A Constraint Language for Static Semantic Analysis Based on Scope Graphs

Hendrik van Antwerpen
TU Delft, The Netherlands
h.vanantwerpen@tudelft.nl

Pierre Néron
TU Delft, The Netherlands
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University, USA
tolmach@pdx.edu

Eelco Visser
TU Delft, The Netherlands
visser@acm.org

Guido Wachsmuth
TU Delft, The Netherlands
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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language classifications; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.2.6 [*Software Engineering*]: Programming Environments

Keywords Language Specification; Name Binding; Types; Domain Specific Languages; Meta-Theory

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 showed 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 or many declarations (or to none). 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 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, we require 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

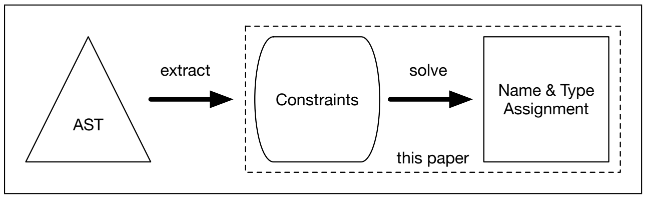


Figure 1. Architecture of our constraint-based approach to static semantic analysis. A language-specific extraction function translates an abstract syntax tree to a set of constraints. The generic constraint solver (independent of the source language), solves the constraints and produces a name and type assignment.

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 name resolution relative to a scope graph. We define a constraint resolution algorithm based on our previous name resolution algorithm, extended to support parameterization by a language-specific policy controlling scope reachability and visibility, combined with a standard unification algorithm. (iv) The constraint language is intended as an internal language for static semantic analysis tools (Fig. 1). Given the abstract syntax tree of a program, a language-specific extractor produces a set of constraints that express the name binding and types of the program. A language-independent solver attempts to find a solution for the set of extracted constraints, and produces a (partial) name and type assignment. Note that the constraint language is not intended as a domain-specific meta-language (such as NaBL [12]) to be used by language designers using a language workbench. Rather, it is intended to be used as an internal language for the implementation of such meta-languages. (v) The application to erroneous programs is work in progress.

Contributions The specific technical contributions of this paper are the following:

- We introduce a constraint notation for the specification of scope graphs and name resolution that is complementary to the description of traditional typing constraints.
- We extend the scope graph framework of [14] with uniqueness and completeness constraints to express properties such as "there are no duplicate declarations in this scope" or "every declared field in this record is initialized."
- We introduce generalized scope graph edge labels to model a wide range of scope combination policies including transitive and non-transitive imports, and non-overriding includes.
- We give a specification for satisfiability of combined sets of name and type resolution constraints.
- We extend the name resolution algorithm of [14] to be parametric over scope reachability and visibility policies defined over (generalized) scope graph edge labels.
- We give an algorithm for solving combined name and type resolution problems and prove that it is sound with respect to the satisfiability specification.

Outline In Section 2, we introduce the constraint language using example programs in a small model language. In Section 3, we formally define the syntax and semantics of the constraint language by defining a satisfaction relation on constraints and an extended resolution calculus. In Section 4 we develop a constraint solver and prove that it is sound with respect to the semantics. In Section 5 we relate this work to previous work by ourselves and others, and discuss limitations and ideas for future work.

2. Constraints for Static Semantics

In this section we introduce our approach to constraint-based name and type resolution. We show how scope graph constraints are used to model name binding and combine them with typing constraints to model type consistency. We illustrate the ideas using LMR (Language with Modules and Records), a small model language that is a variant of the LM (Language with Modules) of [14]. LMR does not aspire to be a real programming language, but is designed to represent typical and challenging name and type resolution idioms.

In the rest of this section we study name and type resolution for a selection of LMR constructs using a series of examples. The full grammar of LMR is defined in Fig. 5 and a constraint extraction algorithm for the entire language is given in Fig. 6. Along the way we gradually introduce the concepts of the constraint language. The full syntax of the constraint language is defined in Fig. 7. Subsequent sections formalize the constraint language and its semantics.

2.1 Declarations and References

We first recall the concepts of the scope graph approach [14], and adapt them to a constraint-based framework. Consider the example in Fig. 2, which shows a simple LMR program with two global declarations (top), and, in the boxes below it, the constraints extracted from it and their solution. Subscripts on expressions and identifiers represent AST positions. Thus, $\mathtt{x}_1$, $\mathtt{x}_4$, and $\mathtt{x}_8$ are different occurrences of the *same* name $\mathtt{x}$. We represent scope graph constraints diagrammatically by the scope graph they specify.

The *nodes* of a scope graph $\mathcal{G}$ represent the three basic notions derived from the program abstract syntax tree (AST): *scopes, declarations, and references*:

- A *scope* is an abstraction of a set of nodes in the AST that behave uniformly with respect to name binding. Scopes are denoted by identifiers drawn from an abstract enumerable set. In a scope graph diagram, scopes are represented by circles with numbers representing their identity, e.g. ① . $\mathcal{S}(\mathcal{G})$ denotes the set of scopes of $\mathcal{G}$.
- A *declaration* is an occurrence of an identifier that introduces a name. We write x_i^{D} for the declaration of name x at position i in the program. We omit the position i when it is unimportant in the context. In diagrams, a declaration is represented by a box with an *incoming arrow*, e.g. $\rightarrow\boxed{\mathtt{x}_1}$. $\mathcal{D}(\mathcal{G})$ denotes the set of declarations of $\mathcal{G}$.
- A *reference* is an occurrence of an identifier referring to a declaration. We write x_i^{R} for a reference with name x at position i. Again, we sometimes omit the position i. In diagrams, a reference is represented by a box with an outgoing arrow, e.g. $\leftarrow\boxed{\mathtt{x}_4}$. $\mathcal{R}(\mathcal{G})$ denotes the set of references of $\mathcal{G}$.

Scope Graph Constraints The *edges* of a scope graph determine the connections between scopes, declarations, and references. Edges are specified directly by means of scope graph constraints (C^G in the grammar of Fig. 7), where the ground terms D, R, and S represent declarations, references, and scopes, respectively. For now, we only consider the two basic edges that connect declarations and references to scopes:

- A *declaration constraint* $s \longrightarrow x^{\mathsf{D}}$ specifies that declaration x^{D} belongs to scope s. Graphically: $\textcircled{s} \rightarrow \boxed{x}$.
- A *reference constraint* $x^{\mathsf{R}} \longrightarrow s$ specifies that reference x^{R} belongs to scope s. Graphically: $\boxed{x} \rightarrow \textcircled{s}$.

The "solution" to a set of scope graph constraints is a *well-formed* scope graph, i.e. one in which each declaration and reference belongs to (is connected by an edge with) exactly one scope. Note that the existence of nodes (declarations, references, and scopes) of the scope graph is specified implicitly by their appearance in an edge constraint. For convenience, we sometimes write $\mathcal{S}c(x^{\mathsf{D}}) = s$ for $s \longrightarrow x^{\mathsf{D}}$ and $\mathcal{S}c(x^{\mathsf{R}}) = s$ for $x^{\mathsf{R}} \longrightarrow s$. We define by comprehension the sets of declarations and references belonging to a scope s, as $\mathcal{D}(s) = \{x^{\mathsf{D}} \mid \mathcal{S}c(x^{\mathsf{D}}) = s\}$ and $\mathcal{R}(s) = \{x^{\mathsf{R}} \mid \mathcal{S}c(x^{\mathsf{R}}) = s\}$. In most contexts, constraints and derived notations are implicitly parameterized by the scope graph under consideration; when they need to be explicitly parameterized by a scope graph $\mathcal{G}$, we use a subscript notation (e.g. $\mathcal{D}_{\mathcal{G}}(s)$).

Resolution Constraints The basic semantic intuition behind scope graphs is that a reference resolves to a declaration iff there is a path from the reference node to the declaration node. In this case we say that the declaration is *visible* from the reference. Resolution constraints (C^{Res} in the grammar) represent requirements on successful name resolution:

- A *resolution constraint* $\mathrm{R} \mapsto \mathrm{D}$ specifies that a given reference must resolve to a given declaration. Typically, the declaration is specified as a declaration variable δ. For example, in Fig. 2 the constraints $\mathrm{x}_4^{\mathsf{R}} \mapsto \delta_4$ and $\mathrm{x}_8^{\mathsf{R}} \mapsto \delta_8$ require that references $\mathrm{x}_4^{\mathsf{R}}$ and $\mathrm{x}_8^{\mathsf{R}}$ resolve to (as yet unknown) declarations δ_4 and δ_8, respectively.

A solution to a set of resolution constraints is a substitution mapping each declaration variable to a declaration, such that applying this substitution to the constraints generates valid resolutions according to the scope graph resolution calculus (which we formalize in Section 3). In Fig. 2, since the only paths starting at $\mathrm{x}_4^{\mathsf{R}}$ and $\mathrm{x}_8^{\mathsf{R}}$ both end at declaration $\mathrm{x}_1^{\mathsf{D}}$, the (sole) solution to these constraints is a substitution mapping both δ_4 and δ_8 to $\mathrm{x}_1^{\mathsf{D}}$. Applying this substitution yields the valid resolutions $\mathrm{x}_4^{\mathsf{R}} \longmapsto \mathrm{x}_1^{\mathsf{D}}$ and $\mathrm{x}_8^{\mathsf{R}} \longmapsto \mathrm{x}_1^{\mathsf{D}}$.

In addition to constraints about the resolution of references, C^{Res} also includes constraints on properties of *name collections* N, which are multisets of identifiers. For now we only consider the uniqueness constraint:

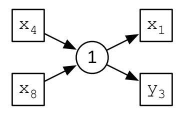
- A *uniqueness constraint* !N specifies that a given name collection N contains no duplicates.
- A *declaration name collection* $\overline{\mathcal{D}}(s)$ is obtained by projecting the identifiers from the set of declarations in scope s.

Thus, for example, in Fig. 2 the constraint $!\overline{\mathcal{D}}(1)$ requires that scope $\textcircled{1}$ should have no duplicate declarations. These types of constraints are satisfied when the property they specify holds.

