A Constraint Language for Static Semantic Analysis Based on Scope Graphs

van Antwerpen, Hendrik; Néron, Pierre; Tolmach, Andrew; Visser, Eelco; Wachsmuth, Guido

DOI
10.1145/2847538.2847543

Publication date
2016

Document Version
Final published version

Published in
PEPM '16

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.
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  
guwsac@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 mod-ules 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 realizably 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...
Contributions The specific technical contributions of this paper are the following:

- 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, \( x_1 \), \( x_4 \), and \( x_8 \) are different occurrences of the same name \( x \). 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. \( \{x_1\} \). \( S(\mathcal{G}) \) denotes the set of scopes of \( \mathcal{G} \).
- A declaration is an occurrence of an identifier that introduces a name. We write \( x_i^o \) 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. \( \downarrow x_1^o \). \( D(\mathcal{G}) \) denotes the set of declarations of \( \mathcal{G} \).
- A reference is an occurrence of an identifier referring to a declaration. We write \( x_i^o \) 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. \( \uparrow x_1^o \). \( R(\mathcal{G}) \) 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\text{Sc}) in the grammar of Fig. 3, 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 \rightarrow xD specifies that declaration xD belongs to scope s. Graphically: [D → x].
- A reference constraint xR \rightarrow s specifies that reference xR belongs to scope s. Graphically: [xR ↪→ 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 SC(xD) = s for s \rightarrow xD and SC(xR) = s for xR \rightarrow s. We define by comprehension the sets of declarations and references belonging to a scope s, as D(s) = \{xD | SC(xD) = s\} and R(s) = \{xR | SC(xR) = 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 \Upsilon, we use a subscript notation (e.g. \Upsilon_D(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\text{Res}) in the grammar) represent requirements on successful name resolution:

- A resolution constraint R \rightarrow D specifies that a given reference must resolve to a given declaration. Typically, the declaration is specified as a declaration variable \delta. For example, in Fig. 2 the constraints xR_4 \rightarrow \delta_4 and xR_8 \rightarrow \delta_8 require that references xR_4 and xR_8 resolve to (as yet unknown) declarations \delta_4 and \delta_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 xR_4 and xR_8 both end at declaration x_1, the (sole) solution to these constraints is a substitution mapping both \delta_4 and \delta_8 to x_1. Applying this substitution yields the valid resolutions x_4^R \mapsto x_1^D and x_8^R \mapsto x_1^D.

In addition to constraints about the resolution of references, C\text{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 \Upsilon(s) is obtained by projecting the identifiers from the set of declarations in scope s.

Thus, for example, in Fig. 2 the constraint \Upsilon(1) requires that scope 1 should have no duplicate declarations. These types of constraints are satisfied when the property they specify holds.

Typing Constraints Typing constraints (C\text{T}) represent requirements for type consistency of the program:

- A type declaration constraint D : T associates a type with a declaration. This constraint is used in two flavors: associating a type variable (\tau) with a concrete declaration, or associating a type variable with a declaration variable. In Fig. 2, the constraints xD_0 : \tau_2 and yD_3 : \tau_7 associate distinct type variables with declarations xD_0 and yD_3. (For ease of reading, we choose type variable names corresponding to subexpression label numbers.) The constraint \delta_4 : \tau_7 requires the type of the declaration to which xD_4 resolves to be the same as the type \tau_7 of the reference considered as an expression.

- A type equality constraint T \equiv T specifies that two types should be equal. In Fig. 2, the constraint \tau_2 \equiv \tau_9 arises from the constant expression \iota_2, and the constraint \tau_5 \equiv \tau_7 arises from the fact that the \texttt{==} operator takes integer operands. The constraint \tau_8 \equiv \tau_9 \equiv \texttt{Bool} arises in two ways, from the fact that \texttt{==} returns a Boolean and the fact that \texttt{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 declarations and type variables that satisfies all the constraints. For example, the substitution for \tau_0 can be deduced either from the constraints \tau_2 \equiv \tau_9 and \tau_7 \equiv \texttt{Int}, or from the constraints \tau_0 \equiv \texttt{Int} and the unification of \tau_7 and \tau_2 (via \texttt{δ_8 = x_2^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 (visible) declarations in one scope 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 \rightarrow l s_2 specifies a direct l-labeled edge from scope s_1 to s_2. (Graphically: [s_1 ↪→ 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.

