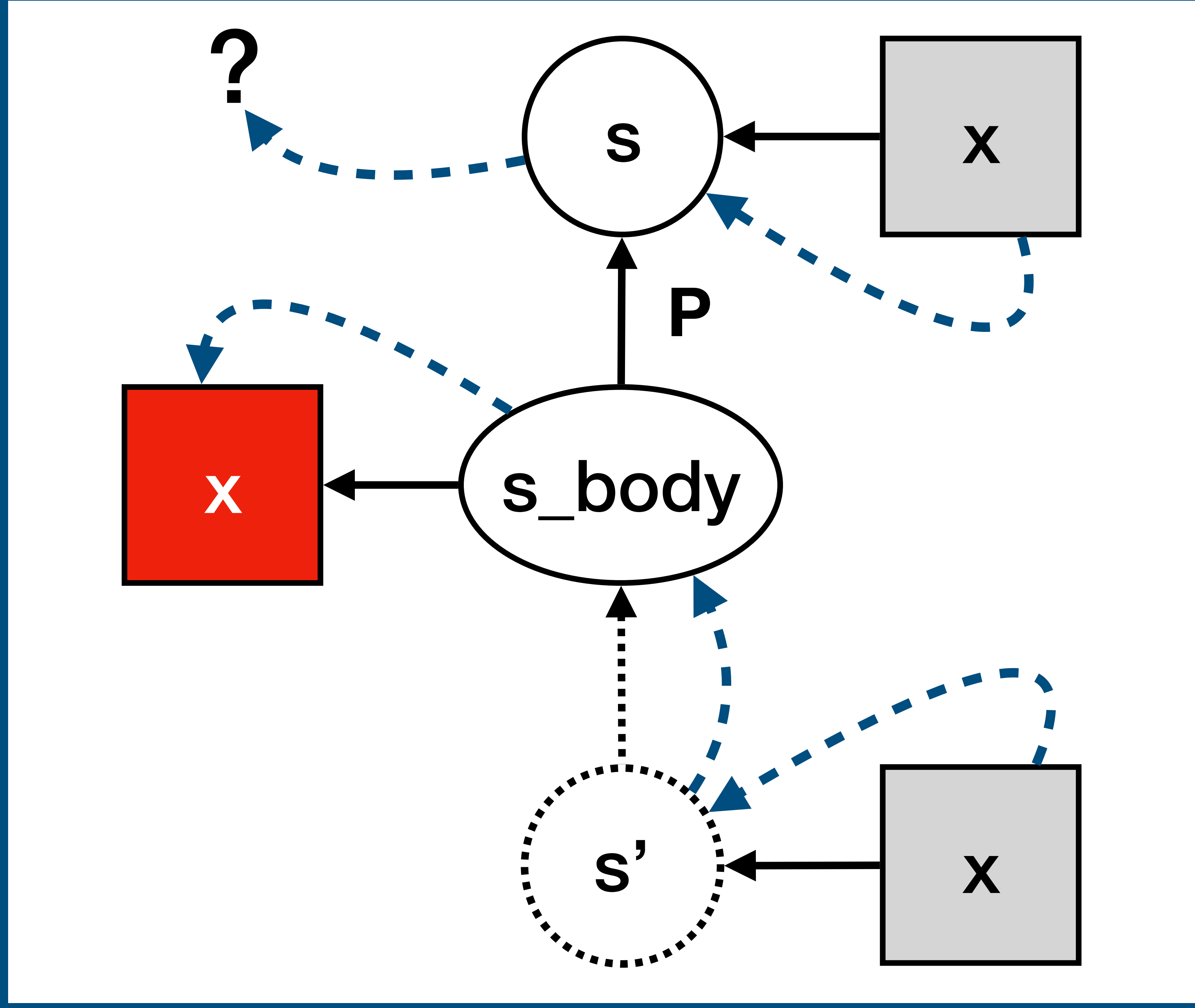
```
new s            // new scope

s1 -L-> s2       // labeled edge from scope s1 to scope s2

N{x} <- s        // x is a declaration in scope s for namespace N

N{x} -> s        // x is a reference in scope s for namespace N

N{x} |-> d       // x resolves to declaration d

[[ e ^ (s) ]]    // constraints for expression e in scope s
```

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

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

```
[[ Let([VarDec(x, t, e)], [e_body]) ^ (s) ]] :=
   new s_body,              // new scope
   s_body -P-> s,           // parent edge to enclosing scope
   Var{x} <- s_body,        // x is a declaration in s_body
   [[ e ^ (s) ]],           // init expression
   [[ e_body ^ (s_body) ]]. // body expression
```

```
[[ Var(x) ^ (s') ]] :=
   Var{x} -> s', // x is a reference in s'
   Var{x} |-> d, // check that x resolves to a declaration
```