Typing Constraints Typing constraints (C^{Ty}) represent requirements for type consistency of the program:

```
def x_1 = 1_2
def y_3 = (if (x_4 == 0_5)_6 then 3_7 else x_8)_9
```

Scope graph constraints



Resolution constraints

$$\begin{gathered} \mathrm{x}_4^{\mathsf{R}} \mapsto \delta_4 \\ \mathrm{x}_8^{\mathsf{R}} \mapsto \delta_8 \\ !\overline{\mathcal{D}}(1) \end{gathered}$$

Typing constraints

$$\begin{array}{ll} \mathrm{x}_1^{\mathsf{D}} : \tau_2 & \tau_2 \equiv \mathit{Int} \\ \mathrm{y}_3^{\mathsf{D}} : \tau_9 & \tau_9 \equiv \tau_7 \\ \tau_9 \equiv \tau_8 & \delta_4 : \tau_4 \\ \delta_8 : \tau_8 & \tau_6 \equiv \mathit{Bool} \\ \tau_4 \equiv \mathit{Int} & \tau_5 \equiv \mathit{Int} \\ \tau_7 \equiv \mathit{Int} & \end{array}$$

Solution

$$\begin{gathered} \delta_4 = \delta_8 = \mathrm{x}_1^{\mathsf{D}} \\ \tau_2 = \tau_4 = \tau_5 = \mathit{Int} \\ \tau_7 = \tau_8 = \tau_9 = \mathit{Int} \\ \tau_6 = \mathit{Bool} \end{gathered}$$

Figure 2. Declarations and references in global scope

- A *type declaration constraint* D : T associates a type with a declaration. This constraint is used in two flavors: associating a type variable (τ) with a concrete declaration, or associating a type variable with a declaration variable. In Fig. 2, the constraints $\mathrm{x}_1^{\mathsf{D}} : \tau_2$ and $\mathrm{y}_3^{\mathsf{D}} : \tau_9$ associate distinct type variables with declarations $\mathrm{x}_1^{\mathsf{D}}$ and $\mathrm{y}_3^{\mathsf{D}}$. (For ease of reading, we choose type variable names corresponding to subexpression label numbers.) The constraint $\delta_4 : \tau_4$ requires the type of the declaration to which $\mathrm{x}_4^{\mathsf{R}}$ resolves to be the same as the type τ_4 of the reference considered as an expression.
- A *type equality constraint* $\mathrm{T} \equiv \mathrm{T}$ specifies that two types should be equal. In Fig. 2, the constraint $\tau_2 \equiv \mathit{Int}$ arises from the constant expression 1_2, and the constraint $\tau_4 \equiv \mathit{Int}$ arises from the fact that the == operator takes integer operands. The constraint $\tau_6 \equiv \mathit{Bool}$ arises in two ways, from the fact that == returns a Boolean and the fact that **if** requires one; since constraints should be thought of as a set, we list each distinct constraint only once.

A solution to a set of typing constraints is a substitution on declaration and type variables that satisfies all the constraints. For example, the substitution for τ_9 can be deduced either from the constraints $\tau_9 \equiv \tau_7$ and $\tau_7 \equiv \mathit{Int}$, or from the constraints $\tau_9 \equiv \tau_8$, $\tau_2 \equiv \mathit{Int}$ and the unification of τ_8 and τ_2 (via $\delta_8 = \mathrm{x}_1^{\mathsf{D}}$).

Note that for a program to be both well-bound and well-typed, we need to find a *single* substitution on declaration and type variables that allows both resolution and typing constraints to be satisfied simultaneously. In this simple example, it is clear that the declaration variables are determined solely by the resolution constraints, but this will not always be the case in general.

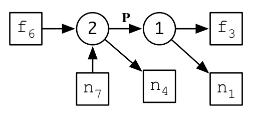
2.2 Lexical Scope

Only very trivial programs have just a single scope. The left part of Fig. 3 shows an LMR example that illustrates nested lexical scopes. Scope graphs use edges *between scopes* to model inclusion of the (visible) declarations in one scope in another. They can be used to model lexical nesting or direct import of all the names from one scope into another, according to the *label* on the edge.

- A *direct edge constraint* $s_1 \xrightarrow{l} s_2$ specifies a direct l-labeled edge from scope s_1 to s_2. (Graphically: $\textcircled{s_1} \xrightarrow{l} \textcircled{s_2}$.) The general meaning of such an edge is that the declarations visible in s_2 are also visible in s_1. Or, following the direction of the arrow, that a reference in s_1 can be resolved by searching for a declaration in s_2.

```
def n_1 = true_2
def f_3 = (
    fun (n_4:Int_5){
      f_6(n_7)
    }_8
)_9
```

Scope graph constraints



Resolution Constraints

$f_6^{\mathsf{R}} \mapsto \delta_6$ $\quad n_7^{\mathsf{R}} \mapsto \delta_7$

$!\overline{\mathcal{D}}(1)$ $\quad !\overline{\mathcal{D}}(2)$

Typing constraints

$n_1^{\mathsf{D}} : \tau_2$ $\quad \tau_2 \equiv Bool$

$f_3^{\mathsf{D}} : \tau_9$ $\quad \tau_9 \equiv Fun[\tau_5,\tau_8]$

$n_4^{\mathsf{D}} : \tau_5$ $\quad \tau_5 \equiv Int$

$\delta_6 : \tau_6$ $\quad \tau_6 \equiv Fun[\tau_7,\tau_8]$

$\delta_7 : \tau_7$

Solution

$\delta_6 = f_3^{\mathsf{D}}$ $\quad \delta_7 = n_4^{\mathsf{D}}$

$\tau_2 = Bool$ $\quad \tau_8 = t_0$

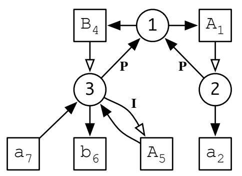
$\tau_5 = \tau_7 = Int$

$\tau_6 = \tau_9 = Fun[Int,t_0]$

where t_0 *is any fixed arbitrary type*

```
module A_1 {
   def a_2 = 4_3
}
module B_4 {
  import A_5
  def b_6 = a_7
}
```

Scope graph constraints



Resolution constraints

$a_7^{\mathsf{R}} \mapsto \delta_7$ $\quad !\overline{\mathcal{D}}(1)$

$!\overline{\mathcal{D}}(2)$ $\quad !\overline{\mathcal{D}}(3)$

Typing constraints

$a_2^{\mathsf{D}} : \tau_3$ $\quad \tau_3 \equiv Int$

$b_6^{\mathsf{D}} : \tau_7$ $\quad \delta_7 \ : \ \tau_7$

Solution

$\delta_7 = a_2^{\mathsf{D}}$

$\tau_3 = Int$ $\quad \tau_7 = Int$

Figure 3. Lexical scope and module imports.

In the left part of Fig. 3, scope ② — corresponding to the body of the **fun** — is nested within the program global scope ①, which is expressed by the scope edge constraint $② \xrightarrow{\mathbf{P}} ①$. This edge is labeled **P** for "parent;" we will see other possible labels shortly. A resolution path starting from a reference may traverse a **P** edge to find a matching declaration, e.g. reference f_6^{R} resolves to f_3^{D}. However, in order to model shadowing of outer declarations by inner ones, paths that traverse fewer (or no) **P** edges are preferred, so reference n_7^{R} resolves to declaration n_4^{D} rather than to n_1^{D}.

The kinds of typing constraints generated by this example are the same as those from the previous example. Note that the solution to the typing constraints leaves f's result type unspecified (since it is never used).

2.3 Imports

In addition to lexical scope, many programming languages provide features for making declarations in scopes selectively available 'at a distance'. Examples of such constructs are modules with imports in ML and classes with inheritance in Java. To model such features, scope graphs provide *associated scopes* and *imports*.

Associated Scope The essence of module-like constructs is that they encapsulate a collection of declarations and make these available through import of the module. That requires an association between the encapsulated declarations and the declaration of the module, which is modeled by *associated scopes*:

- An *association constraint* $x^{\mathsf{D}} \longrightarrow\!\triangleright s$ specifies s as the *associated scope* of declaration x^{D}. Associated scopes can be used to connect the declaration (e.g. a module) of a collection of names to the scope declaring those names (e.g. the body of a module). Graphically: $\boxed{x} \longrightarrow\!\triangleright ⓢ$.

The LMR program in the right part of Fig. 3 consists of two *modules* A_1 and B_4 and an import of the former into the latter. The declarations in these modules are contained in ② and ③. Each of these scopes is *associated* with the corresponding declaration of the name of the module, which is represented in a scope graph diagram with an open arrow, e.g. $\boxed{A_1} \longrightarrow\!\triangleright ②$. These scopes are also child scopes of the program global scope ①.

Imports A *nominal import* makes the declarations in an associated scope visible in another, not necessarily lexically related, *target scope*. A nominal import is represented by (1) a regular reference to the name of the scope being imported, and (2) an import edge of that name into the target scope:

- A *nominal edge constraint* $s \overset{l}{\longrightarrow\!\triangleright} x^{\mathsf{R}}$ specifies a nominal l-labeled edge from scope s to reference x^{R}. (Graphically: $ⓢ \overset{l}{\longrightarrow\!\triangleright} \boxed{x}$) Such an edge makes visible in s all declarations that are visible in the associated scope of the declaration to which x^{R} resolves, according to the label on the edge.

For example, **import** A_5 is represented by the reference A_5^{R} in scope ③ and an import arrow $③ \overset{\mathbf{I}}{\longrightarrow\!\triangleright} \boxed{A_5}$. It is also possible to import the declarations of another scope directly, using an (ordinary) nameless edge; this feature is used in the next sub-section.

Resolving through Imports Name resolution in the presence of associated scopes and imports proceeds as follows. If a scope S_1 contains an import x_i^{R}, which resolves to a declaration x_j^{D} with associated scope S_2, then all declarations in S_2 are reachable in S_1. Thus, in the example, reference a_7^{R} resolves to declaration a_2^{D} since the import A_5^{R} resolves to declaration A_1^{D}, and the associated scope ② of A_1^{D} contains declaration a_2^{D}. Note that the resolution calculus is parameterized by the policy used to disambiguate conflicting resolutions. Here we use a default policy that prefers imported declarations over declarations in parents; alternatives are discussed in Section 3.4.

2.4 Type-Dependent Name Resolution

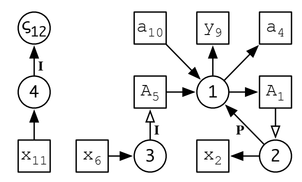
So far, we have seen how to use resolution constraints to express the dependence of type resolution on name resolution. However, for some language constructs the resolution of a name to its declaration depends on the type of another expression. For example, in a field access expression e.f, in order to resolve the field f, one first needs to find the type of the expression e and then to look for f in the scope associated with the type. This scheme induces dependencies on type resolution, not only from name resolution but also from scope graph construction (one does not know in which scope the reference f lies). We model such *type-dependent name resolution* by using scope graph constraints with scope variables. The examples in Fig. 4 illustrate the approach.

Field Declaration and Initialization Before we can study field access proper, we need to consider modeling of record types, field declarations, and record initialization. We identify each record type by the declaration of the record name in its type definition, e.g. $Rec(A_1^{\mathsf{D}})$. We model the fields of a record type definition as declarations (here just x_2^{D}) in a scope (here, scope ②) associated with the record type name declaration A_1^{D}. The resolution constraint $!\overline{\mathcal{D}}(2)$ forbids duplicate field names.