---

<table>
<thead>
<tr>
<th>Scope graph constraints</th>
<th>Typing constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1 \rightarrow \delta_1</td>
<td>xD_0 : \tau_2, \tau_2 \equiv \texttt{Int}</td>
</tr>
<tr>
<td>y_3 \rightarrow \delta_3</td>
<td>yD_3 : \tau_9, \tau_9 \equiv \texttt{Int}</td>
</tr>
<tr>
<td>\delta_4 \rightarrow \tau_7</td>
<td>\tau_7 \equiv \tau_2, \tau_7 \equiv \texttt{Int}</td>
</tr>
<tr>
<td>\tau_9 \equiv \tau_7, \delta_8 \rightarrow \tau_9</td>
<td>\tau_9 \equiv \texttt{Bool}</td>
</tr>
<tr>
<td>\tau_5 \equiv \tau_7</td>
<td>\tau_5 \equiv \texttt{Int}</td>
</tr>
<tr>
<td>x_8 \rightarrow \delta_9</td>
<td>\tau_8 \equiv \tau_9 \equiv \tau_9 \equiv \tau_6 \equiv \tau_5 \equiv \texttt{Int}</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Figure 2. Declarations and references in global scope.
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 → 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_3` resolves to `f_1`. 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_2` resolves to declaration `n_1` rather than to `n_1`.

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 \( n_3 \rightarrow x \) specifies \( s \) as the associated scope of declaration \( x \). 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: \( 2 \rightarrow 3 \).

The LMR program in the right part of Fig. 4 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 \( A_1 \) and \( B_4 \). 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. \( A_1 \leftarrow A_2 \). 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 \( n_1 \rightarrow x \) specifies a nominal labeled edge from scope \( s \) to reference \( x \). (Graphically: \( 2 \rightarrow 4 \).) Such an edge makes visible in \( s \) all declarations that are visible in the associated scope of the declaration to which \( x \) resolves, according to the label on the edge.

For example, `import A_2` is represented by the reference \( A_2 \) in scope \( A_1 \) and an import arrow \( A_1 \leftarrow A_2 \). 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 \( x \), which resolves to a declaration \( x_1 \) with associated scope \( S_2 \), then all declarations in \( S_2 \) are reachable in \( S_1 \). Thus, in the example, reference \( a_2 \) resolves to declaration \( a_3 \) since the import \( A_2 \) resolves to declaration \( A_3 \), and the associated scope \( 2 \) of \( A_2 \) contains declaration \( A_3 \). Note that the resolution calculus is parametrized 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 2.4.

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 \( e.f \), 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(x)`. We model the fields of a record type definition as declarations (here just \( x \) in a scope (here, scope 2) associated with the record type name declaration \( A_1 \). The resolution constraint \( [D(2)] \) forbids duplicate field names.

To construct a new record of a declared record type (e.g. `A_1`), 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...
record A1 { x2 : Int3 }
def a4 = ( new A5 (x4=17) )
def y9 = ( a10 . x11 )

Typing constraints
\[ x_2^D : \tau_3 \quad \tau_3 \equiv \text{Int} \]
\[ x_4^D : \tau_8 \quad \tau_8 \equiv \text{Rec}(\delta_8) \]
\[ y_9^D : \tau_{11} \quad \delta_6 : \tau_6 \]
\[ \delta_{10} : \tau_{10} \quad \delta_{11} : \tau_{12} \]
\[ \tau_7 \equiv \tau_6 \quad \tau_7 \equiv \text{Int} \]
\[ \tau_{10} \equiv \text{Rec}(\delta_{12}) \]

Solution
\[ \delta_6 = \delta_{11} = x_2^D \]
\[ \delta_8 = \delta_{12} = x_4^D \]
\[ \delta_{10} = \delta_{14} \quad \text{g12} = 2 \]
\[ \tau_3 = \tau_6 = \tau_7 = \tau_{12} = \text{Int} \]
\[ \tau_8 = \tau_{10} = \text{Rec}(\delta_{12}) \]

Figure 4. Field access.

An association constraint \( D \rightarrow S \) specifies that a given declaration has a given associated scope.

Specifically, we use \( g12 \sim \text{g12} \) to say that \( g12 \) must be the associated scope of \( g12 \).

Solving these constraints will lead to a solution for \( g12 \) — in this case the associated scope of \( x_2^D \), scope \( 2 \) — such that the appropriate scope can be imported into scope \( 3 \). After that, \( x_4^D \) can be resolved as usual to the corresponding field declaration \( x_2^D \), yielding its type \( \tau_3 \equiv \text{Int} \).

With As a further variant, we discuss an expression form inspired by the \( \text{with} \) statement in the Pascal language. In the expression \( \text{with } e \text{ do } e' \), \( e \) 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 in the enclosing lexical scope. Replacing \((a . x)\) by \( \text{with } a \text{ do } x \) in the code of Fig. 4 produces identical constraints, with the addition of a scope graph edge \( \langle 4 \rightarrow 1 \rangle \).

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. 5, 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 \( C^{SG} \) specify a scope graph which defines the binding structures of the program. Resolution constraints \( C^{RES} \) 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 3. We can combine using conjunction \( C = C \land C' \) and \( C \land C' \) 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 \( c \in C_T \) and a set of labels \( I \in L \). We describe the former here and the latter in Section 3.4.