Type Constraints

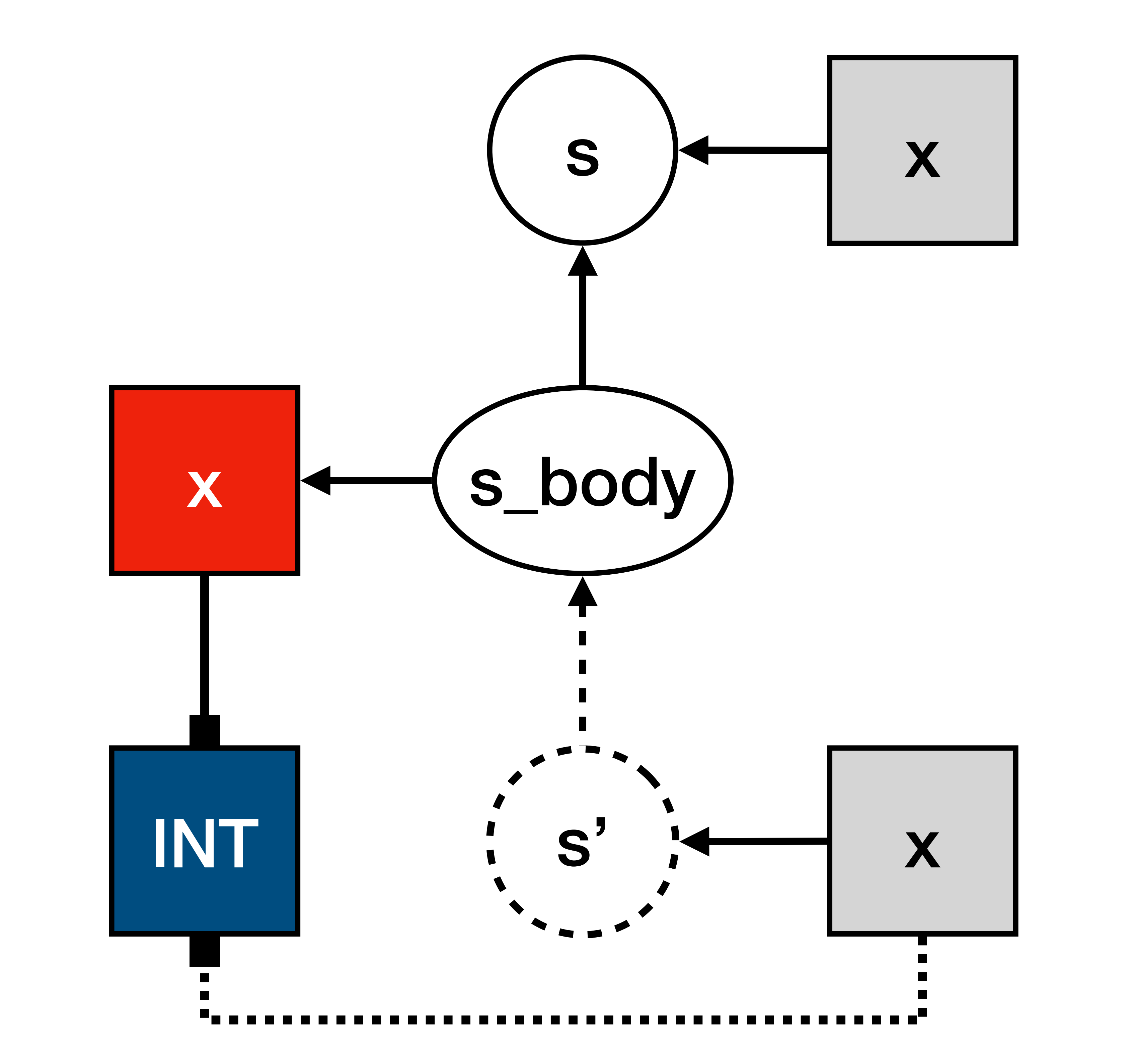
```
d : ty                  // declaration has type

t1 == ty2               // type equality

ty1 <! ty2              // declare sub-type

ty1 <? ty2              // query sub-type

ty_gen genOf ty         // generalization

ty_gen instOf ty        // instantiation

. . .                   // extensions

[[ e ^ (s) : ty ]]      // type of expression in scope
```

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

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

```
[[ Let([VarDec(x, t, e)], [e_body]) ^ (s) : ty' ]] :=
   new s_body,                    // new scope
   s_body -P-> s,                 // parent edge to enclosing scope
   Var{x} <- s_body,              // x is a declaration in s_body
   Var{x} : ty,                   // associate type
   [[ t ^ (s) : ty ]],            // type of type
   [[ e ^ (s) : ty ]],            // type of expression
   [[ e_body ^ (s_body) : ty' ]]. // constraints for body
```

```
[[ Var(x) ^ (s') : ty ]] :=
   Var{x} -> s', // x is a reference in s'
   Var{x} |-> d, // check that x resolves to a declaration
   d : ty.       // type of declaration is type of reference
```



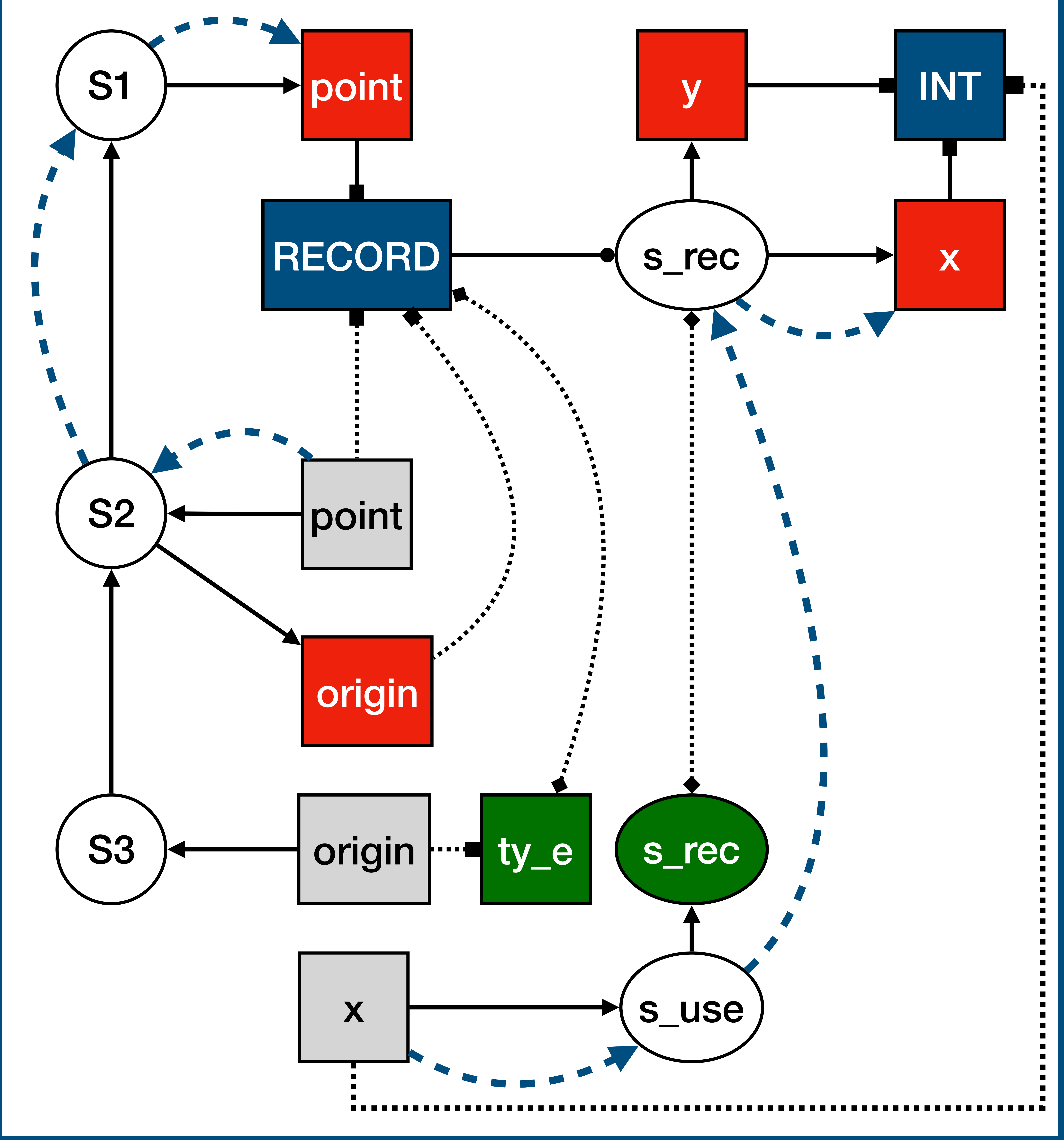
```
let
  type point = {x : int, y : int}
  var origin : point := …
 in origin.x
end
```

```
[[ RecordTy(fields) ^ (s) : ty ]] :=
   ty == RECORD(s_rec),
   new s_rec,
   Map2[[ fields ^ (s_rec, s) ]].
```

```
[[ Field(x, t) ^ (s_rec, s_outer) ]] :=
   Field{x} <- s_rec,
   Field{x} : ty !,
   [[ t ^ (s_outer) : ty ]].
```

```
[[ FieldVar(e, f) ^ (s) : ty ]] :=
   [[ e ^ (s) : ty_e ]],
   new s_use,
   Field{f} -> s_use,
   s_use -I-> s_rec,
   ty_e == RECORD(s_rec),
   Field{f} |-> d,
   d : ty.
```