To construct a new record of a declared record type (e.g. A_1^{D}), we create a new parentless scope (here, scope ③) which imports the field names of the record by importing (the associated scope of) the record declaration (via a reference to the name of the type, here

$\textbf{record}\ \mathtt{A}_1\ \{\ \mathtt{x}_2\ :\ \textbf{Int}_3\ \}$
$\textbf{def}\ \mathtt{a}_4\ =\ (\ \textbf{new}\ \mathtt{A}_5\{\mathtt{x}_6\mathtt{=1}_7\}\ )_8$
$\textbf{def}\ \mathtt{y}_9\ =\ (\ \mathtt{a}_{10}.\mathtt{x}_{11}\ )_{12}$

Scope graph constraints



Resolution constraints

$\mathtt{A}^{\mathsf{R}}_5 \mapsto \delta_8$ $\quad$ $\mathtt{x}^{\mathsf{R}}_6 \mapsto \delta_6$
$\mathtt{a}^{\mathsf{R}}_{10} \mapsto \delta_{10}$ $\quad$ $\mathtt{x}^{\mathsf{R}}_{11} \mapsto \delta_{11}$
$!\overline{\mathcal{D}}(1)$ $\quad$ $!\overline{\mathcal{D}}(2)$
$\overline{\mathcal{V}}(3) \approx \overline{\mathcal{R}}(3)$ $\quad$ $\delta_{12} \rightsquigarrow \varsigma_{12}$

Typing constraints

$\mathtt{x}^{\mathsf{D}}_2 : \tau_3$ $\quad$ $\tau_3 \equiv \mathit{Int}$
$\mathtt{a}^{\mathsf{D}}_4 : \tau_8$ $\quad$ $\tau_8 \equiv \mathit{Rec}(\delta_8)$
$\mathtt{y}^{\mathsf{D}}_9 : \tau_{12}$ $\quad$ $\delta_6 : \tau_6$
$\delta_{10} : \tau_{10}$ $\quad$ $\delta_{11} : \tau_{12}$
$\tau_7 \equiv \tau_6$ $\quad$ $\tau_7 \equiv \mathit{Int}$
$\tau_{10} \equiv \mathit{Rec}(\delta_{12})$

Solution

$\delta_6 = \delta_{11} = \mathtt{x}^{\mathsf{D}}_2$
$\delta_8 = \delta_{12} = \mathtt{A}^{\mathsf{D}}_1$
$\delta_{10} = \mathtt{a}^{\mathsf{D}}_4$ $\quad$ $\varsigma_{12} = 2$
$\tau_3 = \tau_6 = \tau_7 = \tau_{12} = \mathit{Int}$
$\tau_8 = \tau_{10} = \mathit{Rec}(\mathtt{A}^{\mathsf{D}}_1)$

Figure 4. Field access.

$\mathtt{A}^{\mathsf{D}}_5$). We then process field initializers by putting references to the field names (here just $\mathtt{x}^{\mathsf{R}}_6$) into this new scope; these references can only resolve to the field declarations.

In order to check that each field of a record type is initialized, we use the following additional kinds of name collections and constraints:

- A *reference name collection* $\overline{\mathcal{R}}(s)$ denotes the multiset of reference identifiers of scope s.
- A *visible name collection* $\overline{\mathcal{V}}(s)$ denotes the multiset of declaration identifiers that are visible from scope s (i.e., would be visible from a reference to the declared identifier in s).
- A *subset constraint* $\mathrm{N} \subsetapprox \mathrm{N}$ specifies that one name collection is included in another.
- An *iso constraint* $\mathrm{N}_1 \approx \mathrm{N}_2$ is syntactic sugar for $\mathrm{N}_1 \subsetapprox \mathrm{N}_2 \wedge \mathrm{N}_2 \subsetapprox \mathrm{N}_1$ and specifies that two name collections are isomorphic.

Thus, the constraint $\overline{\mathcal{V}}(3) \approx \overline{\mathcal{R}}(3)$ requires that the set of visible field declarations $\overline{\mathcal{V}}(3)$ (the declarations *visible* in scope $\textcircled{3}$) is isomorphic to the set of initializers $\overline{\mathcal{R}}(3)$ (the references in $\textcircled{3}$).

Field Access Now we consider the field access $\mathtt{a}^{\mathsf{R}}_{10}.\mathtt{x}^{\mathsf{R}}_{11}$ at subexpression 12 in Fig. 4. The reference $\mathtt{x}^{\mathsf{R}}_{11}$ is a field access in the record value of $\mathtt{a}^{\mathsf{R}}_{10}$. Thus, $\mathtt{x}^{\mathsf{R}}_{11}$ should be resolved in a scope containing (just) the declarations for the field names, i.e. the associated scope of the *type* of the $\mathtt{a}^{\mathsf{R}}_{10}$, namely $\textcircled{2}$. Once again, we create a parentless scope $\textcircled{4}$ and add the field being accessed (here $\mathtt{x}^{\mathsf{R}}_8$) as a reference in that scope. However, in this case we do not know at constraint extraction time that $\textcircled{2}$ is the correct scope to import, because we do not know the type of $\mathtt{a}^{\mathsf{R}}_{10}$. *That is, the name resolution of* $\mathtt{x}^{\mathsf{R}}_{11}$ *depends on the type resolution of* $\mathtt{a}^{\mathsf{R}}_{10}$.

To model this we proceed as follows. We create a new *scope variable* ς_{12} that acts as a placeholder for the scope that we want to import into scope $\textcircled{4}$. We add a direct edge constraint $\textcircled{4} \xrightarrow{\mathbf{I}} \textcircled{\varsigma_{12}}$, this time labeled with **I** rather than **P**, which makes the resolution process more eager to follow the edge (see Section 3.4 for details). We also have the usual constraints $\mathtt{a}^{\mathsf{R}}_{10} \mapsto \delta_{10}$ and $\delta_{10} : \tau_{10}$ corresponding to reference $\mathtt{a}^{\mathsf{R}}_{10}$. And we have the constraint $\tau_{10} \equiv \mathit{Rec}(\delta_{12})$ for some unknown record type declaration δ_{12} because of the use of $\mathtt{a}^{\mathsf{R}}_{10}$ in the field position of a field access. To make the connection between the declaration of the record type and the placeholder scope, we use an association constraint:

- An *association constraint* $\mathrm{D} \rightsquigarrow \mathrm{S}$ specifies that a given declaration has a given associated scope.

Specifically, we use $\delta_{12} \rightsquigarrow \varsigma_{12}$ to say that ς_{12} must be the associated scope of δ_{12}.

Solving these constraints will lead to a solution for ς_{12} — in this case the associated scope of $\mathtt{A}^{\mathsf{D}}_1$, scope $\textcircled{2}$ — such that the appropriate scope can be imported into scope $\textcircled{4}$. After that, $\mathtt{x}^{\mathsf{R}}_{11}$ can be resolved as usual to the corresponding field declaration $\mathtt{x}^{\mathsf{D}}_2$, yielding its type $\tau_3 \equiv \mathit{Int}$.

With As a further variant, we discuss an expression form inspired by the **with** statement in the Pascal language. In the expression `with e do e'`, `e` should be a record-valued expression; the field names of the record are added to the lexical environment of `e'`. That is, a variable reference `x` in `e'` will be interpreted as a field of the record value when the record has indeed a field with name `x`; otherwise the variable is considered as a regular reference in the enclosing lexical context. Static resolution again requires resolving variables in `e'` in the associated scope of the record type of `e`, but this time also allowing resolution to the enclosing lexical scope. Replacing `(a.x)` by `(with a do x)` in the code of Fig. 4 produces identical constraints, with the addition of a scope graph edge $\textcircled{4} \xrightarrow{\mathbf{P}} \textcircled{1}$.

This concludes the informal explanation-by-example of the constraint language and its application to LMR. A constraint extraction algorithm for the full LMR language is given in Fig. 6, but we do not discuss this in detail. Instead, in the next sections we formalize the syntax and semantics of the constraint language and discuss the definition of a resolution algorithm based on the semantics.

3. Syntax and Semantics of Constraints

In this section we formally define the syntax of the constraint language and its declarative semantics.

3.1 Syntax

Fig. 7 defines the full syntax of the constraint language. Constraints are divided into three categories: Scope graph constraints C^{G} specify a scope graph which defines the binding structures of the program. Resolution constraints $\mathrm{C}^{\mathit{Res}}$ describe requirements for all program names to be properly resolved and, where appropriate, to be unique or complete. Typing constraints C^{Ty} describe requirements for the program to be well-typed. The informal meaning of each constraint form was described by a bulleted definition in Section 2. Constraints can be combined using conjunction ($\mathrm{C} \wedge \mathrm{C}$) and True represents the trivially satisfiable constraint.

A *ground* constraint is one having no variables. A scope graph is *ground* if it is specified by a set of ground scope graph constraints; otherwise it is *incomplete*.

The constraint language is parameterized by a family of type constructors $c \in C_{\mathcal{T}}$ and a set of labels $l \in \mathcal{L}$. We describe the former here and the latter in Section 3.4.

Type Constructors Types in T are either type variables τ or type constructor applications $c(\mathrm{T}, ..., \mathrm{T})$ with $c \in C_{\mathcal{T}}$, a set of language-specific type constructors. Each constructor c has an associated arity $c :: n$. For example, *Int* and *Bool* are type constructors with arity 0 and *Fun* is a type constructor with arity 2. Well-formed constraints respect the arity of the type constructors.

To represent user-defined types, such as classes in object-oriented languages or algebraic data types in functional languages, a type constructor can also include the scope graph declaration corresponding to the type definition. For example, record types in LMR are represented by $\mathit{Rec}(d)$ with d a type name declaration in

```
prog   =  decl*
decl   =  module id { decl* }
       |  import id
       |  def bind
       |  record id { fdecl* }
fdecl  =  id : ty
ty     =  Int
       |  Bool
       |  id
       |  ty → ty
exp    =  int
       |  true
       |  false
       |  id
       |  exp ⊕ exp
       |  if exp then exp else exp
       |  fun ( id : ty ) { exp }
       |  exp exp
       |  letrec tbind in exp
       |  new id { fbind* }
       |  with exp do exp
       |  exp . id
bind   =  id = exp
       |  tbind
tbind  =  id : ty = exp
fbind  =  id = exp
```

Figure 5. Syntax of LMR.

