Lecture 8: Type Constraints

CS4200 Compiler Construction

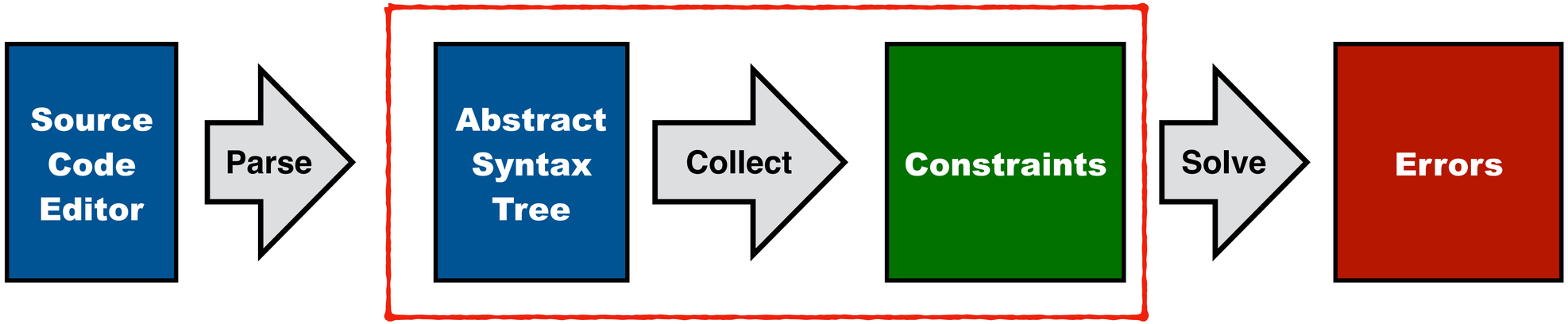
Hendrik van Antwerpen

TU Delft

September 2018

TU Delft

This lecture



- Type checking with constraints
- The meta-language NaBL2 to write type specifications
- Examples of different name binding and typing patterns in NaBL2
- Defining the meaning of constraints with constraint semantics

Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

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

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

Background

SLE 2012

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

Declarative Name Binding and Scope Rules

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

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

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

1 Introduction

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

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

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

This paper introduces scope graphs as a language-independent representation for the binding information in programs.

Best EAPLS paper at ETAPS 2015

ESOP 2015

http://dx.doi.org/10.1007/978-3-662-46669-8_9

A Theory of Name Resolution

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

[1)] Delft University of Technology, The Netherlands,
{p.j.m.neron, e.visser, g.wachsmuth}@tudelft.nl,
[2)] Portland State University, Portland, OR, USA
tolmach@pdx.edu

Abstract. We describe a language-independent theory for name binding and resolution, suitable for programming languages with complex scoping rules including both lexical scoping and modules. We formulate name resolution as a two-stage problem. First a language-independent scope graph is constructed using language-specific rules from an abstract syntax tree. Then references in the scope graph are resolved to corresponding declarations using a language-independent resolution process. We introduce a resolution calculus as a concise, declarative, and language-independent specification of name resolution. We develop a resolution algorithm that is sound and complete with respect to the calculus. Based on the resolution calculus we develop language-independent definitions of α-equivalence and rename refactoring. We illustrate the approach using a small example language with modules. In addition, we show how our approach provides a model for a range of name binding patterns in existing languages.

1 Introduction

Naming is a pervasive concern in the design and implementation of programming languages. Names identify *declarations* of program entities (variables, functions, types, modules, etc.) and allow these entities to be *referenced* from other parts of the program. Name *resolution* associates each reference to its intended declaration(s), according to the semantics of the language. Name resolution underlies most operations on languages and programs, including static checking, translation, mechanized description of semantics, and provision of editor services in IDEs. Resolution is often complicated, because it cuts across the local inductive structure of programs (as described by an abstract syntax tree). For example, the name introduced by a `let` node in an ML AST may be referenced by an arbitrarily distant child node. Languages with explicit name spaces lead to further complexity; for example, resolving a qualified reference in Java requires first resolving the class or package name to a context, and then resolving the member name within that context. But despite this diversity, it is intuitively clear that the basic concepts of resolution reappear in similar form across a broad range of lexically-scoped languages.

In practice, the name resolution rules of real programming languages are usually described using *ad hoc* and informal mechanisms. Even when a language *is* formalized, its resolution rules are typically encoded as part of static

Separating type checking into constraint generation and constraint solving provides more declarative definition of type checkers. This paper introduces a constraint language integrating name resolution into constraint resolution through scope graph constraints.

This is the basis for the design of the NaBL2 static semantics specification language.

PEPM 2016

https://doi.org/10.1145/2847538.2847543

A Constraint Language for Static Semantic Analysis based on Scope Graphs

Hendrik van Antwerpen
Delft University of Technology
h.vanantwerpen@student.tudelft.nl

Pierre Néron
Delft University of Technology
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University
tolmach@pdx.edu

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Abstract

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

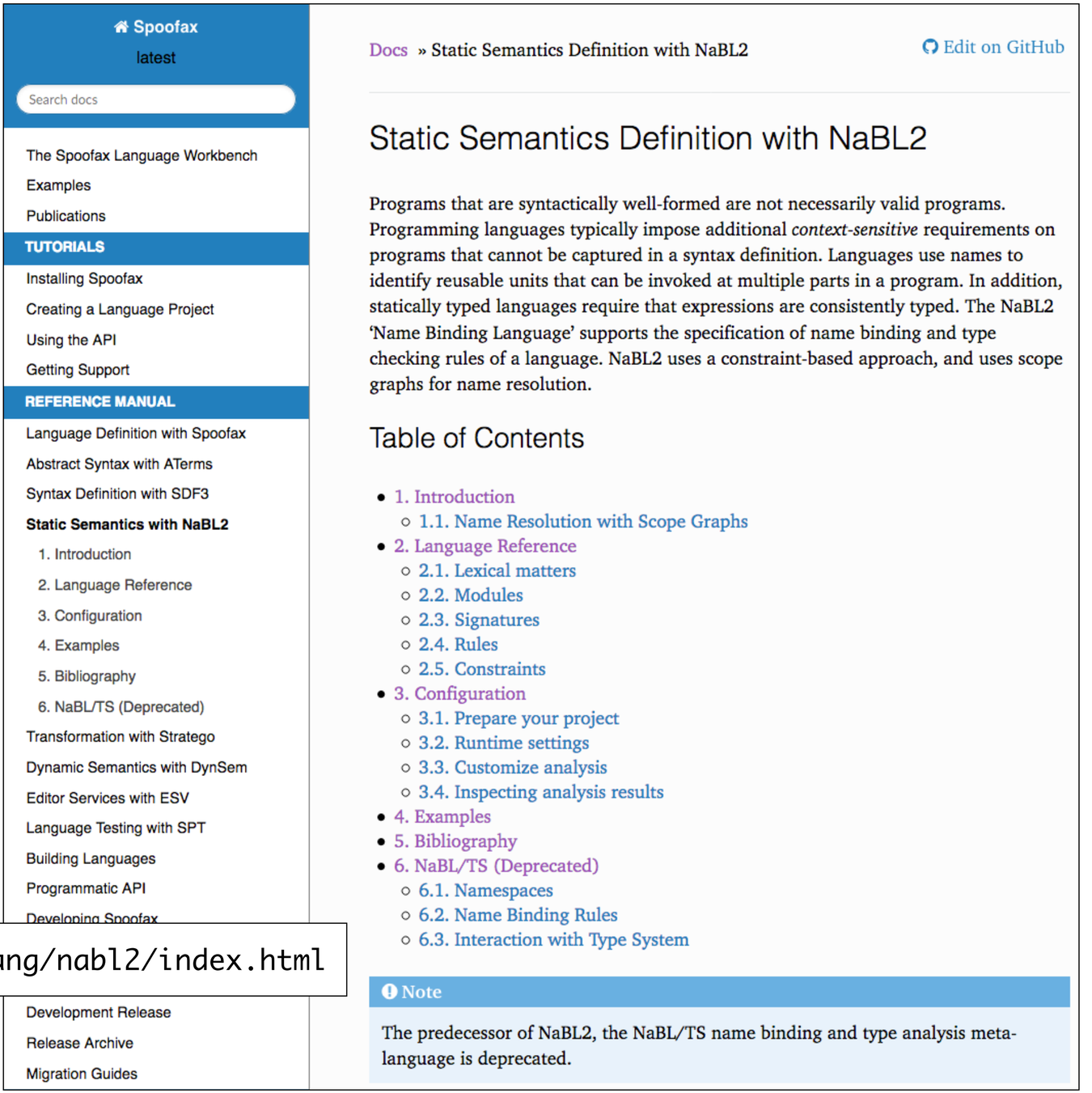
In recent work we show how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one, none, or many declarations. A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