Type Dependent Name Resolution



Polymorphic Types

```
rules // numbers

 Stat[[ Bind(x, e) ^ (s, s_tmp) ]] :=
    s_tmp == s_nxt,
    new s_nxt, s_nxt ---> s,
    {x} <- s_nxt, {x} : ty_gen,
    ty_gen genOf ty,
    [[ e ^ (s) : ty ]].

  [[ App(e1, e2) ^ (s) : ty_res ]] :=
     [[ e1 ^ (s) : ty_fun ]],
     [[ e2 ^ (s) : ty_arg ]],
     FunT(ty_arg, ty_res) instOf ty_fun.
```

Name Constraints

```
// Name constraints
e1 subseteq/proj e2 // inclusion
distinct e          // names occur at most once
// Set expressions
0            // empty set
R(s)/NS      // references in scope s
D(s)/NS      // declarations in scope s
V(s)/NS      // visible names in scope s
(e1 union e2)
(e1 isect e2)
(e1 minus e2)
```

Name Constraints in Definition of Records (1)

```
rules // record type

  [[ RecordTy(fields) ^ (s) : ty ]] :=
     new s_rec,
     ty == RECORD(s_rec),
     NIL() <! ty,
     distinct/name D(s_rec)/Field | error $[Duplicate declaration of field [NAME]] @ NAMES,
     Map2[[ fields ^ (s_rec, s) ]].

  [[ Field(x, t) ^ (s_rec, s_outer) ]] :=
     Field{x} <- s_rec,
     Field{x} : ty !,
     [[ t ^ (s_outer) : ty ]].
```

Name Constraints in Definition of Records (2)

```
rules // record creation

  [[ r@Record(t, inits) ^ (s) : ty ]] :=
     [[ t ^ (s) : ty ]],
     ty == RECORD(s_rec) | error $[record type expected],
     new s_use,
     s_use -I-> s_rec,
     D(s_rec)/Field subseteq/name R(s_use)/Field | error $[Field [NAME] not initialized] @r,
     distinct/name R(s_use)/Field | error $[Duplicate initialization of field [NAME]] @NAMES,
     Map2[[ inits ^ (s_use, s) ]].

  [[ InitField(x, e) ^ (s_use, s) ]] :=
     Field{x} -> s_use,
     Field{x} |-> d,
     d : ty1,
     [[ e ^ (s) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch got [ty2] where [ty1] expected].
```

Tiger Names & Types: Composition

```
module statics/tiger

imports statics/arrays
imports statics/base
imports statics/bindings
imports statics/control-flow
imports statics/functions
imports statics/nabl-lib
imports statics/numbers
imports statics/records
imports statics/strings
imports statics/types
imports statics/variables

rules // top-level module

  [[ Mod(e) ^ (s) : ty ]] :=
     [[ e ^ (s) : ty ]].
```

Tiger Names & Types: Composition

```
module statics/functions

imports signatures/Functions-sig
imports statics/nabl-lib
imports statics/base

rules // function declarations

  Dec[[ FunDecs(fdecs) ^ (s, s_outer) ]] :=
     Map2[[ fdecs ^ (s, s_outer) ]].

  [[ FunDec(f, args, t, e) ^ (s, s_outer) ]] :=
     new s_fun,
     s_fun -P-> s,
     distinct/name D(s_fun) | error $[duplicate argument] @ NAMES,
     MapTs2[[ args ^ (s_fun, s_outer) : tys ]],
     [[ t ^ (s_outer) : ty ]],
     Var{f} <- s,
     Var{f} : FUN(tys, ty) !,
     [[ e ^ (s_fun) : ty_body ]],
     ty == ty_body| error $[return type does not match body] @ t.

  [[ FArg(x, t) ^ (s_fun, s_outer) : ty ]] :=
     Var{x} <- s_fun,
     Var{x} : ty !,
     [[ t ^ (s_outer) : ty ]].

rules // function calls

  [[ Call(f, exps) ^ (s) : ty ]] :=
     Var{f} -> s,
     Var{f} |-> d | error $[Function [f] not declared],
     d : FUN(tys, ty) | error $[Function expected] ,
     MapSTs[[ exps ^ (s) : tys ]].
```

Tiger Names & Types: Composition

```
module statics/bindings

imports signatures/Bindings-sig
imports statics/nabl-lib
imports statics/base
imports statics/control-flow
imports statics/variables

rules // let

  [[ Let(blocks, exps) ^ (s) : ty ]] :=
     new s_body,
     Decs[[ blocks ^ (s, s_body) ]],
     Seq[[ exps ^ (s_body) : ty ]],
     distinct D(s_body).

  Decs[[ [] ^ (s_outer, s_body) ]] :=
    s_body -P-> s_outer.

  Decs[[ [block] ^ (s_outer, s_body) ]] :=
    s_body -P-> s_outer,
    Dec[[ block ^ (s_body, s_outer) ]].

  Decs[[ [block | blocks@[_|_]] ^ (s_outer, s_body) ]] :=
    new s_dec,
    s_dec -P-> s_outer,
    Dec[[ block ^ (s_dec, s_outer) ]],
    Decs[[ blocks ^ (s_dec, s_body) ]],
    distinct/name D(s_dec) | error $[duplicate declaration] @NAMES.
```

```
// Nested scopes: The scope of a variable or parameter includes the
// bodies of any function definitions in that scope. That is, access
// to variables in outer scopes is permitted, as in Pascal and Algol

/* Local redeclarations: A variable or function declaration may be
  hidden by the redeclaration of the same name (as a variable or
  function) in a smaller scope; for example, this function prints
   "6 7 6 8 6" when applied to 5:

  let
    function f(v : int) =
      let var v := 6
       in print(v);
          let  var v := 7 in print(v) end;
          print(v);
          let var v := 8 in print(v) end;
          print(v)
      end
   in f(4)
  end
*/
```