For simplicity, we describe the algorithm as operating over LMR's concrete syntax. The algorithm is defined by a family of functions indexed by syntactic category (*decl*, *exp*, etc.). Each function takes a syntactic component and possibly one or more auxiliary parameters, and returns a constraint, possibly involving one or more fresh variables or new scope identifiers. Functions are defined by a set of rules, one for each possible syntactic form in the category. For example, $\llbracket - \rrbracket^{exp}_{s,t}$ has twelve rules, and is parameterized by the scope s in which identifier references within the expression are to go and the expected type t of the expression. We use the notation $\llbracket - \rrbracket^{c^*}$ on sequences of items of syntactic category c to mean the result of applying $\llbracket - \rrbracket^{c}$ to each item and returning the conjunction of the resulting constraints, or True for the empty sequence.

$$
\begin{array}{rcl}
\llbracket ds \rrbracket^{prog} & := & !\overline{\mathcal{D}}(s) \wedge \llbracket ds \rrbracket^{decl^*}_{s} \\[1ex]
\llbracket \mathbf{module}\ x_i\ \{\ ds\ \} \rrbracket^{decl}_{s} & := & s \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} \rightarrowtriangle s' \wedge s' \xrightarrow{\mathsf{P}} s \wedge !\overline{\mathcal{D}}(s') \wedge \llbracket ds \rrbracket^{decl^*}_{s'} \\
\llbracket \mathbf{import}\ x_i \rrbracket^{decl}_{s} & := & x_i^{\mathsf{R}} \longrightarrow s \wedge s \overset{\mathsf{I}}{\rightarrowtriangle} x_i^{\mathsf{R}} \\
\llbracket \mathbf{def}\ b \rrbracket^{decl}_{s} & := & \llbracket b \rrbracket^{bind}_{s} \\
\llbracket \mathbf{record}\ x_i\ \{\ fs\ \} \rrbracket^{decl}_{s} & := & s \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} \rightarrowtriangle s' \wedge s' \xrightarrow{\mathsf{P}} s \wedge !\overline{\mathcal{D}}(s') \wedge \llbracket fs \rrbracket^{fdecl^*}_{s,s'} \\[1ex]
\llbracket x_i = e \rrbracket^{bind}_{s} & := & s \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} : \tau \wedge \llbracket e \rrbracket^{exp}_{s,\tau} \\
\llbracket x_i\ \mathbf{:}\ t = e \rrbracket^{bind}_{s} & := & s \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} : \tau \wedge \llbracket t \rrbracket^{ty}_{s,\tau} \wedge \llbracket e \rrbracket^{exp}_{s,\tau} \\
\llbracket x_i \mathbf{:} t \rrbracket^{fdecl}_{s_r,s_d} & := & s_d \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} : \tau \wedge \llbracket t \rrbracket^{ty}_{s_r,\tau} \\[1ex]
\llbracket \mathbf{Int} \rrbracket^{ty}_{s,t} & := & t \equiv \mathit{Int} \\
\llbracket \mathbf{Bool} \rrbracket^{ty}_{s,t} & := & t \equiv \mathit{Bool} \\
\llbracket t_1 \rightarrow t_2 \rrbracket^{ty}_{s,t} & := & t \equiv \mathit{Fun}[\tau_1,\tau_2] \wedge \llbracket t_1 \rrbracket^{ty}_{s,\tau_1} \wedge \llbracket t_2 \rrbracket^{ty}_{s,\tau_2} \\
\llbracket x_i \rrbracket^{ty}_{s,t} & := & t \equiv \mathit{Rec}(\delta) \wedge x_i^{\mathsf{R}} \longrightarrow s \wedge x_i^{\mathsf{R}} \mapsto \delta \\[1ex]
\llbracket \mathbf{fun}\ (x_i \mathbf{:} t_1)\ \{ e \} \rrbracket^{exp}_{s,t} & := & t \equiv \mathit{Fun}[\tau_1,\tau_2] \wedge s' \xrightarrow{\mathsf{P}} s \wedge !\overline{\mathcal{D}}(s') \wedge s' \longrightarrow x_i^{\mathsf{D}} \\
 & & \wedge\, x_i^{\mathsf{D}} : \tau_1 \wedge \llbracket t_1 \rrbracket^{ty}_{s,\tau_1} \wedge \llbracket e \rrbracket^{exp}_{s',\tau_2} \\
\llbracket \mathbf{letrec}\ bs\ \mathbf{in}\ e \rrbracket^{exp}_{s,t} & := & s' \xrightarrow{\mathsf{P}} s \wedge !\overline{\mathcal{D}}(s') \wedge \llbracket bs \rrbracket^{bind}_{s'} \wedge \llbracket e \rrbracket^{exp}_{s',t} \\
\llbracket n \rrbracket^{exp}_{s,t} & := & t \equiv \mathit{Int} \\
\llbracket \mathbf{true} \rrbracket^{exp}_{s,t} & := & t \equiv \mathit{Bool} \\
\llbracket \mathbf{false} \rrbracket^{exp}_{s,t} & := & t \equiv \mathit{Bool} \\
\llbracket e_1 \oplus e_2 \rrbracket^{exp}_{s,t} & := & t \equiv t_3 \wedge \tau_1 \equiv t_1 \wedge \tau_2 \equiv t_2 \wedge \llbracket e_1 \rrbracket^{exp}_{s,\tau_1} \wedge \llbracket e_2 \rrbracket^{exp}_{s,\tau_2} \\
 & & \textit{(where } \oplus \textit{ has type } t_1 \times t_2 \rightarrow t_3\textit{)} \\
\llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrbracket^{exp}_{s,t} & := & \tau_1 \equiv \mathit{Bool} \wedge \llbracket e_1 \rrbracket^{exp}_{s,\tau_1} \wedge \llbracket e_2 \rrbracket^{exp}_{s,t} \wedge \llbracket e_3 \rrbracket^{exp}_{s,t} \\
\llbracket x_i \rrbracket^{exp}_{s,t} & := & x_i^{\mathsf{R}} \longrightarrow s \wedge x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : t \\
\llbracket e_1\ e_2 \rrbracket^{exp}_{s,t} & := & \tau \equiv \mathit{Fun}[\tau_1,t] \wedge \llbracket e_1 \rrbracket^{exp}_{s,\tau} \wedge \llbracket e_2 \rrbracket^{exp}_{s,\tau_1} \\
\llbracket e\,\mathbf{.}\,x_i \rrbracket^{exp}_{s,t} & := & \llbracket e \rrbracket^{exp}_{s,\tau} \wedge \tau \equiv \mathit{Rec}(\delta) \wedge \delta \rightsquigarrow \varsigma \wedge s' \xrightarrow{\mathsf{I}} \varsigma \wedge \llbracket x_i \rrbracket^{exp}_{s',t} \\
\llbracket \mathbf{with}\ e_1\ \mathbf{do}\ e_2 \rrbracket^{exp}_{s,t} & := & \llbracket e_1 \rrbracket^{exp}_{s,\tau} \wedge \tau \equiv \mathit{Rec}(\delta) \wedge \delta \rightsquigarrow \varsigma \\
 & & \wedge\, s' \xrightarrow{\mathsf{P}} s \wedge s' \xrightarrow{\mathsf{I}} \varsigma \wedge \llbracket e_2 \rrbracket^{exp}_{s',t} \\
\llbracket \mathbf{new}\ x_i\ \{\ bs\ \} \rrbracket^{exp}_{s,t} & := & x_i^{\mathsf{R}} \longrightarrow s \wedge x_i^{\mathsf{R}} \mapsto \delta \wedge s' \overset{\mathsf{I}}{\rightarrowtriangle} x_i^{\mathsf{R}} \\
 & & \wedge\, \llbracket bs \rrbracket^{fbind^*}_{s,s'} \wedge \overline{\mathcal{V}}(s') \approx \overline{\mathcal{R}}(s') \wedge t \equiv \mathit{Rec}(\delta) \\[1ex]
\llbracket x_i = e \rrbracket^{fbind}_{s,s'} & := & x_i^{\mathsf{R}} \longrightarrow s' \wedge x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : \tau \wedge \llbracket e \rrbracket^{exp}_{s,\tau}
\end{array}
$$

Figure 6. Constraint extraction for LMR. Scope names (s) occurring free in rhs of rules are new ground scopes. Type variables (τ), scope variables (ς), and declaration variables (δ) occurring free are fresh variables.

the program; thus, in Fig. 4, the record definition A defines the type $\mathit{Rec}(\mathtt{A}_1^{\mathsf{D}})$.

3.2 Constraint Satisfaction

In our approach, the abstract syntax tree of a program p is reduced by the language-specific extraction function to a constraint $\llbracket p \rrbracket = C^G_p \wedge C^{Res}_p \wedge C^{Ty}_p$ where commutativity and associativity of conjunction let us group the subconstraints into categories.

Our basic approach to defining satisfaction is as follows. First assume that we have only ground constraints. Then we can interpret scope graph constraints C^G directly as a ground scope graph. We next define a satisfiability relation $\models$ by cases on ground resolution constraints C^{Res} and typing constraints C^{Ty} relative to a context $(\mathcal{G}, \psi)$, where $\mathcal{G}$ is a ground scope graph and ψ is a typing environment mapping declarations in $\mathcal{D}(\mathcal{G})$ to unique ground types in T. In particular, resolution constraints are checked against $\mathcal{G}$ using the

$$
\begin{array}{rcl}
\mathrm{C} & := & \mathrm{C}^{G} \mid \mathrm{C}^{Ty} \mid \mathrm{C}^{Res} \mid \mathsf{C} \wedge \mathsf{C} \mid \mathsf{True} \\
\mathrm{C}^{G} & := & \mathrm{R} \longrightarrow \mathrm{S} \mid \mathrm{S} \longrightarrow \mathrm{D} \mid \mathrm{S} \xrightarrow{l} \mathrm{S} \mid \mathrm{D} \rightarrowtriangle \mathrm{S} \mid \mathrm{S} \xrightarrow{l}\!\!\triangleright\, \mathrm{R} \\
\mathrm{C}^{Res} & := & \mathrm{R} \mapsto \mathrm{D} \mid \mathrm{D} \rightsquigarrow \mathrm{S} \mid !\mathrm{N} \mid \mathrm{N} \subsetneqq \mathrm{N} \\
\mathrm{C}^{Ty} & := & \mathrm{T} \equiv \mathrm{T} \mid \mathrm{D} : \mathrm{T} \\
\mathrm{D} & := & \delta \mid x_i^{\mathsf{D}} \\
\mathrm{R} & := & x_i^{\mathsf{R}} \\
\mathrm{S} & := & \varsigma \mid n \\
\mathrm{T} & := & \tau \mid c(\mathrm{T}, ..., \mathrm{T}) \text{ with } c \in C_{\mathcal{T}} \\
\mathrm{N} & := & \overline{\mathcal{D}}(\mathrm{S}) \mid \overline{\mathcal{R}}(\mathrm{S}) \mid \overline{\mathcal{V}}(\mathrm{S})
\end{array}
$$

Figure 7. Syntax of constraints

scope graph resolution calculus (described in Section 3.3). Finally, we apply $\models$ with $\mathcal{G}$ set to C^{G}.

To lift this approach to constraints with variables, we simply apply a multi-sorted substitution ϕ, mapping type variables τ to ground types, declaration variables δ to ground declarations and scope variables ς to ground scopes. Thus, our overall definition of satisfaction for a program p is:

$$\phi(\mathrm{C}_p^{G}), \psi \models \phi(\mathrm{C}_p^{Res}) \wedge \phi(\mathrm{C}_p^{Ty}) \qquad (\diamond)$$

where $\phi(E)$ denotes the application of the substitution ϕ to all the variables appearing in E that are in the domain of ϕ. When the proposition $\diamond$ holds we say that ψ and ϕ *resolve* p.