In this paper, we refine and extend the scope graph framework of [14] to construct a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checking step–would work. But the full story is more complicated, because sometimes name resolution also depends on type resolution. For example, in a language that uses dot notation for object field projection, determining the resolution of `x` in the expression `r.x` requires first determining the object type of `r`, which in turn requires name resolution again. Thus, our framework requires a unified mechanism for expressing and solving arbitrarily interdependent naming and typing resolution problems.

To address this challenge, we base our framework on a language of *constraints*. Term equality constraints are a standard choice for describing type inference problems while abstracting away from the details of an AST in a particular language. Adopting constraints to describe both typing and scoping requirements has the advantage of uniform notation, and, more importantly, provides a clean way to combine naming and typing problems. In particular, we extend our previous work to support *incomplete* scope graphs, which correspond to constraint sets with (as yet) unresolved variables.

Our new framework continues to satisfy the criteria outlined above. (i) The resolution calculus and standard term equality constraint theory provide a solid language-independent theory for name and type resolution. (ii) Our framework supports type checking and inference for statically typed, monomorphic languages with user-defined types, and can also express uniqueness and completeness requirements on declarations and initializers. The framework inherits from scope graphs the ability to model a broad range of binding patterns, including many variants of lexical scoping, records, and modules. (iii) The constraint language has a declarative semantics given by a constraint satisfaction relation, which employs the resolution calculus to define the semantics of name resolution relative to a scope graph. We define a constraint resolution algorithm based on our previous name resolution algorithm,

Documentation for NaBL2 at the metaborg.org website.



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

This paper describes the next generation of the approach.

Addresses (previously) open issues in expressiveness of scope graphs for type systems:
- Structural types
- Generic types

Addresses open issue with staging of information in type systems.

Introduces Statix DSL for definition of type systems.

Prototype of Statix is available in Spoofax HEAD, but not ready for use in project yet.

The future

OOPSLA 2018

To appear

Scopes as Types

HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands
CASPER BACH POULSEN, Delft University of Technology, Netherlands
ARJEN ROUVOET, Delft University of Technology, Netherlands
EELCO VISSER, Delft University of Technology, Netherlands

Scope graphs are a promising generic framework to model the binding structures of programming languages, bridging formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs. However, previous work on scope graphs has been limited to simple, nominal type systems. In this paper, we show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries. We introduce Statix, a new domain-specific meta-language for the specification of static semantics, based on scope graphs and constraints. We evaluate the scopes as types approach and the Statix design in case studies of the simply-typed lambda calculus with records, System F, and Featherweight Generic Java.

CCS Concepts: • **Software and its engineering → Semantics**; **Domain specific languages**;

Additional Key Words and Phrases: static semantics, type system, type checker, name resolution, scope graphs, domain-specific language

ACM Reference Format:
Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (November 2018), 30 pages. https://doi.org/10.1145/3276484

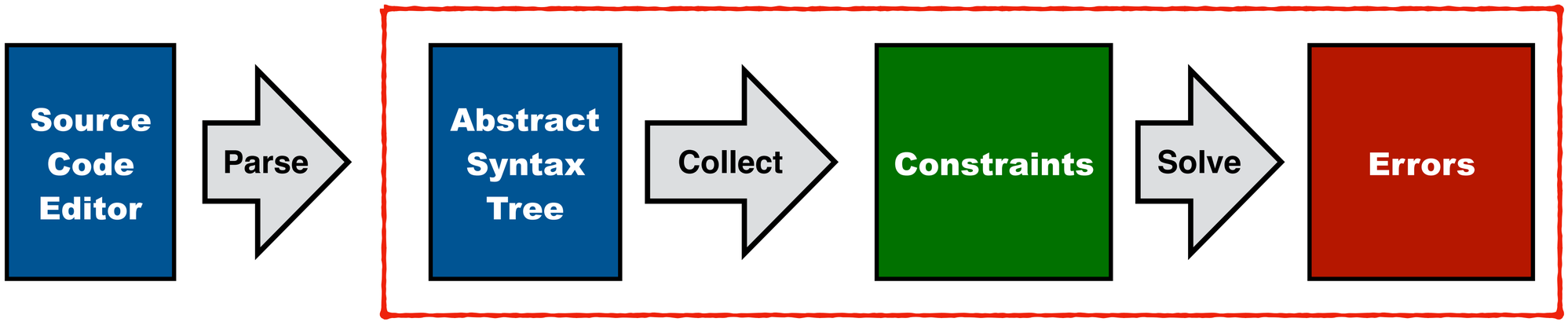
1 INTRODUCTION

The goal of our work is to support high-level specification of type systems that can be used for multiple purposes, including reasoning (about type safety among other things) and the implementation of type checkers [Visser et al. 2014]. Traditional approaches to type system specification do not reflect the commonality underlying the name binding mechanisms for different languages. Furthermore, operationalizing name binding in a type checker requires carefully staging the traversals of the abstract syntax tree in order to collect information before it is needed. In this paper, we introduce an approach to the declarative specification of type systems that is close in abstraction to traditional type system specifications, but can be directly interpreted as type checking rules. The approach is based on scope graphs for name resolution, and constraints to separate traversal order from solving order.

Authors' addresses: Hendrik van Antwerpen, Delft University of Technology, Delft, Netherlands, H.vanAntwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, Netherlands, C.B.Poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, Delft, Netherlands, A.J.Rouvoet@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, Netherlands, E.Visser@tudelft.nl.

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

Type Checking with Constraints



Type checking proceeds in two steps

Typing Constraints

What is constraint generation?