Type Constructors Types in \( T \) are either type variables \( \tau \) or type constructor applications \( c(T, \ldots, T) \) with \( c \in C_T \), a set of language-specific type constructors. Each constructor \( c \) has an associated arity \( c :: n \). For example, \( \text{Int} \) and \( \text{Bool} \) are type constructors with arity 0 and \( \text{Fun} \) 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 \( \text{Rec}(d) \) with \( d \) a type name declaration in
the program; thus, in Fig. 3 the record definition $\wedge$ defines the type $Rec(\lambda^1)$.

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 
\[
[p] = C_p^R \land C_p^{res} \land C_p^{s}\]
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 $C^s$ directly as a ground scope graph. We next define a satisfiability relation $|=\text{by cases on ground resolution}$ based on our ground resolution $\equiv$ to unique ground types in $T$.

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, $[-][-]^c$ 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 $[-][-]$ on sequences of items of syntactic category $c$ to mean the result of applying $[-][-]$ to each item and returning the conjunction of the resulting constraints, or True for the empty sequence.

For our actual implementation, we have extended this algorithm to handle additional features of LMR, such as modules and records. We have also developed a parser that generates the abstract syntax tree of a program, and a constraint analyzer that computes the constraint $\land [ds]^{decl}$ from this tree.

Figure 5. Syntax of LMR.

Figure 6. Constraint extraction for LMR. Scope names $(s)$ occurring free in rhs of rules are new ground scopes. Type variables $(\tau)$, scope variables $(c)$, and declaration variables $(\delta)$ occurring free are fresh variables.
ground types, declaration variables
we apply

Resolution and Typing Constraints The \(\models\) relation is given by the inductive rules in Fig. 8, where \(\models\) is the syntactic equality on terms and \(\vdash\) the inductive rules in Fig. 9. These rules all implicitly apply to references or declarations. Given a multiset \(\{1\}\) from reference to declaration through a sequence of edges. A path \(l,S\) is a list of steps representing the atomic scope transitions in the calculus rules in Fig. 9. These rules all implicitly apply to a fixed graph \(G\), which we omit to avoid clutter. The calculus defines the resolution relation in terms of edges in the scope graph, reachability declarations, and visible declarations. Here \(I\) 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 \(I\) from a resolution when it is empty. The \(S\) 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.

\[ C := C^G | C^T | C^R | C \land C | True \]
\[ C^G := R \rightarrow S | S \rightarrow D | S \rightarrow S | D \rightarrow S | S \rightarrow R \]
\[ C^R := R \rightarrow D | D \rightarrow S | I \vdash N \mid N \not\subseteq N \]
\[ C^T := T \equiv T | D : T \]
\[ D := \delta \mid x_i^D \]
\[ R := x_i^R \]
\[ S := \varsigma \mid n \]
\[ T := \tau \mid c(T, ..., T) \]
\[ N := \mathcal{T}(S) | \mathcal{R}(S) | \mathcal{V}(S) \]

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 \(E(l, S_2)\) 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 \(N(l, y^R, S)\) 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 \(D(x^D)\) 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 \(\vdash_G p : x_i^R \rightarrow x_i^D\) according to the calculus rules in Fig. 9. These rules all implicitly apply to a fixed graph \(G\), 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 \(I\) 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 \(I\) from a resolution when it is empty. The \(S\) 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.

Resolution paths

\[ s := D(x_i^D) \mid E(l, S_2) \mid N(l, x_i^R, S) \]
\[ p := [] \mid s | p \cdot p \quad \text{(inductively generated)} \]
\[ \vdash s \cdot p : S \rightarrow x_i^D \]
\[ \vdash \neg(s \cdot p) : x_i^D \rightarrow \neg(s \cdot p) \]

Edges in scope graph

\[ \vdash E(l, S_2) : S_1 \rightarrow S_2 \]
\[ \vdash N(l, y_i^R, S_2) : S_1 \rightarrow S_2 \]
\[ \vdash D(x_i^D) : S \rightarrow x_i^D \]

Transitive closure

\[ \vdash A \rightarrow A \]
\[ \vdash B \not\rightarrow S : A \rightarrow B \]
\[ \vdash p : B \rightarrow C \]
\[ \vdash s \cdot p : A \rightarrow C \]

Reachable declarations

\[ \vdash \neg(s \cdot p) : x_i^D \rightarrow \neg(s \cdot p) \]

Visible declarations

\[ \vdash \neg(s \cdot p) : x_i^D \rightarrow \neg(s \cdot p) \]

Reference resolution

\[ \vdash x_i^R : S \rightarrow x_i^D \]
\[ \vdash S \rightarrow x_i^D \]