Resolution and Typing Constraints The $\models$ relation is given by the inductive rules in Fig. 8, where $=$ is the syntactic equality on terms and $\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$ is the resolution relation for graph $\mathcal{G}$. The interpretation of a name collection $[\![\mathrm{N}]\!]_{\mathcal{G}}$ is the multiset defined as follows: $[\![\overline{\mathcal{D}}(S)]\!]_{\mathcal{G}} = \pi(\mathcal{D}_{\mathcal{G}}(S))$, $[\![\overline{\mathcal{R}}(S)]\!]_{\mathcal{G}} = \pi(\mathcal{R}_{\mathcal{G}}(S))$, and $[\![\overline{\mathcal{V}}(S)]\!]_{\mathcal{G}} = \pi(\{x_i^{\mathsf{D}} \mid \exists p, \vdash_{\mathcal{G}} p : S \longmapsto x_i^{\mathsf{D}}\})$ where $\pi(A)$ is the multiset produced by projecting the identifiers from a set A of references or declarations. Given a multiset M, $\mathbf{1}_M(x)$ denotes the multiplicity of x in M.

3.3 Resolution Calculus

The *resolution calculus* defines the *resolution* of a reference to a declaration in a scope graph as a *most specific*, *well-formed* path from reference to declaration through a sequence of edges. A path p is a list of steps representing the atomic scope transitions in the graph. There are three kinds of steps:

- A (direct) edge step $\mathbf{E}(l, S_2)$ is a direct transition from the current scope to the scope S_2. This step records the label of the scope transition that is used.
- A nominal edge step $\mathbf{N}(l, y^{\mathsf{R}}, S)$ requires the resolution of reference y^{R} to a declaration with associated scope S to allow a transition between the current scope and scope S.
- A complete path always ends with a declaration step $\mathbf{D}(x^{\mathsf{D}})$ that stores the declaration the path is leading to.

A path p is a valid resolution in the graph from reference x_i^{R} to declaration x_i^{D} such that $\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$ according to the calculus rules in Fig. 9. These rules all implicitly apply to a fixed graph $\mathcal{G}$, which we omit to avoid clutter. The calculus defines the resolution relation in terms of edges in the scope graph, reachable declarations, and visible declarations. Here $\mathbb{I}$ is the set of *seen imports*, a technical device needed to avoid "out of thin air" anomalies in resolution of nominal imports. We often drop $\mathbb{I}$ from a resolution when it is empty. The $\mathbb{S}$ component that appears in the transitive closure rules is the set of *seen scopes* that is used to prevent cycles in the resolution path of a given reference.

$$\frac{}{\mathcal{G}, \psi \models \mathsf{True}} \quad (\text{C-TRUE})$$

$$\frac{\mathcal{G}, \psi \models C_1 \quad \mathcal{G}, \psi \models C_2}{\mathcal{G}, \psi \models C_1 \wedge C_2} \quad (\text{C-AND})$$

$$\frac{\psi(d) = T}{\mathcal{G}, \psi \models d : T} \quad (\text{C-TYPEOF})$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}}{\mathcal{G}, \psi \models x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}} \quad (\text{C-RESOLVE})$$

$$\frac{d \rightarrowtriangle_{\mathcal{G}} S}{\mathcal{G}, \psi \models d \rightsquigarrow S} \quad (\text{C-SCOPEOF})$$

$$\frac{\forall x, \mathbf{1}_{[\![N]\!]_{\mathcal{G}}}(x) \leq 1}{\mathcal{G}, \psi \models !N} \quad (\text{C-UNIQUE})$$

$$\frac{[\![N_1]\!]_{\mathcal{G}} \subseteq [\![N_2]\!]_{\mathcal{G}}}{\mathcal{G}, \psi \models N_1 \subsetneqq N_2} \quad (\text{C-SUBNAME})$$

$$\frac{t_1 = t_2}{\mathcal{G}, \psi \models t_1 \equiv t_2} \quad (\text{C-EQ})$$

Figure 8. Interpretation of resolution and typing constraints

Resolution paths

$$
\begin{array}{rcl}
s & := & \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{E}(l, S) \mid \mathbf{N}(l, x_i^{\mathsf{R}}, S) \\
p & := & [\,] \mid s \mid p \cdot p \qquad \text{(inductively generated)}
\end{array}
$$

$$[\,] \cdot p = p \cdot [\,] = p$$

$$(p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$$

Well-formed paths

$$WF(p) \Leftrightarrow labels(p) \in \mathcal{E}$$

Visibility ordering on paths

$$\frac{label(s_1) < label(s_2)}{s_1 \cdot p_1 < s_2 \cdot p_2} \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2}$$

Edges in scope graph

$$\frac{S_1 \xrightarrow{l} S_2}{\mathbb{I} \vdash \mathbf{E}(l, S_2) : S_1 \longrightarrow S_2} \quad (E)$$

$$\frac{S_1 \xrightarrow{l}\!\!\triangleright\, y_i^{\mathsf{R}} \quad y_i^{\mathsf{R}} \notin \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}} \quad y_j^{\mathsf{D}} \rightarrowtriangle S_2}{\mathbb{I} \vdash \mathbf{N}(l, y_i^{\mathsf{R}}, S_2) : S_1 \longrightarrow S_2} \quad (N)$$

Transitive closure

$$\frac{}{\mathbb{I}, \mathbb{S} \vdash [\,] : A \twoheadrightarrow A} \quad (I)$$

$$\frac{B \notin \mathbb{S} \quad \mathbb{I} \vdash s : A \longrightarrow B \quad \mathbb{I}, \{B\} \cup \mathbb{S} \vdash p : B \twoheadrightarrow C}{\mathbb{I}, \mathbb{S} \vdash s \cdot p : A \twoheadrightarrow C} \quad (T)$$

Reachable declarations

$$\frac{\mathbb{I}, \{S\} \vdash p : S \twoheadrightarrow S' \quad WF(p) \quad S' \longrightarrow x_i^{\mathsf{D}}}{\mathbb{I} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \quad (R)$$

Visible declarations

$$\frac{\begin{array}{c}\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \\ \forall j, p' (\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))\end{array}}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}} \quad (V)$$

Reference resolution

$$\frac{x_i^{\mathsf{R}} \longrightarrow S \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \quad (X)$$

Figure 9. Resolution calculus from [14] extended for arbitrary edge labels and parameterized with well-formedness predicate *WF* and visibility ordering $<$. Here *label* projects the label from a step and *labels* projects the sequence of labels from a path.

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

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

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

Figure 10. Example reachability and visibility policies by instantiation of path well-formedness and visibility.

3.4 Parameterization

In order to model the name binding features and resolution policies from different programming languages, the scope graph and resolution calculus are parameterized with a set of labels $\mathcal{L}$, a regular expression $\mathcal{E}$ that defines the scope reachability policy, and an order $<$ on the $\mathcal{L}$ (extended with the built-in **D** label) that defines the scope visibility policy. Fig. 9 defines generic predicates derived from these parameters and used in the calculus. The regular expression $\mathcal{E}$ entails a *well-formedness predicate WF* on paths obtained by projecting the sequence of labels from the path and testing it for membership in the language of $\mathcal{E}$. The ordering relation on labels entails an ordering relation on paths using the lexicographic order on the projected label sequences.

Fig. 10 presents several example instantiations for these parameters, encoding different policies. The first policy defines lexical scope in which scopes are transitively linked through parent edges (**P**) and local declarations shadow declarations in parents. The next policy extends lexical scope with non-transitive imports (**I**). The well-formedness predicate allows an optional import at the end of a lexical scope chain, ruling out access to the parents of an imported scope. Further, the policy states that imported declarations shadow declarations in the lexical context. The transitive imports policy extends this by allowing paths with a chain of imports (**TI**). The transitive includes policy is a variation in which local declarations do not shadow included (**Inc**) declarations. The final policy combines three import policies, not providing rules to disambiguate between paths through different kinds of import edges. Thus, a reference that can be resolved through an import *and* an include edge is ambiguous and can be flagged as an error.

4. Resolution Algorithm

In this section, we describe an algorithm for solving constraints in the sense of Section 3.2, i.e. finding ϕ and ψ that satisfy ($\diamond$). Our algorithm works only for a restricted class of generated constraints: all constraints in C_p^G must be ground, except that scope variables ς can appear as targets in direct edge constraints (e.g. $S \xrightarrow{l} \varsigma$). This restriction is met by the constraints generated by the LMR collection algorithm in Section 2. Broader classes of constraints might be useful for other languages; we defer exploration of algorithms that could handle these to future work.

4.1 Variables in Scope Graph Constraints

The basic approach of the algorithm is to interpret the scope graph constraints as a scope graph $\mathcal{G}$ and then use it to resolve resolution and typing constraints using a conventional unification-based algorithm. However, since scope graph constraints can contain variables, we cannot fully define the scope graph before starting constraint resolution, because we do not fully know ϕ. Thus, our algorithm builds ϕ (and Ψ) incrementally. The key idea is that we can solve some resolution and typing constraints even when ϕ is not yet fully defined, in such a way that the solution remains valid as it becomes more defined.

$$R[\mathbb{I}](x^{\mathsf{R}}) := \text{let } (r, s) = Env_{\mathcal{E}}[\{x^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](\mathcal{S}c(x^{\mathsf{R}}))\} \text{ in}$$
$$\begin{cases} \mathsf{U} & \text{if } r = \mathsf{P} \text{ and } \{x^{\mathsf{D}} | x^{\mathsf{D}} \in s\} = \emptyset \\ \{x^{\mathsf{D}} | x^{\mathsf{D}} \in s\} \end{cases}$$

$$Env_{re}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{T}, \emptyset) & \text{if } S \in \mathbb{S} \text{ or } re = \varnothing \\ Env_{re}^{\mathcal{L} \cup \{\mathbf{D}\}}[\mathbb{I}, \mathbb{S}](S) \end{cases}$$

$$Env_{re}^{L}[\mathbb{I}, \mathbb{S}](S) := \bigcup_{l \in Max(L)} \left(Env_{re}^{\{l' \in L | l' < l\}}[\mathbb{I}, \mathbb{S}](S) \lhd Env_{re}^{l}[\mathbb{I}, \mathbb{S}](S) \right)$$

$$Env_{re}^{\mathbf{D}}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{T}, \emptyset) & \text{if } [] \notin re \\ (\mathsf{T}, \mathcal{D}(S)) \end{cases}$$

$$Env_{re}^{l}[\mathbb{I}, \mathbb{S}](S) := \begin{cases} (\mathsf{P}, \emptyset) \text{ if } S_l^{\blacktriangleright} \text{ contains a variable or } IS^l[\mathbb{I}](S) = \mathsf{U} \\ \displaystyle\bigcup_{S' \in \left(IS^l[\mathbb{I}](S) \cup S_l^{\blacktriangleright}\right)} Env_{(l^{-1}re)}[\mathbb{I}, \{S\} \cup \mathbb{S}](S') \end{cases}$$

$$IS^l[\mathbb{I}](S) := \begin{cases} \mathsf{U} & \text{if } \exists y^{\mathsf{R}} \in (S_l^{\triangleright} \setminus \mathbb{I}) \; s.t. \; R[\mathbb{I}](y^{\mathsf{R}}) = \mathsf{U} \\ \{S' \mid y^{\mathsf{R}} \in (S_l^{\triangleright} \setminus \mathbb{I}) \wedge y^{\mathsf{D}} \in R[\mathbb{I}](y^{\mathsf{R}}) \wedge y^{\mathsf{D}} \longrightarrow\!\triangleright S'\} \end{cases}$$

Figure 11. Name Resolution Algorithm

4.2 Name Resolution Algorithm

In order to solve resolution constraints (e.g. $x^{\mathsf{R}} \mapsto \delta$) or to compute the set of visible elements from a scope ($\mathcal{V}(S)$) we need an algorithm that computes the name resolution relation ($x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$) specified by the calculus presented in Section 3.3. We introduced such an algorithm in our prior work [14], but it was specific to a particular set of labels, visibility order, and well-formedness predicate. In this section, we present a generic version of the algorithm that is parameterized by $\mathcal{L}$, $\mathcal{E}$ and $<$ as described in Section 3.4.