- Function that maps a program to constraints
- Defined as a top-down traversal, parameters can be passed down
- *Specification* of what it means to be well-typed!

What are constraints?

- Logical assertions that should hold for well-typed programs
  - Logical variables represent unknown values
- Constraint language determines what assertions can be made
  - Type equality, sub typing, name resolution, ...
- Determines the *expressiveness* of the specification!

Constraint Solving

- Find an assignment for all logical variables, such that constraints are satisfied
- More on this next week

Typing Constraints

Challenges for type checker implementations?

- Collecting (non-lexical) binding information before use
- Dealing with unknown (type) values

Constraints separate *what* from *how*

- Constraint generator + constraint language says *what* is a well-typed program
- Constraint solver says *how* to determine that a program is well-typed

Constraints separate computation from program structure

- Constraint generator follows the structure of the program
- Constraint solver is flexible in order of resolution

Constraint naturally support unknown values

- Variables represent unknown values
- Constraint resolution *infers* appropriate values

NaBL2

NaBL2

What is NaBL2?

- Domain-specific specification language…
- … to write constraint generators
- Comes with a generic solver

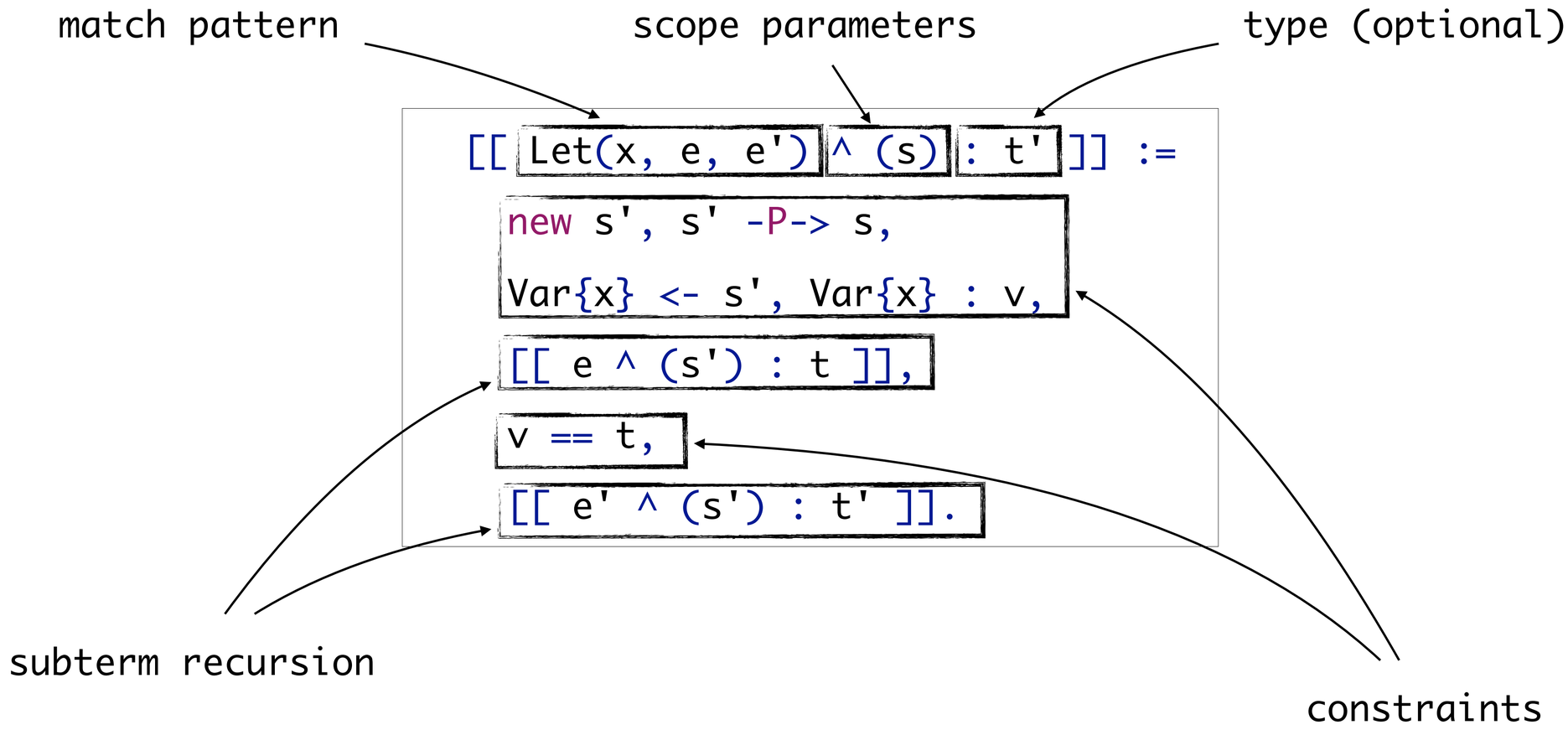
What features does it support?

- Rich binding structures using scope graphs
- Type equality and nominal sub-typing
- Type-dependent name resolution

Limitations

- Restricted to the domain-specific (= restricted) model
  - Not all name binding patterns in the wild can be expressed
- Hypothesis is that all sensible patterns are expressible

Constraint Rules

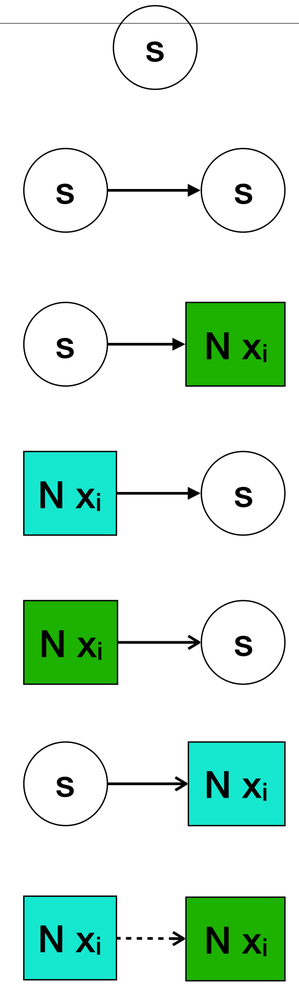


Tips for writing correct specifications

- Rules have an implicit signature: sort of the matched term, number of scopes, whether it has a type or not
- All rules for terms of a certain sort (e.g., all expression rules) must have the same signature
- All subterm rule invocations must follow the signature of the rules of the subterm sort
- Traverse every term at most once

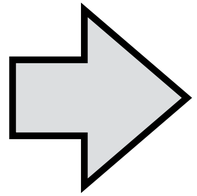
Scope Graph Constraints

```
new s              // create a new scope

s1 -L-> s2         // edge labeled with L 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} =L=> s        // declaration x is associated with scope s

N{x} <=L= s        // scope s imports via reference x

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

[[ e ^ (s) ]]      // subterm e is scoped by scoped s
```