Tiger Names & Types: Variables

```
module statics/variables

imports signatures/Variables-sig
imports statics/nabl-lib
imports statics/base

rules // variable declarations

  Dec[[ VarDec(x, t, e) ^ (s, s_outer) ]] :=
     [[ t ^ (s_outer) : ty1 ]],
     [[ e ^ (s_outer) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch got [ty2] where [ty1] expected] @ e,
     Var{x} <- s,
     Var{x} : ty1 !.

  Dec[[ VarDecNoType(x, e) ^ (s, s_outer) ]] :=
     [[ e ^ (s_outer) : ty ]],
     ty != NIL() | error $[explicit type expected for variable initialized with nil],
     Var{x} <- s,
     Var{x} : ty !.

rules // variable references

  [[ Var(x) ^ (s) : ty ]] :=
     Var{x} -> s,  // declare x as variable reference
     Var{x} |-> d, // check that x resolves to a declaration
     d : ty.       // type of declaration is type of reference

rules // statements

  [[ Assign(e1, e2) ^ (s) : UNIT() ]] :=
     [[ e1 ^ (s) : ty1 ]],
     [[ e2 ^ (s) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch got [ty2] where [ty1] expected] @ e2.
```

Tiger Names & Types: Records (1)

```
module statics/records

imports signatures/Records-sig
imports statics/nabl-lib
imports statics/base

rules // record type

  [[ RecordTy(fields) ^ (s) : ty ]] :=
     new s_rec,
     ty == RECORD(s_rec),
     NIL() <! ty,
     distinct/name D(s_rec)/Field | error $[Duplicate declaration of field [NAME]] @ NAMES,
     Map2[[ fields ^ (s_rec, s) ]].

  [[ Field(x, t) ^ (s_rec, s_outer) ]] :=
     Field{x} <- s_rec,
     Field{x} : ty !,
     [[ t ^ (s_outer) : ty ]].
```

Tiger Names & Types: Records (2)

```
module statics/records

…

rules // record creation

  [[ r@Record(t, inits) ^ (s) : ty ]] :=
     [[ t ^ (s) : ty ]],
     ty == RECORD(s_rec) | error $[record type expected],
     new s_use, s_use -I-> s_rec,
     D(s_rec)/Field subseteq/name R(s_use)/Field | error $[Field [NAME] not initialized] @r,
     distinct/name R(s_use)/Field | error $[Duplicate initialization of field [NAME]] @NAMES,
     Map2[[ inits ^ (s_use, s) ]].

  [[ InitField(x, e) ^ (s_use, s) ]] :=
     Field{x} -> s_use,
     Field{x} |-> d,
     d : ty1,
     [[ e ^ (s) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch got [ty2] where [ty1] expected].

rules // record field access

  [[ FieldVar(e, f) ^ (s) : ty ]] :=
     [[ e ^ (s) : ty_e ]],
     ty_e == RECORD(s_rec),
     new s_use,
     s_use -I-> s_rec,
     Field{f} -> s_use,
     Field{f} |-> d,
     d : ty.
```

Tiger Names & Types: Records (2)

```
module statics/arrays

imports signatures/Arrays-sig
imports statics/nabl-lib
imports statics/base

rules // array type

  [[ ArrayTy(t) ^ (s) : ARRAY(ty, s')]] :=
     new s', // unique token to distinghuish the array type
     [[ t ^ (s) : ty ]].

rules // array creation

  [[ Array(t, e1, e2) ^ (s) : ty ]] :=
     [[ t ^ (s) : ty ]],
     ty == ARRAY(ty_elem, s_arr) | error $[array type expected],
     ty_elem2 <? ty_elem | error $[type mismatch [ty_indic] expected] @ e2,
     [[ e1 ^ (s) : INT() ]], // length
     [[ e2 ^ (s) : ty_elem2 ]]. // initial value

rules // array indexing

  [[ Subscript(e1, e2) ^ (s) : ty ]] :=
     [[ e1 ^ (s) : ty_arr ]],
     ty_arr == ARRAY(ty, s_arr),
     [[ e2 ^ (s) : INT() ]].
```

Tiger Names & Types: Numbers

```
module statics/numbers

imports signatures/Numbers-sig
imports statics/nabl-lib
imports statics/base

rules // literals

  [[ Int(i) ^ (s) : INT() ]].

rules // operators

  [[ Uminus(e) ^ (s) : INT() ]] :=
     [[ e ^ (s) : INT() ]].

  [[ Divide(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : INT() ]], [[ e2 ^ (s): INT() ]].

  [[ Times(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : INT() ]], [[ e2 ^ (s): INT() ]].

  [[ Minus(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : INT() ]], [[ e2 ^ (s): INT() ]].

  [[ Plus(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : INT() ]], [[ e2 ^ (s): INT() ]].

  [[ Eq(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.
```

```
  [[ Neq(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.

  [[ Gt(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.

  [[ Lt(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.

  [[ Geq(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.

  [[ Leq(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.

  [[ Or(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.

  [[ And(e1, e2) ^ (s) : INT() ]] :=
     [[ e1 ^ (s) : ty1 ]], [[ e2 ^ (s): ty2 ]],
     ty1 == ty2.
```