Incomplete Scope Graphs A further new requirement on the algorithm is that it can operate on an incomplete scope graph, specified by a set of constraints that may still contains variables as the targets of direct edges. The non-strictly positive premise of the (V) rule of the resolution calculus makes the derivation of a resolution relation from a graph non-monotonic with respect to additions to the graph. For example, suppose that in some graph $\mathcal{G}$ a reference x^{R} in a scope S resolves to declaration x_i^{D} in the parent scope S'. In a bigger graph $\mathcal{G}'$ that also has a declaration $x_{i'}^{\mathsf{D}}$ in S itself, x^{R} will resolve to $x_{i'}^{\mathsf{D}}$, and the old resolution to x_i^{D} will be shadowed. Thus we cannot simply restrict resolution to the complete part of the graph, and expect the results to remain valid as the graph becomes more completely known. Instead, we modify the original algorithm to signal when a result is preliminary.

The Algorithm Fig. 11 defines a resolution algorithm that works on such incomplete scope graphs.The function for resolving a single reference, $R[\mathbb{I}](x^{\mathsf{R}})$, returns either a set of declarations or U (*unknown*) if the reference cannot be resolved in the current graph. Similarly, the environment functions $Env_{\bar{re}}[\mathbb{I}, \mathbb{S}](S)$ return a pair consisting of:

- a result flag, T (*total*) if all declarations visible from S can be computed or P (*partial*) if there are still possible additional resolutions (some scope variables are accessible)

- a set of declarations corresponding to resolutions from scope S that are already certain in this incomplete graph.

When a scope graph contains no variables (i.e. when no partial or unknown flags are raised) the intended behavior of the different functions is the following:

- $R[\mathbb{I}](x^{\mathsf{R}})$ returns the set of declarations to which the reference resolves.
- $Env_{re}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations that are reachable from scope S with a minimal path satisfying the regular expression *re*.
- $Env^{L}_{re}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations visible from S through labels in set L after application of the shadowing policy. Using the label order, the declarations accessible through smaller labels shadow the declarations accessible through larger ones.
- $Env^{\mathbf{D}}_{re}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations accessible from S with a **D** step, i.e. the set of declarations in S.
- $Env^{l}_{re}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations accessible from S with an l-labeled step.
- $IS^{l}[\mathbb{I}](S)$ returns the set of scopes that are accessible through a *nominal edge* by resolving the reference and returning its associated scope.

The algorithm uses the following auxiliary notation and definitions: $\varnothing$ denotes the empty regular expression and given a path p and a regular expression re, $p \in re$ denotes that *labels*(p) is in the language of re. The shadowing operator $\lhd$ on sets of declarations is defined by:

$$D_1 \lhd D_2 \triangleq \left\{x^{\mathsf{D}}_i \mid x^{\mathsf{D}}_i \in D_1 \vee (x^{\mathsf{D}}_i \in D_2 \wedge \nexists j, x^{\mathsf{D}}_j \in D_1\right\}.$$

The shadowing operators on pairs with result flag are defined by:

$$(f_1, D_1) \lhd (f_2, D_2) \triangleq \left\{ \begin{array}{ll} (f_2, D_1 \lhd D_2) & \text{if } f_1 = \mathsf{T} \\ (\mathsf{P}, D_1) & \text{otherwise} \end{array} \right.$$

The union $\cup$ operator over pairs with result flag is defined as:

$$\bigcup_{i\in I}(f_i, D_i) \triangleq \left\{ \begin{array}{ll} (\mathsf{T}, D) & \text{if } \forall i \in I, (f_i = \mathsf{T}) \\ (\mathsf{P}, D) & \text{otherwise} \end{array} \right.$$

where $D = \left\{x^{\mathsf{D}} \in \cup_{i\in I} D_i \mid (\forall j \in I, f_j = \mathsf{T} \vee \exists x^{\mathsf{D}}_k \in D_j)\right\}$. Given a regular expression over labels re and a label l, $l^{-1}re$ denotes the Brzozowski derivative[2] of re by l. Given a partially ordered set L, $Max(L)$ denotes the set of maximal elements of L, i.e. $\{l \in L \mid \nexists l' \in L, l < l'\}$. Given a scope S and a label l, we define:

$$S^{\triangleright}_{l} \triangleq \{x^{\mathsf{R}} \mid S \xrightarrow{l}\!\!\triangleright x^{\mathsf{R}}\} \qquad S^{\blacktriangleright}_{l} \triangleq \{S' \mid S \xrightarrow{l}\!\!\blacktriangleright S'\}$$

4.3 Correctness

We want to prove the correctness of this algorithm with respect to the calculus introduced in Section 3.3. Details of the proofs can be found in the appendix of the extended version [17].

Termination First notice that the algorithm terminates using the lexicographic ordering $(\#(\mathcal{R}(\mathcal{G})\backslash\mathbb{I}), \#(\mathcal{S}(\mathcal{G})\backslash\mathbb{S}), \mathcal{O})$, where $\#(A)$ denotes the cardinality of set A and $\mathcal{O}$ is the following well founded order among the different functions:

$$Env_{re} > Env^{L}_{re} > Env^{l}_{re} > Env^{D}_{re} > IS > R$$

This termination order is used as the induction principle in most of the proofs.

Correctness on ground scope graphs We want to prove that when this algorithm operates on a ground scope graph, it is sound and complete with respect to the calculus presented in Fig. 9. First, it is trivial to prove that on a ground scope graph, the return flag can never be P or U, therefore in this section we forget about the flag and assume that the *Env* functions return a set of declarations.

To prove the correctness of the algorithm, we consider the set of paths that corresponds to the sets of declarations returned by the different functions. Given two sets of scopes $\mathbb{I}$ and $\mathbb{S}$ and a scope S, we define $\mathbb{P}[\mathbb{I},\mathbb{S}](S)$ as:

$$\{p \cdot \mathbf{D}(d) \mid \exists S', \mathbb{I}, \mathbb{S} \cup \{S\} \vdash p : S \twoheadrightarrow S' \wedge \mathcal{S}c(d) = S'\}$$

and given a path p such that $p = p' \cdot \mathbf{D}(d)$, $\Delta(p)$ denotes the declaration d. For a set of paths S, $\Delta(S)$ denotes its corresponding set of declarations $\{\Delta(p) \mid p \in S\}$ and

$$\lhd S \triangleq \{\, p \cdot \mathbf{D}(x^{\mathsf{D}}_i) \in S \mid \forall\, (p' \cdot \mathbf{D}(x^{\mathsf{D}}_j)) \in S,\ \neg\, p' < p\}$$

Given these definitions, we can state the correctness of the algorithm:

Lemma 1 (Resolution algorithm correctness)**.** *On a ground scope graph, we have the following equivalences*

$$R[\mathbb{I}](x^{\mathsf{R}}) = \Delta(\{p \mid \exists d,\ \mathbb{I} \vdash p : x^{\mathsf{R}} \longmapsto d\})$$

$$Env_{re}[\mathbb{I},\mathbb{S}](S) = \begin{cases}\emptyset & \textit{if } S \in \mathbb{S} \\ \Delta(\lhd\{p \cdot \mathbf{D}(d) \in \mathbb{P}[\mathbb{I},\mathbb{S}](S) \mid p \in re\}) & \textit{otherwise}\end{cases}$$

$$Env^{L}_{re}[\mathbb{I},\mathbb{S}](S) = \Delta(\lhd\{p \mid \exists l \in L, p \in \mathbb{P}^{l}_{re}[\mathbb{I},\mathbb{S}](S)\})$$

$$Env^{\mathbf{D}}_{re}[\mathbb{I},\mathbb{S}](S) = \Delta(\{\mathbf{D}(d) \mid [] \in re \wedge \mathcal{S}c(d) = S\})$$

$$Env^{l}_{re}[\mathbb{I},\mathbb{S}](S) = \Delta\left(\left\{ s \cdot p \;\middle|\; \begin{array}{l} label(s) = l\, \wedge \\ \mathbb{I} \vdash s : S \longrightarrow S' \wedge \\ p \in \mathbb{P}_{l^{-1}re}[\mathbb{I}, \mathbb{S} \cup \{S\}](S') \end{array} \right\}\right)$$

$$IS^{l}[\mathbb{I}](S) = \{S' \mid \exists y^{\mathsf{R}}, \mathbb{I} \vdash \mathbf{N}(l, y^{\mathsf{R}}, S') : S \longrightarrow S'\}$$

Proof. The proof is by induction on the termination order of the algorithm. Key observations are that all the considered sets of paths are finite since all the paths are acyclic and if there is a minimal path $s \cdot p$ from scope S with $\mathbb{I} \vdash s : S \longrightarrow S'$ then its tail p is also minimal from S', due to the lexicographic ordering. □

Correctness on incomplete scope graphs We now want to state the general correctness of the algorithm that can operate on incomplete scope graphs. We first extend this definition of resolution as follows. Given an incomplete scope graph $\mathcal{G}$, a reference x^{R} is said to resolve to a declaration x^{D}_i if and only if this resolution is valid in all ground instances of $\mathcal{G}$:

$$\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x^{\mathsf{D}}_i \triangleq \forall\, \phi, \vdash_{\phi(\mathcal{G})} x^{\mathsf{R}} \longmapsto x^{\mathsf{D}}_i \qquad (\blacklozenge)$$

where we write $\vdash_{\mathcal{G}}$ for the resolution relation for graph $\mathcal{G}$ and $\phi(G)$ is the ground scope graph corresponding to the application of substitution ϕ to variables in $\mathcal{G}$. Similarly a declaration x^{D} is visible from scope S in an incomplete scope graph $\mathcal{G}$ if and only if it is visible in all the ground instances.