Scope Graph Example

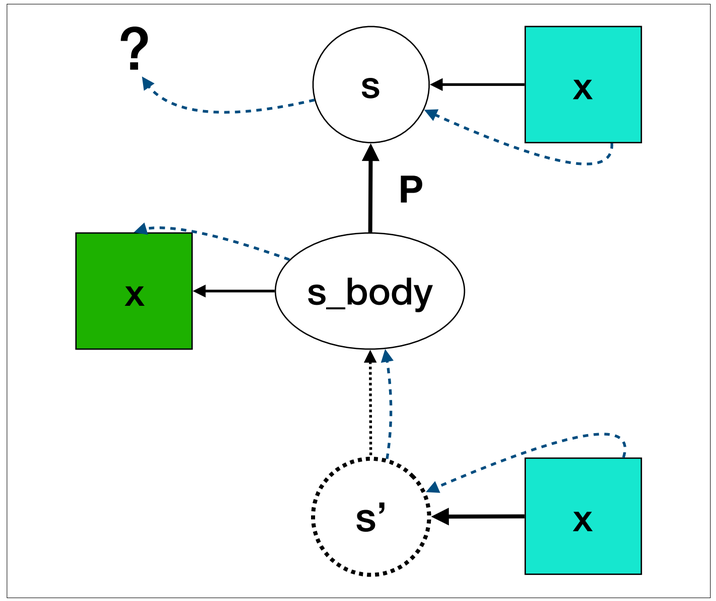
```
let
  var x : int := x + 1
 in
   x + 1
end
```



```
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
```

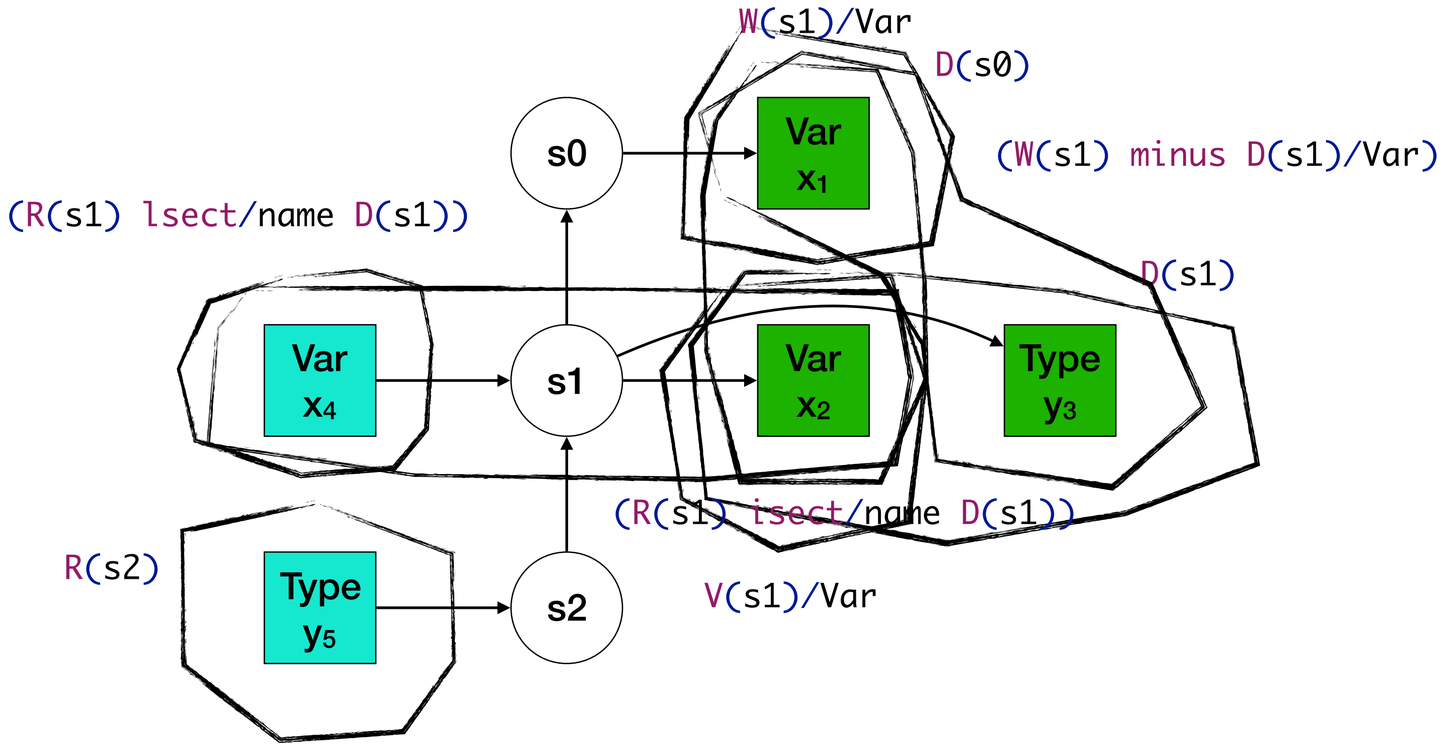


Name Set Constraints & Expressions

```
distinct S              distinct/name S          // elements/names in S are distinct
S1 subseteq S2          S1 subseteq/name S2      // elements/names in S1 are a subset of S2
S1 seteq S2             S1 seteq/name S2         // elements/names in S1 are equal to S2
```

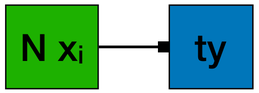
```
0                                                // empty set
D(s)                    D(s)/N                   // all/namespace N declarations in s
R(s)                    R(s)/N                   // all/namespace N references in s
V(s)                    V(s)/N                   // visible declarations in s (w/ shadowing)
W(s)                    W(s)/N                   // reachable declarations in s (no shadowing)
(S1 union S2)                                    // union of S1 and S2
(S1 isect S2)           (S1 isect/name S2)       // intersection of elements/names S1 and S2
(S1 lsect S2)           (S1 lsect/name S2)       // elements/names in S1 that are in S2
(S1 minus S2)           (S1 minus/name S2)       // elements/names in S1 that are not in S2
```

Name Set Examples



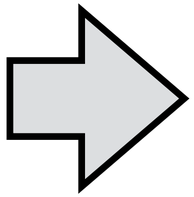
Type Constraints

```
ty1 == ty2              // types ty1 and ty2 are equal

ty1 <R! ty2             // declare ty1 to be related to ty2 in R

ty1 <R? ty2             // require ty1 to be related to ty2 in R

   ....                 // … extensions …

d : ty                  // declaration d has type ty

[[ e ^ (s) : ty ]]      // subterm e has type ty
```