Tiger Names & Types: Numbers

```
module statics/control-flow

imports signatures/Control-Flow-sig
imports statics/nabl-lib
imports statics/base

rules // sequence

  Seq[[ [] ^ (s) : UNIT() ]].

  Seq[[ [e] ^ (s) : ty ]] :=
    [[ e ^ (s) : ty ]].

  Seq[[ [ e | es@[_|_] ] ^ (s) : ty ]] :=
    [[ e ^ (s) : ty' ]], Seq[[ es ^ (s) : ty ]].

  [[ Seq(es) ^ (s) : ty ]] :=
     Seq[[ es ^ (s) : ty ]].

  [[ If(e1, e2, e3) ^ (s) : ty2 ]] :=
     [[ e1 ^ (s) : INT() ]],
     [[ e2 ^ (s) : ty2 ]],
     [[ e3 ^ (s) : ty3 ]],
     ty2 == ty3 | error $[branches should have same type].

  [[ IfThen(e1, e2) ^ (s) : UNIT() ]] :=
     [[ e1 ^ (s) : INT() ]],
     [[ e2 ^ (s) : UNIT() ]].

  [[ While(e1, e2) ^ (s) : UNIT() ]] :=
     new s', s' -P-> s,
     Loop{""} <- s',
     [[ e1 ^ (s) : INT() ]],
     [[ e2 ^ (s') : UNIT() ]].
```

```
  [[ stm@For(Var(x), e1, e2, e3) ^ (s) : UNIT() ]] :=
     new s_for,
     s_for -P-> s,
     Var{x} <- s_for,
     Var{x} : INT(),
     Loop{Break()@stm} <- s_for,
     [[ e1 ^ (s) : INT() ]], // x not bound in loop bounds
     [[ e2 ^ (s) : INT() ]],
     [[ e3 ^ (s_for) : UNIT() ]]. // x bound in body

  [[ stm@Break() ^ (s) : UNIT() ]] :=
     Loop{Break()@stm} -> s,
     Loop{Break()@stm} |-> d.
```

Separation of Concerns in Name Binding

Representation

- Scope Graphs

Declarative Rules

- Scope & Type Constraint Rules

Language-Independent Tooling

- Name resolution
- Code completion
- Refactoring
- …

A language for talking about name binding

NaBL2 in Spoofax Language Workbench

workspace - Java - org.metaborg.lang.tiger/trans/statics/records.nabl2 - Eclipse

records.nabl2

```
rules // literals

  [[ NilExp()  ^ (s) : NIL() ]] := true.

rules // record creation

  [[ r@Record(t, inits) ^ (s) : ty ]] :=
     [[ t ^ (s) : ty ]],
     ty == RECORD(s_rec) | error $[record type expected],
     new s_use, s_use -I-> s_rec,
     D(s_rec)/Field subseteq/name R(s_use)/Field | error $[Field [NAME] not initialized] @r,
     distinct/name R(s_use)/Field | error $[Duplicate initialization of field [NAME]] @NAMES,
     Map2[[ inits ^ (s_use, s) ]].

  [[ InitField(x, e) ^ (s_use, s) ]] :=
     Field{x} -> s_use,
     Field{x} |-> d,
     d : ty1,
     [[ e ^ (s) : ty2 ]],
     ty2 <? ty1 | error $[type mismatch got [ty2] where [ty1] expected].

rules // record field access

  [[ FieldVar(e, f) ^ (s) : ty ]] :=
     [[ e ^ (s) : ty_e ]],
     ty_e == RECORD(s_rec),
     new s_use, s_use -I-> s_rec,
     Field{f} -> s_use,
     Field{f} |-> d,
     d : ty.
```

record.tig

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

Writable | Insert

http://spoofax.org

Applications

Domain-Specific Languages

- IceDust2 [ECOOP17]
- Green-Marl (Oracle)