In order to be able to resolve uniqueness constraints for a program we also want to ensure that an incomplete graph provides *all* the possible resolutions of a given reference. In particular, if a resolution is unique in an incomplete graph, we want to be sure it is unique in all its ground instances. An incomplete graph $\mathcal{G}$ is *stable* for a reference or a scope o, denoted $\mathcal{G} \downarrow o$, if all the resolutions in all its ground instances are the same:

$$\mathcal{G} \downarrow o \triangleq \forall \phi, \phi' \vdash_{\phi(\mathcal{G})} o \longmapsto x^{\mathsf{D}}_i \Rightarrow \vdash_{\phi'(\mathcal{G})} o \longmapsto x^{\mathsf{D}}_i$$

Soundness Given this definition, we can prove that the algorithm on incomplete graphs is correct with respect with the calculus:

Lemma 2. *For any incomplete graph $\mathcal{G}$:*

$$x^{\mathsf{D}}_i \in R_{\mathcal{G}}(x^{\mathsf{R}}) \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x^{\mathsf{D}}_i \wedge \mathcal{G} \downarrow x^{\mathsf{R}}$$

where $R_{\mathcal{G}}(x^{\mathsf{R}})$ denotes the top-level resolution function $R[\emptyset](x^{\mathsf{R}})$ for the graph $\mathcal{G}$.

Lemma 1 states that this property holds when the graph $\mathcal{G}$ is ground. We next prove that if the resolution on an incomplete graph $\mathcal{G}$ terminates with a total flag T then for any graph $\mathcal{G}'$ that is an instance of $\mathcal{G}$, the result is the same.

$$Env_{re}[\mathbb{I},\mathbb{S}](x^{\mathsf{R}})_{\mathcal{G}} = (\mathsf{T},\ D) \Longrightarrow$$
$$Env_{re}[\mathbb{I},\mathbb{S}](x^{\mathsf{R}})_{\mathcal{G}'} = (\mathsf{T},\ D) \quad \text{(i)}$$

Proof. We prove this result along with similar result for all the other functions by induction on the termination order of the algorithm. The fact that the result is total implies that the results of all the recursive calls are also total and this allows us to apply the desired induction hypothesis (when a P or U flag is raised it is always propagated). □

Now we show that the resolution is also correct in the partial case. Let $\mathcal{G}$ be an incomplete scope graph and $\mathcal{G}'$ one of its instances. If a resolution on $\mathcal{G}$ contains a set of declarations for a given name then the resolution on $\mathcal{G}'$ contains the same declarations for this name:

$$Env_{re}[\mathbb{I},\mathbb{S}](S)_{\mathcal{G}} = (\_,\ D) \Longrightarrow Env_{re}[\mathbb{I},\mathbb{S}](S)_{\mathcal{G}'} = (\_,\ D') \Longrightarrow$$
$$\forall x, \{x^{\mathsf{D}} \in D\} \neq \emptyset \Rightarrow \{x^{\mathsf{D}} \in D\} = \{x^{\mathsf{D}} \in D'\} \quad \text{(ii)}$$

Proof. We prove this result along with similar result for all the other functions by induction on the termination order of the algorithm, using (i). □

Finally, we can prove Lemma 2:

Proof. Let $S_x = R_{\mathcal{G}}(x^{\mathsf{R}})$ and pick $x_i^{\mathsf{D}} \in S_x$. To prove that x^{R} resolves to x_i^{D} in $\mathcal{G}$, let $\mathcal{G}'$ be an arbitrary ground instance of $\mathcal{G}$. Using (ii) we have $x_i^{\mathsf{D}} \in R_{\mathcal{G}'}(x^{\mathsf{R}})$ and by Lemma 1 we have $\vdash_{\mathcal{G}'} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$. By $\blacklozenge$, we get that $\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$.

To prove stability, let $\mathcal{G}_1$ and $\mathcal{G}_2$ be ground instances of $\mathcal{G}$. Then using (ii), we have $R_{\mathcal{G}_1}(x^{\mathsf{R}}) = R_{\mathcal{G}_2}(x^{\mathsf{R}}) = S_x$, so by definition we have $\mathcal{G} \downarrow x^{\mathsf{R}}$. □

4.4 Name Collection Computation

This resolution algorithm on partial graphs is used to compute not only resolution of references but also the set of names visible from a given scope. Given an incomplete graph $\mathcal{G}$ and a scope S, we compute name collections as:

$$N_{\mathcal{G}}(\overline{\mathcal{D}}(S)) = \pi(\mathcal{D}_{\mathcal{G}}(S)) \qquad N_{\mathcal{G}}(\overline{\mathcal{R}}(S)) = \pi(\mathcal{R}_{\mathcal{G}}(S))$$
$$N_{\mathcal{G}}(\overline{\mathcal{V}}(S)) = \pi(\{x_i^{\mathsf{D}} \mid \exists E,\ Env_{\mathcal{E}}[\emptyset,\emptyset](S)_{\mathcal{G}} = (\mathsf{T}, E) \wedge x_i^{\mathsf{D}} \in E\})$$

Lemma 3 (Name computation soundness)**.** *If the computation of a name collection E terminates on an incomplete graph $\mathcal{G}$, its results is the semantics of the name collection for any graph $\mathcal{G}'$ that is an instance of $\mathcal{G}$:*
$$N_{\mathcal{G}}(E) = M \Longrightarrow [\![E]\!]_{\mathcal{G}'} = M.$$

4.5 Constraint Solving Algorithm

With this name resolution algorithm in hand, Fig. 12 gives an algorithm to solve the constraint system from Section 3. The algorithm is a non-deterministic rewrite system working over tuples $(C, \mathcal{G}, \psi)$ of a constraint, a scope graph, and a typing environment. It is non-deterministic in the sense that rules may be applied to any atomic constraint in any order considering that $\wedge$ is associative and commutative.

Name resolution introduces ambiguity, since a reference x^{R} may resolve to multiple definitions. If this happens the solver branches, picking a different resolution for x^{R} in every branch. The returned solution is a set of all the $(C, \mathcal{G}, \psi)$ tuples the solver was able to construct. The initial state of the solver is the collected constraint, the (incomplete) scope graph built from the scope graph constraints and an empty typing environment. The algorithm will eliminate clauses from C while instantiating $\mathcal{G}$ and filling ψ. The algorithm terminates when the constraint is empty or no more clauses can be solved. Each rule solves one constraint, possibly updating components of the tuple or applying a substitution to it.

- Rule S-Resolve solves resolution constraints $x^{\mathsf{R}} \mapsto \delta$ using the resolution algorithm from Fig. 11. If a resolution is found, it is substituted for the variable δ. If the scope graph is incomplete, the algorithm might return U, in which case the constraint is left to be solved later.
- Rule S-Assoc solves scope association constraints $x^{\mathsf{D}} \rightsquigarrow \varsigma$ by looking up the scope S associated with ground declaration x^{D} in the scope graph. By substituting S for ς, the scope graph becomes more complete, possibly allowing more references to be resolved.
- Rule S-Equal solves equality constraints $T_1 \equiv T_2$. It uses first order unification $\mathcal{U}(T_1, T_2)$, as described in [1]. The resulting substitution is applied to the tuple.
- Rule S-Unique solves $!N$ constraints by checking that the identifier collection N can be computed and all identifiers in it are distinct. ($\mathbf{1}_A(x)$ is the multiplicity of x in A).
- Rule S-SubName solves $N_1 \subsetneq N_2$ constraints by checking that the identifier collections N_1 and N_2 can be computed and that every identifier in N_1 is also in N_2.
- Rule S-TypeOf solves type assignment constraints $x^{\mathsf{D}} : T$. The rule considers two cases. When no type assignment is declared for x^{D} in ψ (i.e. the first time that it is encountered) the assignment is added to the typing environment ψ. When a type assignment *is* declared (i.e. for subsequent encounters), the type T from the constraint is unified with the type $\psi(x^{\mathsf{D}})$ from the typing environment.

The constraint resolution algorithm is sound with respect to the constraints semantics.

Lemma 4 (Constraint Solver correctness)**.** *If the algorithm produces a solution to a resolution problem then the solution is valid: for all $C, \mathcal{G}, \mathcal{G}', \psi'$:*

$$(C, \mathcal{G}, \emptyset) \longrightarrow^* (\mathsf{True}, \mathcal{G}', \psi') \Longrightarrow$$
$$\exists \phi, \phi(\mathcal{G}) = \mathcal{G}' \wedge \forall \sigma, \sigma\mathcal{G}', \sigma\psi' \models \sigma(\phi(C))$$

Proof. To prove this result we first state some results on the auxiliary unification.

Unification: If $\mathcal{U}(t_1, t_2) = \sigma$ then $\sigma t_1 = \sigma t_2 \wedge \sigma\sigma = \sigma$. See [1] for a survey on unification problem and unification algorithms for first order terms.

Resolution Soundness: Now we can prove the Lemma 4 of the constraint resolution algorithm. We first prove that for each reduction step, if the output is satisfiable, the input is also satisfiable in the same definition-to-type environment:

$$\forall (C_1, \mathcal{G}_1, \psi_1), (C_2, \mathcal{G}_2, \psi_2), (C_1, \mathcal{G}_1, \psi_1) \longrightarrow (C_2, \mathcal{G}_2, \psi_2) \Rightarrow$$
$$\exists \sigma', \sigma'(\mathcal{G}_1) = \mathcal{G}_2 \wedge$$
$$\left(\begin{array}{l} \forall \sigma, (\sigma(\mathcal{G}_2), \sigma\psi_2) \models \sigma(C_2) \Rightarrow \\ \qquad (\sigma\mathcal{G}_2, \sigma\psi_2) \models \sigma\sigma'(C_1) \end{array} \right) \quad (1)$$

The proof of this property is by case analysis on the reduction step. From it, we can prove Lemma 4 by a simple induction on the number of reduction steps. □

$$\begin{array}{rcll}
(x^{\mathsf{R}} \mapsto \delta \land C, \mathcal{G}, \psi) & \longrightarrow & [\delta \mapsto x^{\mathsf{D}}](C, \mathcal{G}, \psi) & \textit{where } x^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}}) \quad \text{(S-RESOLVE)} \\
(x^{\mathsf{D}} \rightsquigarrow \varsigma \land C, \mathcal{G}, \psi) & \longrightarrow & [\varsigma \mapsto S](C, \mathcal{G}, \psi) & \textit{where } x^{\mathsf{D}} \longrightarrow\!\triangleright S \quad \text{(S-ASSOC)} \\
(T_1 \equiv T_2 \land C, \mathcal{G}, \psi) & \longrightarrow & \sigma(C, \mathcal{G}, \psi) & \textit{where } \mathcal{U}(T_1, T_2) = \sigma \quad \text{(S-EQUAL)} \\
(!N \land C, \mathcal{G}, \psi) & \longrightarrow & (C, \mathcal{G}, \psi) & \textit{where } \forall x \in N_{\mathcal{G}}(N), \mathbf{1}_{N_{\mathcal{G}}(N)}(x) = 1 \quad \text{(S-UNIQUE)} \\
(N_1 \subsetneq N_2 \land C, \mathcal{G}, \psi) & \longrightarrow & (C, \mathcal{G}, \psi) & \textit{where } N_{\mathcal{G}}(N_1) \subseteq N_{\mathcal{G}}(N_2) \quad \text{(S-SUBNAME)} \\
(x^{\mathsf{D}} : T \land C, \mathcal{G}, \psi) & \longrightarrow & \begin{cases} (C, \mathcal{G}, \{x^{\mathsf{D}} \mapsto T\} \cup \psi) & \textit{if } x^{\mathsf{D}} \notin \mathrm{dom}(\psi) \\ (\psi(x^{\mathsf{D}}) \equiv T \land C, \mathcal{G}, \psi) & \textit{otherwise} \end{cases} & \text{(S-TYPEOF)} \\
(\mathsf{True} \land C, \mathcal{G}, \psi) & \longrightarrow & (C, \mathcal{G}, \psi) & \text{(S-TRUE)}
\end{array}$$

Figure 12. Constraint solving algorithm

5. Related Work and Discussion

In this section, we discuss the relation of this paper with previous and other related work, and discuss limitations and ideas for future work.