Type Example

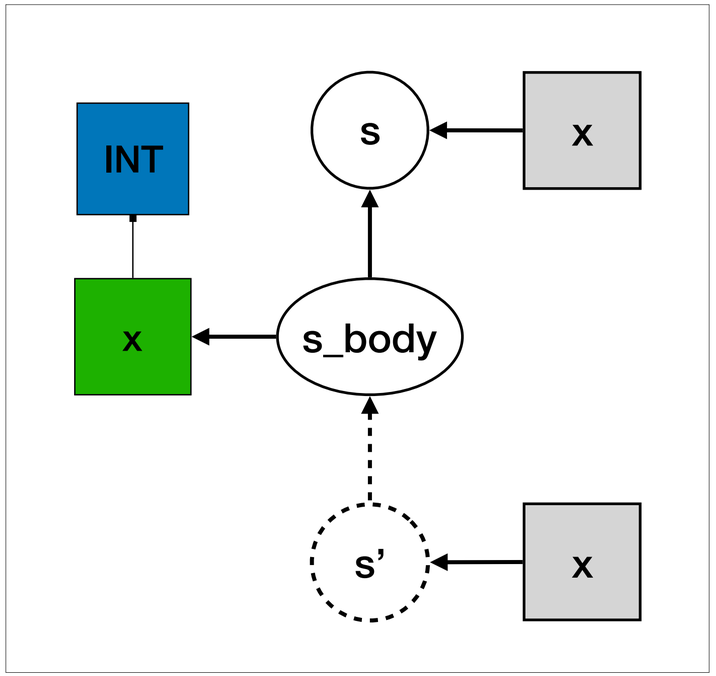
```
let
  var x : int := x + 1
 in
   x + 1
end
```



```
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
```



NaBL2 Configuration

```
signature

  namespaces
    Var

  name resolution
    labels P I
    order D < P, D < I, I < P
    well-formedness P* I*

  relations
    reflexive, transitive, anti-symmetric sub : Type * Type
```

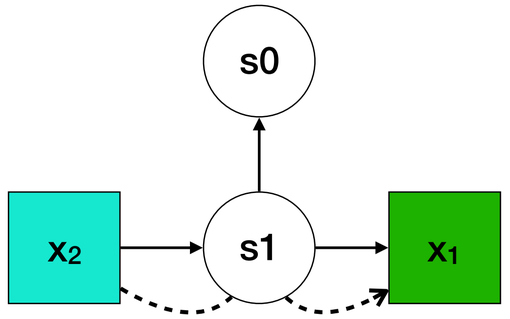
Tiger in NaBL2

Variable Declarations and References

```
let                          s0
  var x1 : int := 1
in                           s1
  x2 + 1
end
```

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

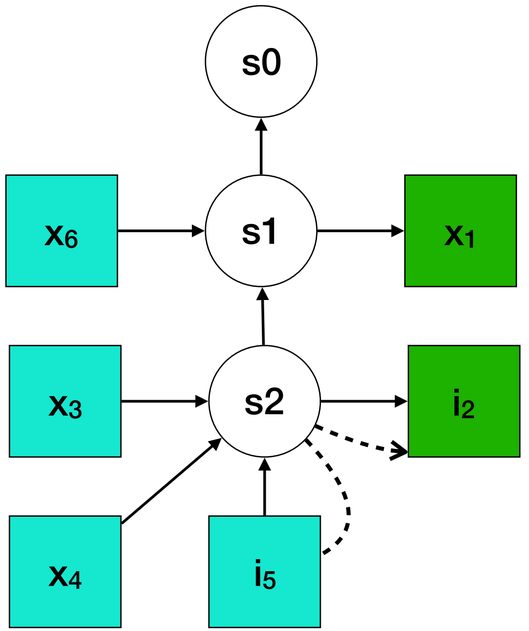
```
Dec[[ VarDec(x, t, e) ^ (s, s_outer) ]] :=
  [[ t ^ (s_outer) ]],
  [[ e ^ (s_outer) ]],
  Var{x} <- s.

[[ Var(x) ^ (s) ]] :=
  Var{x} -> s,
  Var{x} |-> d.
```



Scoping of Loop Variable

```
let                                          s0
   var x1 : int := 0
in                                           s1
   for i2 := 1 to 4 do
      x3 := x4 + i5;                         s2
   x6 * 7
end
```

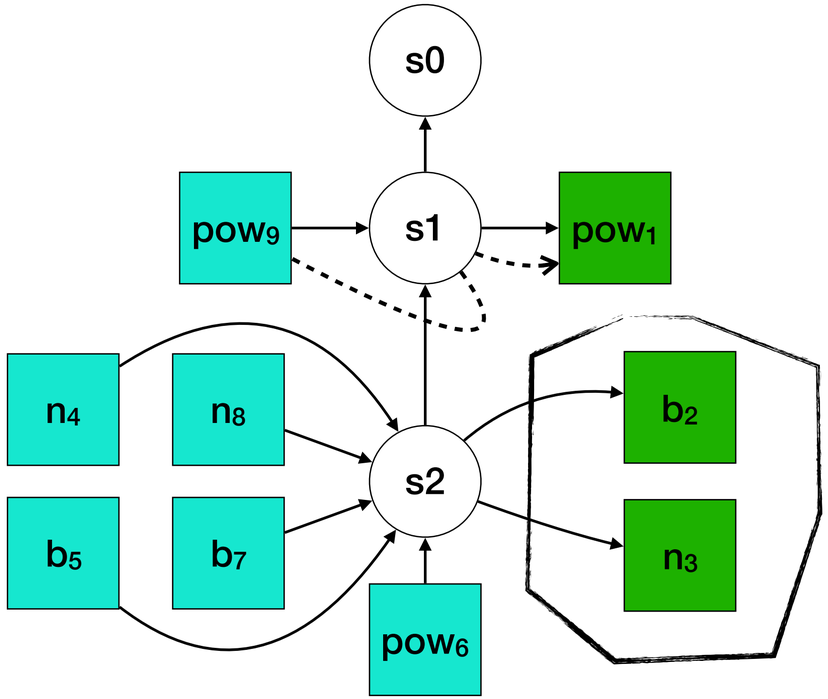
```
[[ stm@For(Var(x), e1, e2, e3) ^ (s) ]] :=
     new s_for,
     s_for -P-> s,
     Var{x} <- s_for,
     Loop{Break()@stm} <- s_for,
     [[ e1 ^ (s) ]],
     [[ e2 ^ (s) ]],
     [[ e3 ^ (s_for) ]].
```



Function Declarations and References

```
                                                                    s0
let
                                                                    s1
  function pow1(b2 : int, n3 : int) : int =
                                                                    s2
    if n4 < 1
      then 1
      else (b5 * pow6(b7, n8 - 1))
in
  pow9(2, 7)
end
```

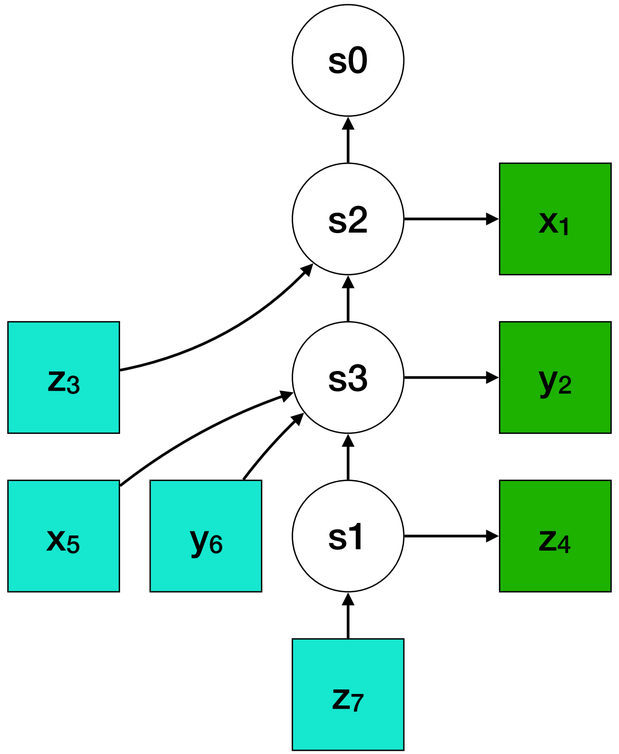
```
[[ FunDec(f, args, t, e) ^ (s, s_outer) ]] :=
    Var{f} <- s,
    new s_fun,
    s_fun -P-> s,
    Map2[[ args ^ (s_fun, s_outer) ]],
    distinct/name D(s_fun),
    [[ e ^ (s_fun) ]].

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

[[ Call(f, exps) ^ (s) ]] :=
    Var{f} -> s,
    Var{f} |-> d,
    Map1[[ exps ^ (s) ]].
```