Education

- Mini-Java, Tiger, Calc

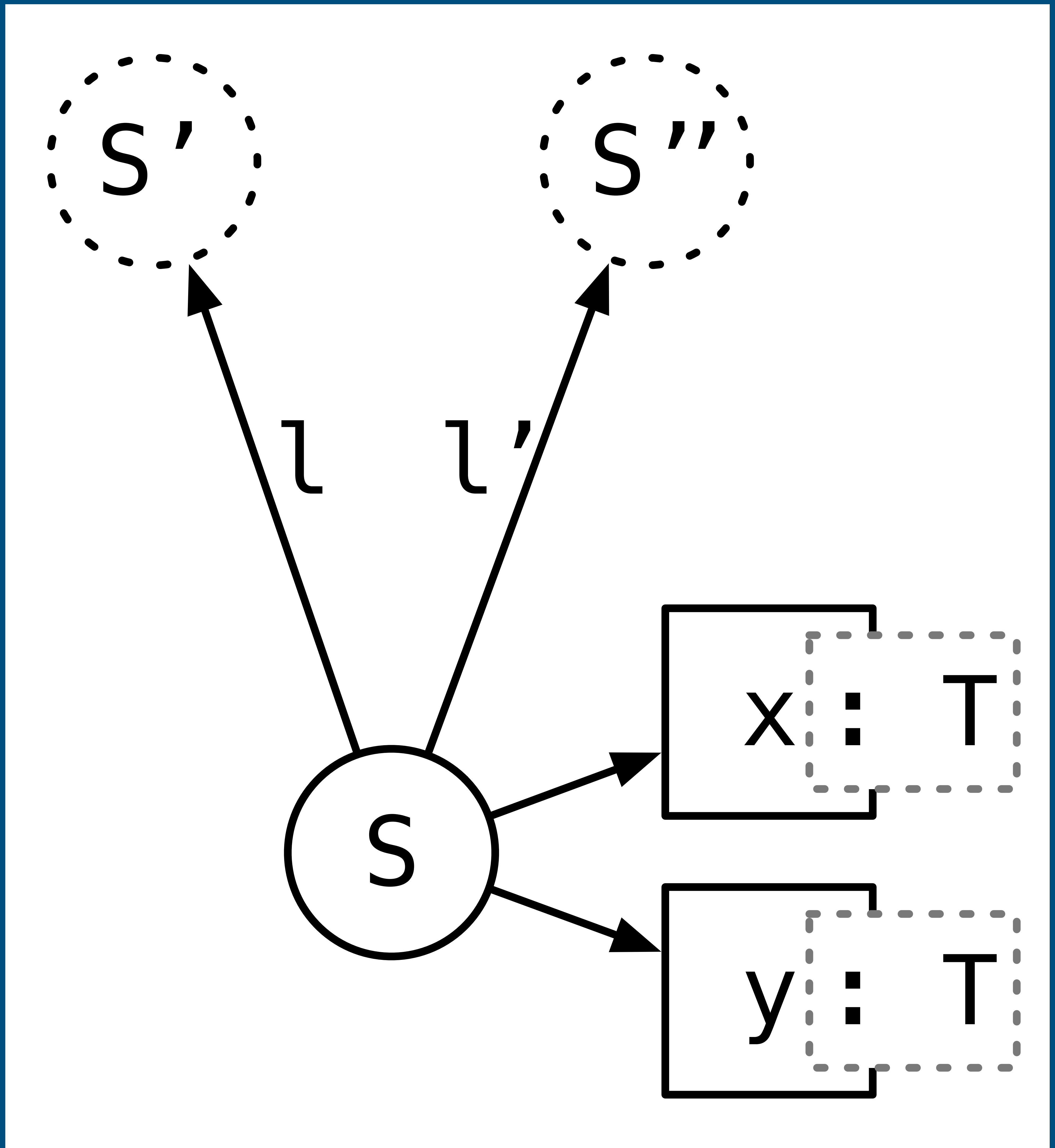
Programming languages

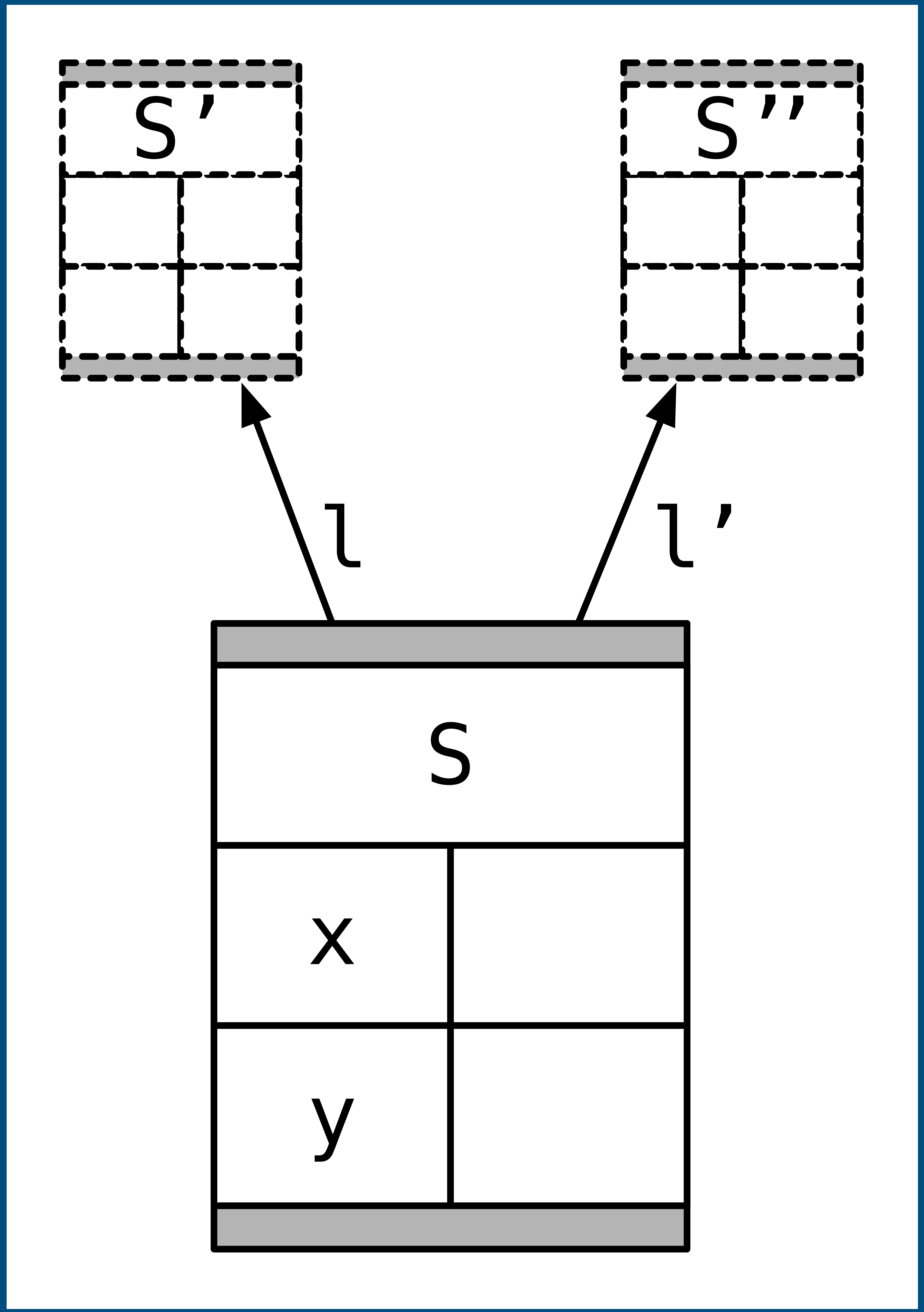
- Pascal, TypeScript, F#, **Go**, Rust
- (student projects in progress)

Bootstrapping language workbench

- NaBL2, …

Scopes Describe Frames [ECOOP16]





A Uniform Model for Memory Layout
in Dynamic Semantics

Scope Graphs for Name Binding: Status

Theory

- Resolution calculus
- Name binding and type constraints
- Resolution algorithm sound wrt calculus
- Mapping to run-time memory layout

Declarative specification

- NaBL2: generation of name and type constraints

Tooling

- Solver (second version)
- Integrated in Spoofax Language Workbench
  - editors with name and type checking
  - navigation

Scope Graphs for Name Binding: Limitations

A domain-specific (= restricted) model

- cannot describe all name resolution algorithms implemented in Turing complete languages

Normative model

- 'this is name binding'