Previous Work The work in this paper is based closely on our previous *theory of name resolution* [14], which we extend and generalize here as follows: (i) a scope graph is now defined directly by a set of constraints; (ii) we generalize the parent relation to an arbitrary labeled direct edge between pairs of scopes, and the named import relation to an arbitrary labeled nominal edge between scopes and references; (iii) we extend the resolution algorithm to handle arbitrary well-formedness conditions expressed as regular expressions over arbitrary sets of path labels and arbitrary visibility orderings on labels; (iv) we support partial resolution over incomplete scope graphs; (v) we add the seen-scopes component, previously an artifact of the resolution algorithm, to the resolution calculus to prevent cyclic resolution paths.

The development of the scope graph framework fits in an ongoing line of research to provide high-level domain-specific support for name binding and type analysis in the Spoofax Language Workbench [10] using the NaBL and TS meta-DSLs [12, 19, 18]. NaBL is a DSL for defining the name binding rules of programming languages by identifying the references, definitions, scopes, and imports in an abstract syntax tree without recourse to environments or symbol tables [12]. TS is a complementary DSL for defining type analysis rules. (The design of TS is not formally published, but it is sketched in [18].) Rules in TS are similar to traditional typing judgments, relating an expression to a type. However, type rules do not have to propagate context information, since that is taken care of by the separate binding rules. TS rules refer to the results of name analysis produced by NaBL (e.g. `definition of x has type t`), and NaBL rules refer to the results of type analysis to achieve type-dependent name resolution. NaBL and TS are implemented by generation of (1) a language-specific AST traversal that generates 'tasks', and (2) a language-independent task engine that evaluates tasks in order to (incrementally) compute a name and type assignment [19]. The resulting name and type analysis engines produce Eclipse IDE support for editor services such as name and type error checking, reference resolution, and code completion.

While NaBL and TS are used in practice to build language definitions with Spoofax, the lack of a solid theoretical foundation was a problem for further development. The aim to verify properties of language definitions [18] requires a semantics that can be explained to a proof assistant such as Coq. In particular, the semantics of notions such as imports and 'subsequent scope' were hard to capture. NaBL has some limitations in its coverage of name binding patterns. For example, it cannot express variations on let bindings such as sequential and parallel let. While the task engine is constraint-like, its type resolution is not based on unification, which entails that TS cannot be used to express languages requiring type inference. The constraint language developed in this paper provides a solid formal basis for developing a new generation of name binding and type specification languages.

Prototype Implementation We have developed a prototype implementation of the constraint solver and applied it in the IDE generated with the Spoofax Language Workbench [10] for the LMR model language used in this paper. However, the prototype does not yet implement the parameterized name resolution algorithm developed in this paper, but uses the fixed policy from [14]. In the prototype implementation, sets of constraints for erroneous programs lead to partial solutions with unsolvable residual constraints that can be translated into error messages in an IDE. However, we have not formalized this; we have only proven the *soundness* of the solver for successful reductions. Furthermore, the implementation is not optimized, nor does it support incremental evaluation of constraints in the sense of the NaBL/TS task engine [19].

Constraints The use of constraints to abstract out type inference problems from the abstract syntax tree is a common approach in implementations and extensions of the Hindley/Milner type system [13] and has been applied to a huge variety of typing features. However, these approaches do not address name resolution using constraints, but rather perform name resolution during constraint collection. For example, in the work of Palsberg et al. [15, 16] on object-oriented type systems, constraints are associated with identifiers, which requires these to be resolved before constraint collection. We believe that our use of constraints to define static name resolution is novel. Instead of performing name resolution during constraint collection, we provide a reusable set of constraints to express name resolution problems, including name resolution for 'remote' names through imports and the interaction between name and type resolution in type-dependent name resolution.

A variation on traditional type system definitions using inference rules is the co-contextual approach of Erdweg et al. [5]. Instead of propagating an environment to the sub-terms, environments are 'synthesized' along with type constraints, and the constraints and environments for sub-terms are merged. This allows for compositional and incremental processing of name and type constraints. Name resolution is expressed using operations on environments. It would be interesting to consider a bottom-up collection of constraints in our approach. The extraction algorithm of Fig. 6 can be reformulated as a bottom-up collector, using scope variables as placeholders for as yet unknown scopes. However, a key difference with our approach is the support for imports (and nominal instead of structural record types, which requires inspecting the AST associated with a type declaration), which precludes a representation of context information using a flat environment. A general challenge lies in the convergence of these approaches: how to realize incremental name and type analysis in the face of imports?

Attribute Grammars Another common approach to the implementation of static semantic analysis is by means of attribute grammars [11]. In traditional attribute grammars all 'semantic' operations are carried out in the value domain. Thus, name resolution is expressed by propagating a type environment or symbol table through attribute values. Kastens and Waite [9] provide a reusable ADT for the definition of name analysis that bears some resemblance to our scope graph framework, although the treatment of modules and imports is only discussed at the implementation level. Such attribute grammars would be a suitable mechanism for the definition of constraint collection. The extraction algorithm in Fig. 6 could easily be rephrased as an attribute grammar with scopes and type variables as inherited attributes and constraints as synthesized attribute. In *reference attribute grammars* [7], attributes can get references to tree nodes as values. Thus, attributes can be used to link references (in the scope graph sense) to their declarations. For example, Ekman and Hedin [3] provide a generic framework for name resolution based on generic reference attributes. Though this framework is part of the JastAdd Java compiler, it can be reused for other languages as well. The framework needs to be instantiated with language-specific lookup functions to resolve names. These can be specified modularly per language construct, making it possible to echo the structure of the Java language specification of name binding closely. However, these lookup functions programatically encode name binding idioms such as lexical scoping, shadowing, and hiding. Reference attributes can also be used in the specification of type analysis. Similar to our approach, name binding and typing rules can be specified mostly separately. In a generic framework, Ekman and Hedin [4] use reference attributes to link language constructs to their types and to represent type relations such as subtyping. Similar to name resolution, instantiations of the framework need to be encoded programatically. Modularity and extensibility require particular encoding patterns such as double dispatch.

The distinctive feature of our approach is that we treat name resolution using a largely separate mechanism, the scope graph, rather than integrating it into type resolution. Since some language constructs require type-dependent name resolution, there is inevitably some interaction between naming and typing, but we are still able to reuse most of our existing name resolution theory, which gives us the ability to handle a very rich variety of name binding schemes.

Future Work There are many directions for future work. One important goal is to extend our theory to handle languages with more sophisticated typing features, including subtyping, type-parameterized classes and functions, and modules with type signatures. To support popular OO language idioms, we also need to add support for multiple independent name spaces (and disambiguation across them) and type-based overloading resolution. As we make such extensions, we would also like to address the completeness of the constraint resolution algorithm (on suitably restricted sets of constraints). In particular, it would be interesting to integrate approaches to type error recovery [8, 20, 21] in order to generate good quality type error messages automatically.

On a pragmatic front, more analysis and implementation experiments are needed to determine if our approach will scale to real-world tools. In particular, we need to assess the theoretical and actual efficiency of our constraint solving algorithm. In addition, many applications for semantic analysis (e.g. in IDEs) require efficient incremental computation of name and type resolution.

On the usability front, we are interested in evaluating the expressivity and understandability of our constraint language and of higher-level name and type specification languages that we express in terms of it. Is there a payoff to the use of high-level, but perhaps more abstract concepts, in contrast to a direct implementation?

Finally, we are interested in extending the application of our building block approach to other tasks where constraint-based methods have proved useful, such as pointer analysis.

Acknowledgments

We thank the anonymous reviewers for their feedback on previous versions of this paper. This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[2] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.

[3] T. Ekman and G. Hedin. Modular name analysis for Java using JastAdd. In *GTTSE*, pages 422–436, 2006.

[4] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, pages 1–18, 2007.

[5] S. Erdweg, O. Bracevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *OOPSLA*, pages 880–897, 2015.

[6] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[7] G. Hedin. Reference attributed grammars. *informaticaSI*, 24(3), 2000.

[8] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *ICFP*, pages 3–13, 2003.

[9] U. Kastens and W. M. Waite. An abstract data type for name analysis. *ACTA*, 28(6):539–558, 1991.

[10] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010.

[11] D. E. Knuth. Semantics of context-free languages. *mst*, 2(2):127–145, 1968.

[12] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *SLE*, pages 311–331, 2012.

[13] R. Milner. A theory of type polymorphism in programming. *jcss*, 17(3):348–375, 1978.

[14] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *ESOP*, pages 205–231, 2015.

[15] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.

[16] J. Palsberg and M. I. Schwartzbach. *Object-oriented type systems*. Wiley professional computing. Wiley, 1994.

[17] H. van Antwerpen, P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A constraint language for static semantic analysis based on scope graphs with proofs. Technical Report TUD-SERG-2015-012, Software Engineering Research Group, Delft University of Technology, 2015. Available at `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2015-012.pdf`.

[18] E. Visser, G. Wachsmuth, A. P. Tolmach, P. Neron, V. A. Vergu, A. Passalaqua, and G. D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *OOPSLA*, pages 95–111, 2014.

[19] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In *SLE*, pages 260–280, 2013.

[20] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *POPL*, pages 569–582, 2014.

[21] D. Zhang, A. C. Myers, D. Vytiniotis, and S. L. P. Jones. Diagnosing type errors with class. In *PLDI*, pages 12–21, 2015.