Sequential Let

```
let                                  s0
    var x1 := 2
    var y2 := z3 + 11                s2
    var z4 := x5 * y6                s3
in z7 end                            s1
```

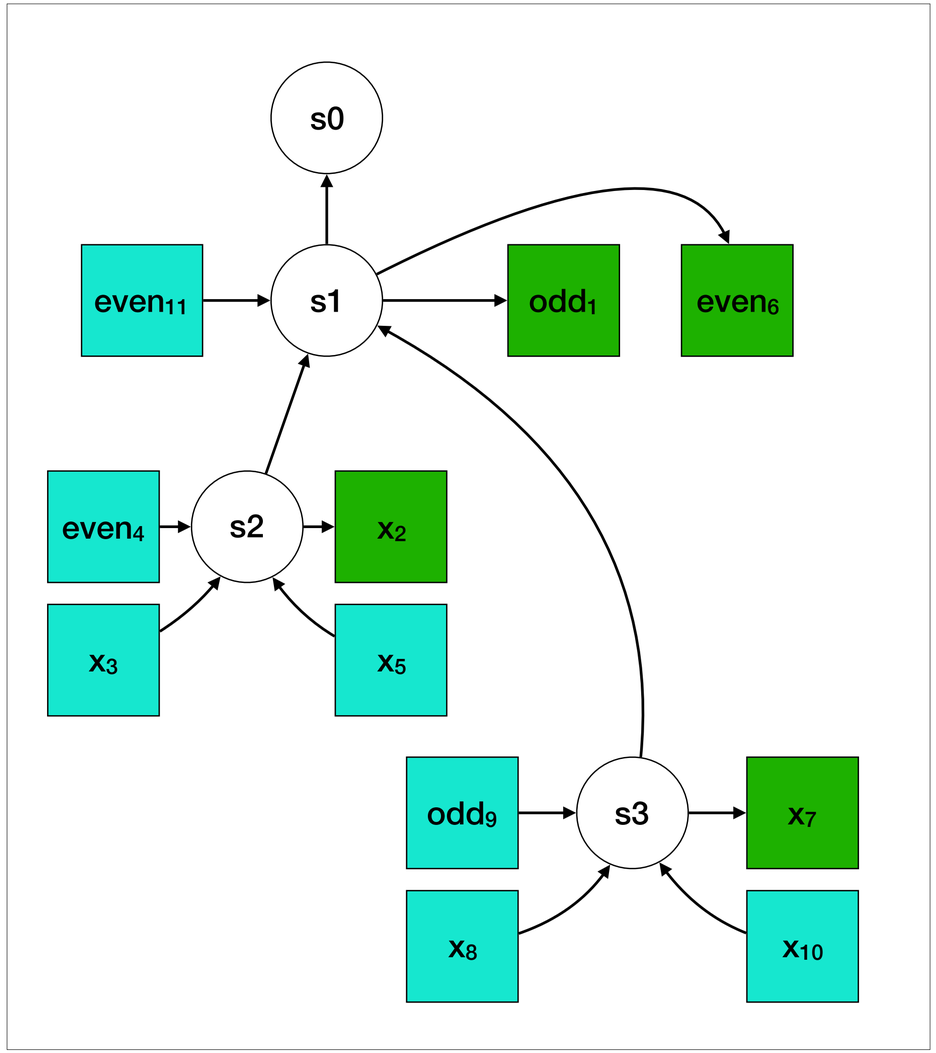
```
[[ Let(blocks, exps) ^ (s) ]] :=
    new s_body,
    Decs[[ blocks ^ (s, s_body) ]],
    Seq[[ exps ^ (s_body) ]],
    distinct D(s_body).

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).
```



Mutually Recursive Functions

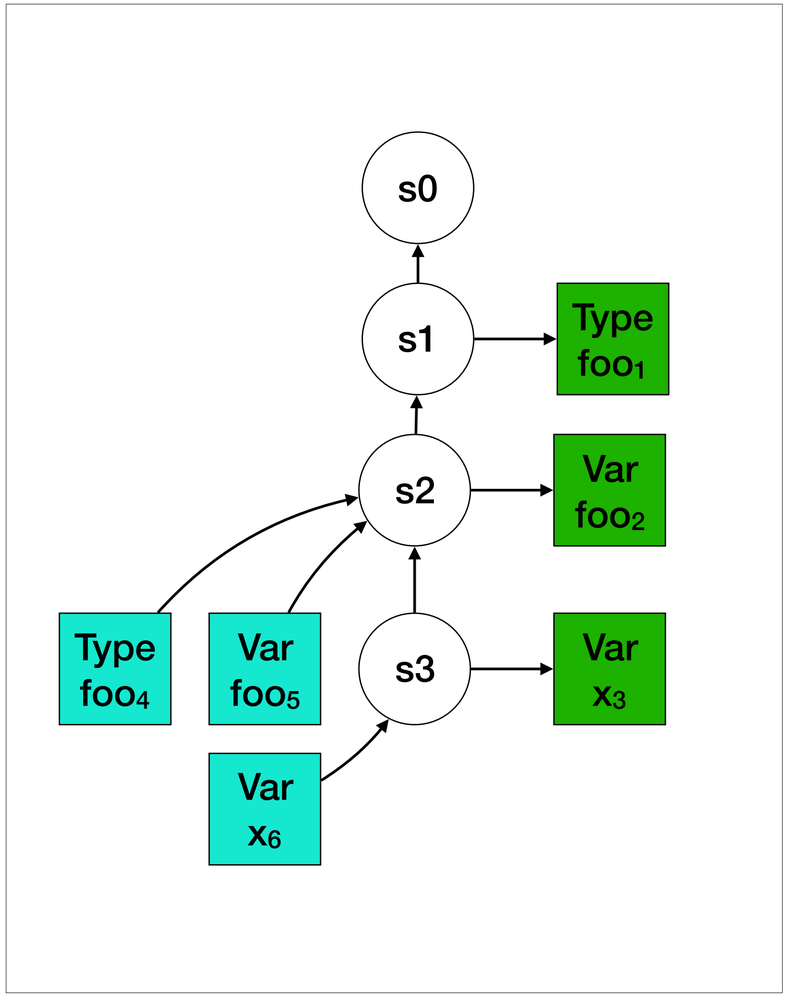
```
                                                        s0
let
  function odd1(x2 : int) : int =                      s1
    if x3 > 0 then even4(x5-1) else 0                   s2
  function even6(x7 : int) : int =
    if x8 > 0 then odd9(x10-1) else 1                   s3
 in
   even11(34)
end
```

```
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),
  MapTs2[[ args ^ (s_fun, s_outer) ]],
```