Claim/hypothesis

- Describes all sane models of name binding

Scope Graphs for Name Binding: Future Work

Theory

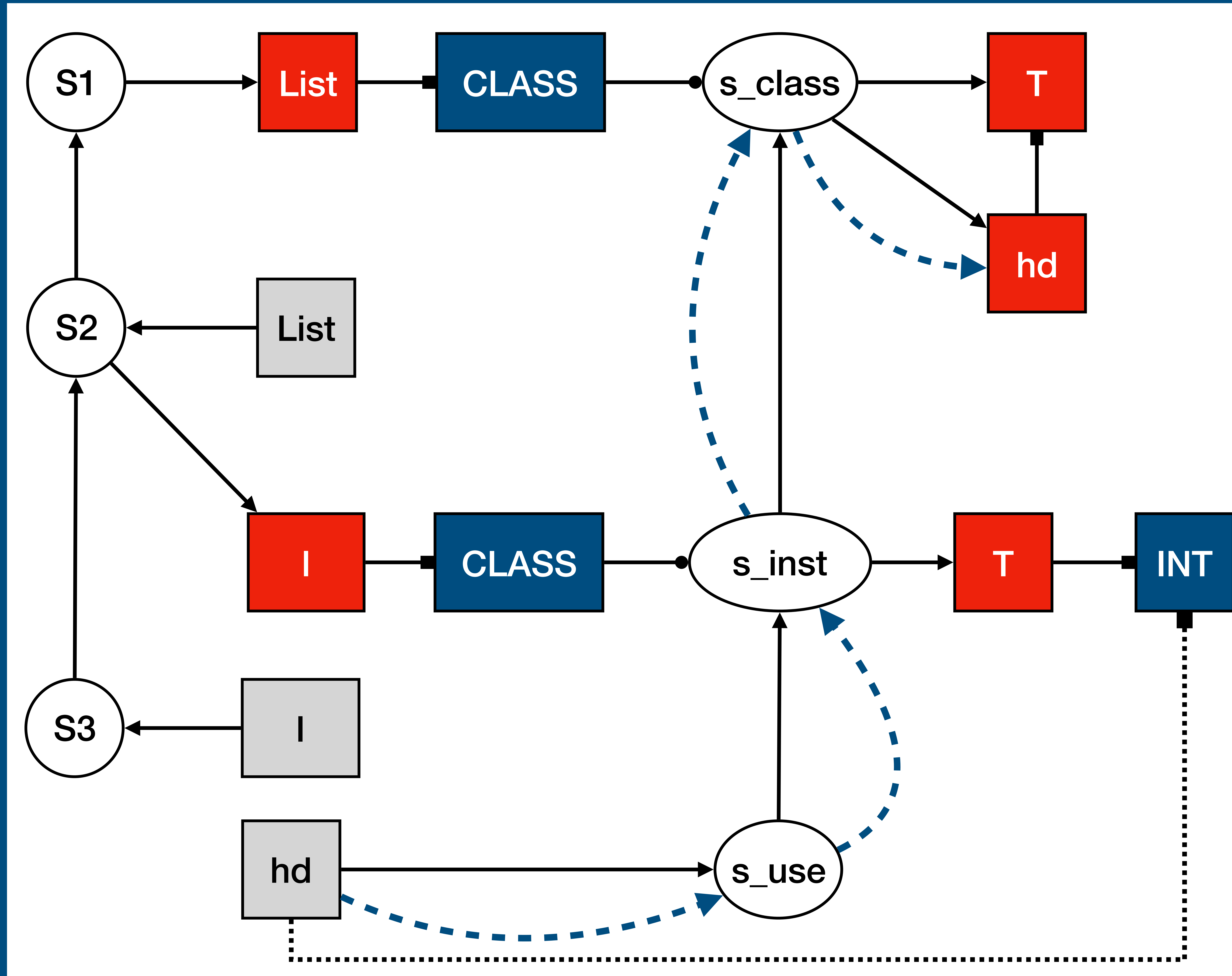
- Scopes = structural types?
  - operations for scope / type comparison
- Generics
  - DOT-style?
- Type soundness of interpreters — automatically

Tooling

- Tune name binding language (notation)
- Incremental analysis (in progress)
- Code completion
- Refactoring (renaming)

Generics

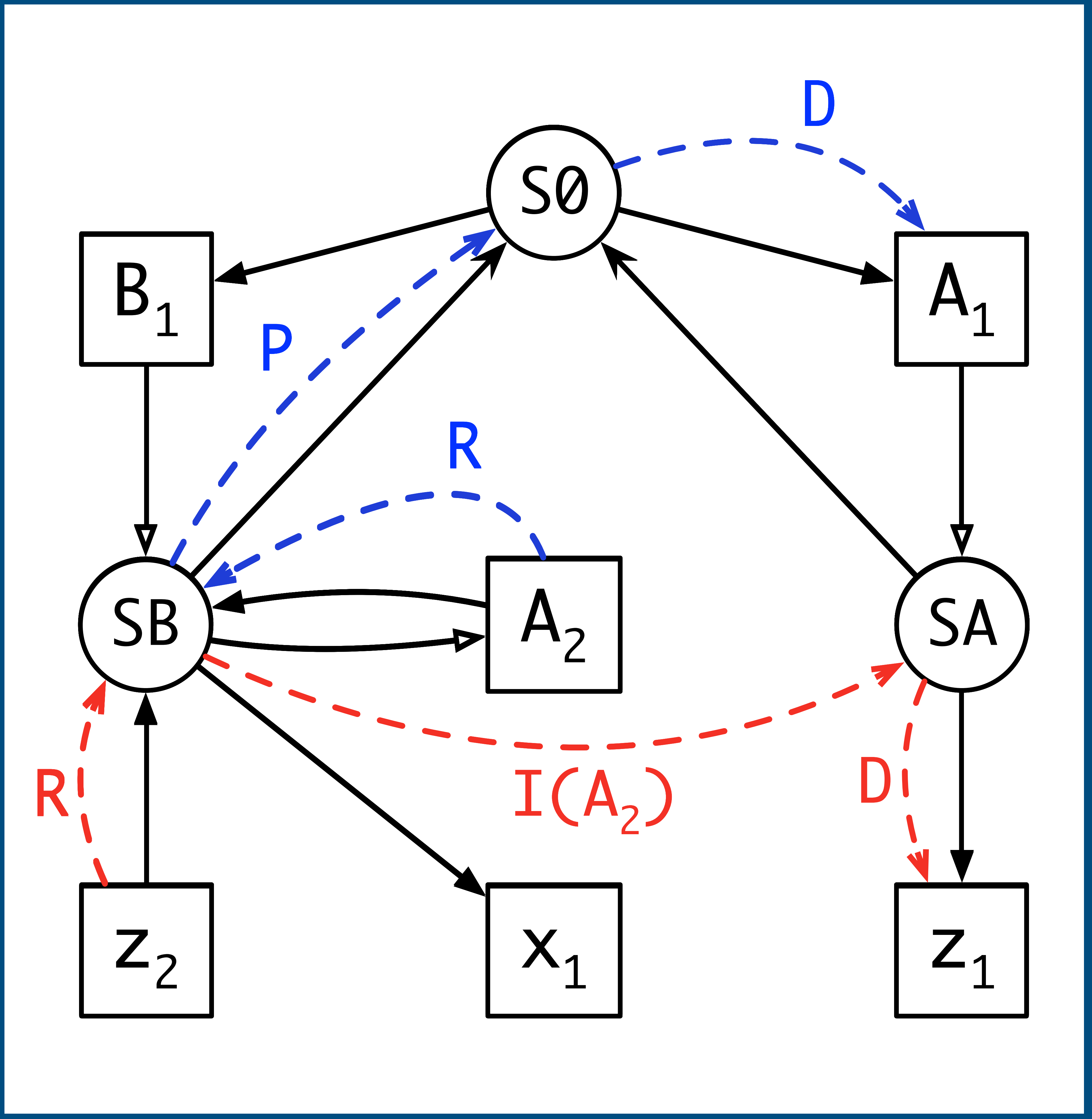
```
class List[T] {
  def hd : T
  def tl : List[T]
}
val l = new List[Int]
l.hd
```