Namespaces

```
let                                          s0
  type foo1 = int
  var foo2 : int := 24                       s1
  var x3 : foo4 := foo5                      s2
in                                           s3
  x6
end
```




```
[[ TypeDec(x, t) ^ (s) ]] :=
    [[ t ^ (s) ]],
    Type{x} <- s.
```

```
[[ VarDec(x, t, e) ^ (s, s_outer) ]] :=
    [[ t ^ (s_outer) ]],
    [[ e ^ (s_outer) ]],
    Var{x} <- s.
```

Explicit versus Inferred Variable Type

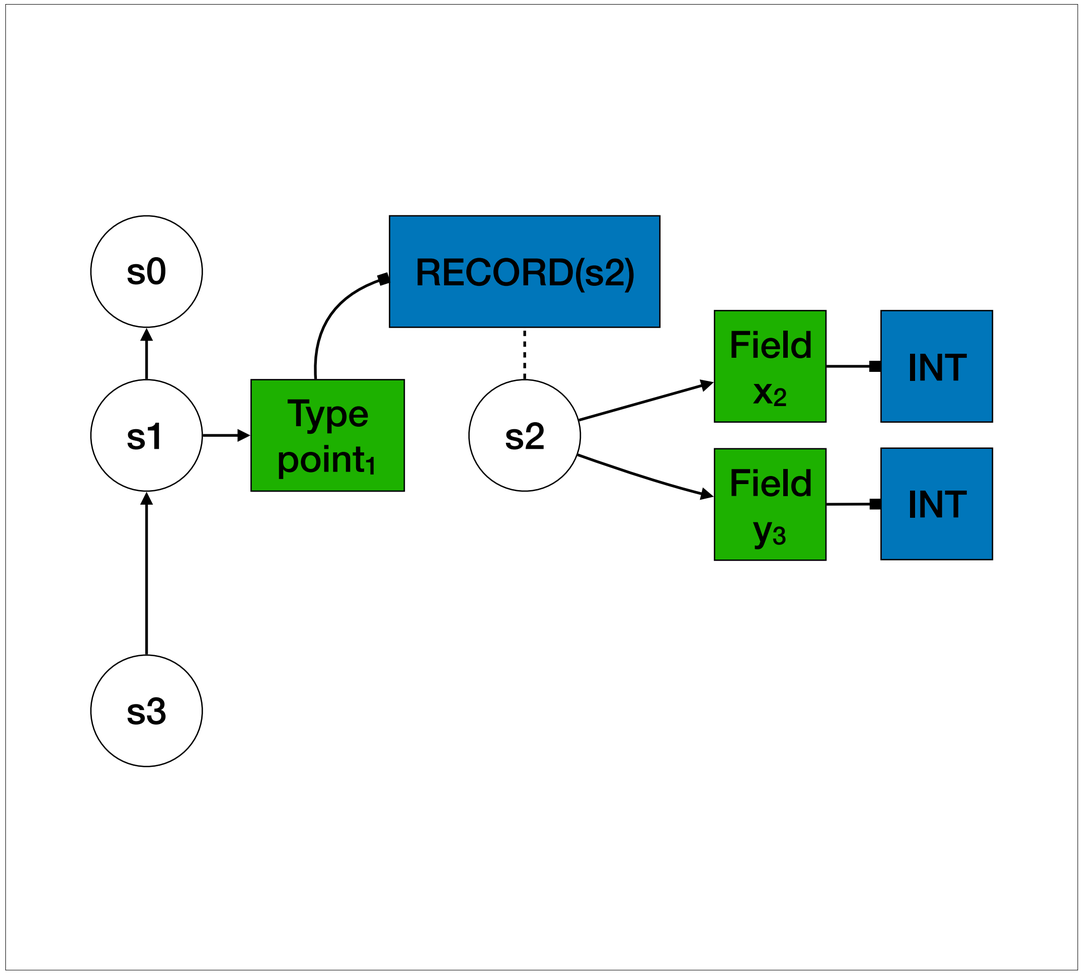
```
let
  var x : int := 20 + 1
  var y       := 21
in
  x + y
end
```

```
[[ VarDec(x, t, e) ^ (s, s_outer) ]] :=
    [[ t ^ (s_outer) : ty1 ]],
    [[ e ^ (s_outer) : ty2 ]],
    ty2 <? ty1,
    Var{x} <- s,
    Var{x} : ty1 !.

[[ VarDecNoType(x, e) ^ (s, s_outer) ]] :=
    [[ e ^ (s_outer) : ty ]],
    ty != NIL(),
    Var{x} <- s,
    Var{x} : ty !.
```

Record Definitions

```
                                                                    s0
let
  type point1 = { x2 : int, y3 : int }     s2
  var p4 := point5{ x6 = 4, y7 = 5 }                                s1
in                                                                  s3
  p8.x9
end
```

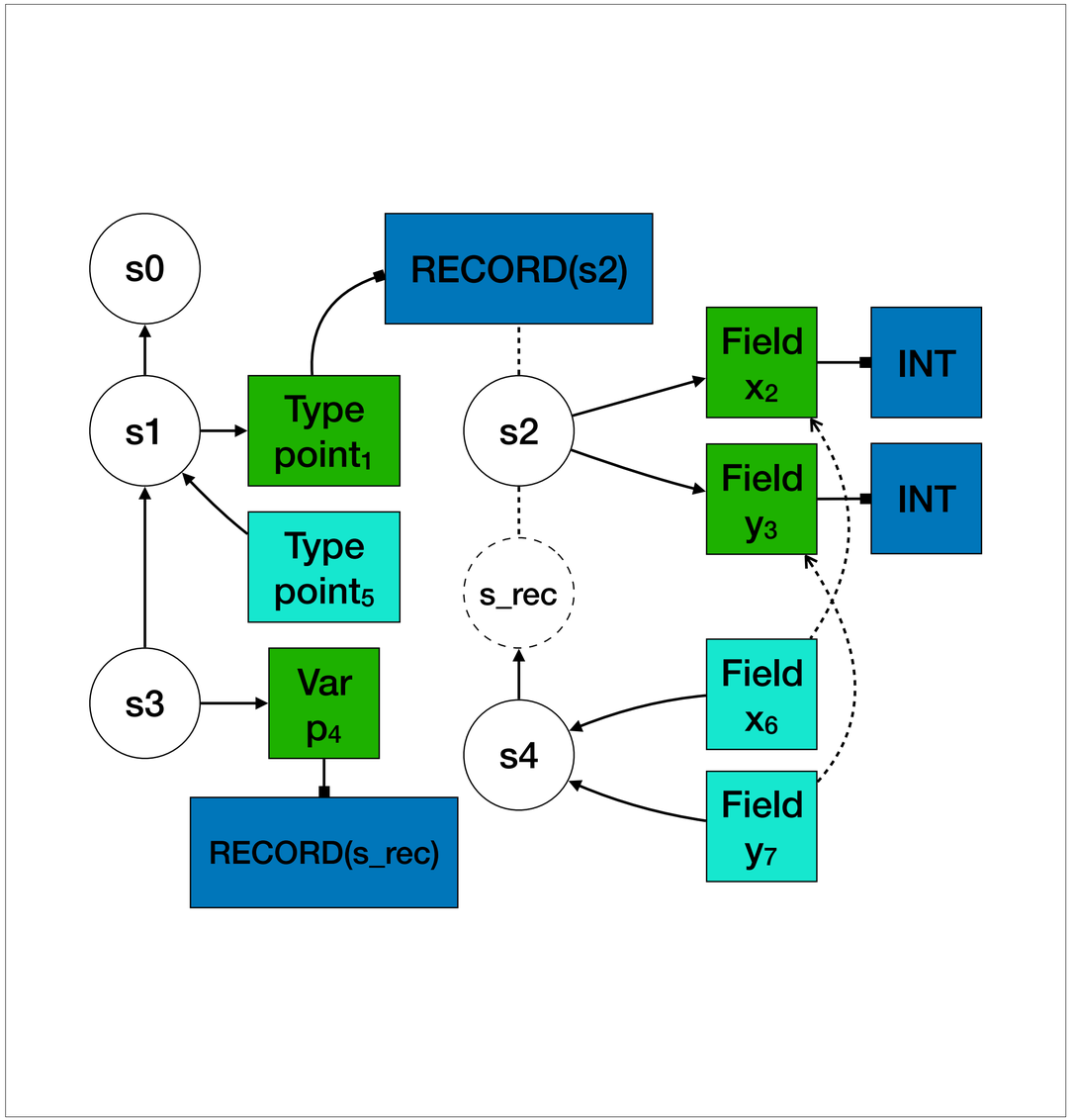
```
[[ RecordTy(fields) ^ (s) : ty ]] :=
    new s_rec,
    ty == RECORD(s_rec),
    NIL() <! ty,
    distinct/name D(s_rec)/Field,
    Map2[[ fields ^ (s_rec, s) ]].

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



Record Creation

```
                                                    s0
let
  type point1 = { x2 : int, y3 : int }     s2
  var p4 := point5{ x6 = 4, y7 = 5 }       s4      s1
in                                                 s3
  p8.x9
end
```

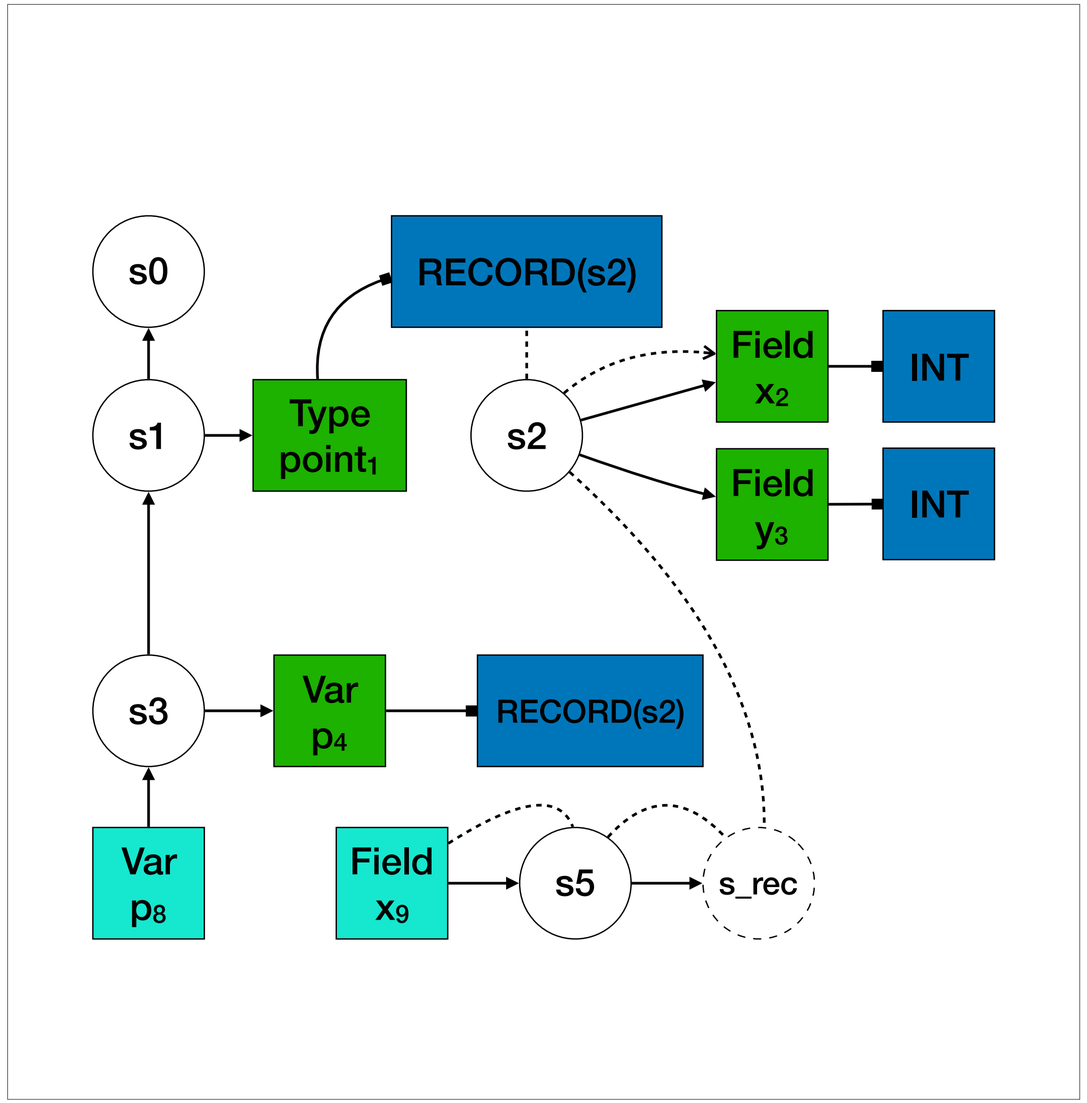
```
[[ r@Record(t, inits) ^ (s) : ty ]] :=
    [[ t ^ (s) : ty ]],
    ty == RECORD(s_rec),
    new s_use, s_use -I-> s_rec,
    D(s_rec)/Field subseteq/name
        R(s_use)/Field,
    distinct/name R(s_use)/Field,
    Map2[[ inits ^ (s_use, s) ]].

[[ InitField(x, e) ^ (s_use, s) ]] :=
    Field{x} -> s_use,
    Field{x} |-> d,
    d : ty1,
    [[ e ^ (s) : ty2 ]],
    ty2 <? ty1.
```



Record Field Access

```
                                                              s0
let
  type point1 = { x2 : int, y3 : int }     s2
  var p4 := point5{ x6 = 4, y7 = 5 }       s4          s1
in                                                     s3
  p8.x9   s5
end
```

```
[[ 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.
```



Full Tiger Specification Online

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

https://github.com/MetaBorgCube/metaborg-tiger/blob/master/org.metaborg.lang.tiger/trans/static-semantics/static-semantics.nabl2