Scope Graphs for Name Binding: The Future

A common (cross-language) understanding of name binding

A foundation for formalization and implementation of programming languages



Tiger Syntax

Tiger Syntax: Composition

```
module Tiger

imports Whitespace
imports Types
imports Identifiers
imports Bindings
imports Variables
imports Functions
imports Numbers
imports Strings
imports Records
imports Arrays
imports Control-Flow

context-free start-symbols Module

context-free syntax

  Module.Mod = Exp

context-free priorities

  Exp.Or > Exp.Array > Exp.Assign ,

  {Exp.Uminus LValue.FieldVar LValue.Subscript}
  > {left : Exp.Times Exp.Divide}
```

Tiger Syntax: Lexical Syntax

```
module Identifiers

lexical syntax

  Id = [a-zA-Z] [a-zA-Z0-9\_]*

lexical restrictions

  Id -/- [a-zA-Z0-9\_]

lexical syntax

  Id = "nil" {reject}
  Id = "let" {reject}
  Id = … {reject}
```

```
module Strings

sorts StrConst

lexical syntax

  StrConst = "\"" StrChar* "\""
  StrChar = ~[\\\"\n]
  StrChar = [\\] [n]
  StrChar = [\\] [t]
  StrChar = [\\] [\^] [A-Z]
  StrChar = [\\] [0-9] [0-9] [0-9]
  StrChar = [\\] [\"]
  StrChar = [\\] [\\]
  StrChar = [\\] [\ \t\n]+ [\\]

context-free syntax // records

  Exp.String = StrConst
```

Tiger Syntax: Whitespace

```
module Whitespace

lexical syntax

  LAYOUT           = [\ \t\n\r]
  CommentChar      = [\*]
  LAYOUT           = "/*" InsideComment* "*/"
  InsideComment    = ~[\*]
  InsideComment    = CommentChar
  LAYOUT           = SingleLineComment
  SingleLineComment = "//" ~[\n\r]* NewLineEOF
  NewLineEOF       = [\n\r]
  NewLineEOF       = EOF
  EOF              =

lexical restrictions

  // Ensure greedy matching for lexicals

  CommentChar   -/- [\/]
  EOF -/- ~[]

context-free restrictions

  // Ensure greedy matching for comments

  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\/]
  LAYOUT? -/- [\/].[\*]
```

Tiger Syntax: Variables and Functions

```
module Bindings

imports Control-Flow
imports Identifiers
imports Types
imports Functions
imports Variables

sorts Declarations

context-free syntax

  Exp.Let = <
    let
      <{Dec "\n"}*>
     in
      <{Exp ";\n"}*>
    end
  >

  Declarations.Declarations = <
    declarations <{Dec "\n"}*>
  >
```

```
module Variables

imports Identifiers
imports Types

sorts Var

context-free syntax

  Dec.VarDec = <var <Id> : <Type> := <Exp>>


  Dec.VarDecNoType = <var <Id> := <Exp>>
  Var.Var = Id

  LValue = Var

  Exp = LValue

  Exp.Assign = <<LValue> := <Exp>>
```

```
module Functions

imports Identifiers
imports Types

context-free syntax

  Dec.FunDecs = <<{FunDec "\n"}+>> {longest-match}

  FunDec.ProcDec = <
    function <Id>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <Id>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >

  FArg.FArg = <<Id> : <Type>>

  Exp.Call = <<Id>(<{Exp ", "}*>)>
```

Tiger Syntax: Numbers

```
module Numbers

lexical syntax

  IntConst = [0-9]+

lexical syntax

  RealConst.RealConstNoExp = IntConst "." IntConst
  RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst
  Sign = "+"
  Sign = "-"

context-free syntax

  Exp.Int      = IntConst

  Exp.Uminus   = [- [Exp]]
  Exp.Times    = [[Exp] * [Exp]]    {left}
  Exp.Divide   = [[Exp] / [Exp]]    {left}
  Exp.Plus     = [[Exp] + [Exp]]    {left}
  Exp.Minus    = [[Exp] - [Exp]]    {left}

  Exp.Eq       = [[Exp] = [Exp]]    {non-assoc}
  Exp.Neq      = [[Exp] <> [Exp]]   {non-assoc}
  Exp.Gt       = [[Exp] > [Exp]]    {non-assoc}
  Exp.Lt       = [[Exp] < [Exp]]    {non-assoc}
  Exp.Geq      = [[Exp] >= [Exp]]   {non-assoc}
  Exp.Leq      = [[Exp] <= [Exp]]   {non-assoc}

  Exp.And      = [[Exp] & [Exp]]    {left}
  Exp.Or       = [[Exp] | [Exp]]    {left}
```

```
context-free priorities

  {Exp.Uminus}
  > {left :
    Exp.Times
    Exp.Divide}
  > {left :
    Exp.Plus
    Exp.Minus}
  > {non-assoc :
    Exp.Eq
    Exp.Neq
    Exp.Gt
    Exp.Lt
    Exp.Geq
    Exp.Leq}
  > Exp.And
  > Exp.Or
```

Tiger Syntax: Records, Arrays, Types

```
module Records
imports Base
imports Identifiers
imports Types
context-free syntax // records

  Type.RecordTy = <
    {
       <{Field ", \n"}*>
    }
  >

  Field.Field = <<Id> : <TypeId>>

  Exp.NilExp = <nil>

  Exp.Record = <<TypeId>{ <{InitField ", "}*> }>

  InitField.InitField = <<Id> = <Exp>>

  LValue.FieldVar = <<LValue>.<Id>>
```

```
module Arrays
imports Types
context-free syntax // arrays

  Type.ArrayTy = <array of <TypeId>>

  Exp.Array = <<TypeId>[<Exp>] of <Exp>>

  LValue.Subscript = <<LValue>[<Index>]>

  Index = Exp
```

```
module Types

imports Identifiers
imports Bindings

sorts Type

context-free syntax // type declarations

  Dec.TypeDecs = <<{TypeDec "\n"}+>> {longest-match}

  TypeDec.TypeDec = <type <Id> = <Type>>

context-free syntax // type expressions

  Type = TypeId
  TypeId.Tid  = Id

sorts Ty
context-free syntax // semantic types

  Ty.INT      = <INT>
  Ty.STRING   = <STRING>
  Ty.NIL      = <NIL>
  Ty.UNIT     = <UNIT>
  Ty.NAME     = <NAME <Id>>
  Ty.RECORD   = <RECORD <Id>>
  Ty.ARRAY    = <ARRAY <Ty> <Id>>
  Ty.FUN      = <FUN ( <{Ty ","}*> ) <Ty>>
```