Figure 7. Syntax of constraints

Figure 8. Interpretation of resolution and typing constraints

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.
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 \( \mathcal{D} \) label) that defines the scope visibility policy. Fig. 10 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 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 (\( \mathcal{P} \)) and local declarations shadow declarations in parents. The next policy extends lexical scope with non-transitive imports (\( \mathcal{I} \)). 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 (\( \mathcal{TI} \)). The transitive includes policy is a variation in which a local declaration does not shadow included (\( \mathcal{Inc} \)) declarations. The final policy combines three import policies, not providing rules to disambiguate between paths through different kinds of imports. 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 2.2, i.e., finding \( \phi \) and \( \psi \) that satisfy \( \Box \). Our algorithm works only for a restricted class of generated constraints: all constraints in \( C^G \) must be ground, except that scope variables \( \zeta \) can appear as targets in direct edge constraints (e.g., \( S \xrightarrow{\zeta} \)). This restriction is met by the constraints generated by the LMR collection algorithm in Section 2.2. 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 \( G \) 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 \( \phi \). Thus, our algorithm builds \( \phi \) (and \( \psi \)) incrementally. The key idea is that we can solve some resolution and typing constraints even when \( \phi \) is not yet fully defined, in such a way that the solution remains valid as it becomes more defined.

4.2 Name Resolution Algorithm

In order to solve resolution constraints (e.g., \( x^R \to \delta \)) or to compute the set of visible elements from a scope (\( V(S) \)) we need an algorithm that computes the name resolution relation (\( x^R \to x^D \)) 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 3.3.

Incomplete Scope Graphs

A new feature on the algorithm is that it can operate on an incomplete scope graph, specified by a set of constraints that may still contain 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 \( G \) a reference \( x^R \) in a scope \( S \) resolves to declaration \( x^D \) in the parent scope \( S' \). In a bigger graph \( G' \) that also has a declaration \( x^D \) in \( S \) itself, \( x^R \) will resolve to \( x^D \), and the old resolution to \( x^D \) 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, \( R[\ell](x^R) \), returns either a set of declarations or \( U \) (unknown) if the reference cannot be resolved in the current graph. Similarly, the environment functions \( Env_{\ell}(\ell, S)(S) \) 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:

- \( R^p[I][x^R] \) returns the set of declarations to which the reference resolves.
- \( Env_v[I, S](S) \) returns the set of declarations that are reachable from scope \( S \) with a minimal path satisfying the regular expression \( re \).
- \( Env^v_re[I, S](S) \) 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.
- \( Env^p_re[I, S](S) \) returns the set of declarations accessible from \( S \) with a \( D \) step, i.e. the set of declarations in \( S \).
- \( Env^v_p_re[I, S](S) \) returns the set of declarations accessible from \( S \) with an \( l \)-labeled step.
- \( IS^v[I](S) \) 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: \( \emptyset \) 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 \( \triangleleft \) on sets of declarations is defined by:

\[
D_1 \triangleleft D_2 \overset{\Delta}{=} \{ x^D_1 \mid x^D_1 \in D_1 \vee (x^D_i \in D_2 \land \exists j, x^D_j \in D_1) \} .
\]

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

\[
(f_1, D_1) \triangleleft (f_2, D_2) \overset{\Delta}{=} \begin{cases} (f_2, D_1 \triangleleft D_2) & \text{if } f_1 = \top \\ (P, D_1) & \text{otherwise} \end{cases}
\]

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

\[
\bigcup_{i \in I} (f_i, D_i) \overset{\Delta}{=} \begin{cases} (T, D) & \text{if } \forall i \in I, (f_i = \top) \\ (P, D) & \text{otherwise} \end{cases}
\]

where \( D = \{ x^D_i \mid (\forall i \in I, \exists j, x^D_j \in D_i) \} \). Given a regular expression over labels \( re \) and a label \( L \), \( L^{-1} re \) denotes the Brzozowski derivative \( \overline{re} \) of \( re \) by \( L \). Given a partially ordered set \( L \), \( Max(L) \) denotes the set of maximal elements of \( L \), i.e. \( \{ l \in L \mid \exists l' \in L, l < l' \} \). Given a scope \( S \) and a label \( l \), we define:

\[
S^r_{l^+} \overset{\Delta}{=} \{ x^R \mid S \xrightarrow{l^+} x^R \} \quad S^r_{l^+} \overset{\Delta}{=} \{ S' \mid S \xrightarrow{\longrightarrow} S' \}
\]

### 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 \( \#(\overline{R}(G)) \), \( \#(S(G)) \), \( \#(O) \), where \( \#(A) \) denotes the cardinality of set \( A \) and \( O \) is the following well founded order among the different functions:

\[
Env_v > Env^v_re > Env^v_p_re > IS > R
\]

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 set of declarations returned by the different functions. Given two sets of scopes \( I \) and \( S \) and a scope \( S \), we define \( \mathbb{P}[I, S](S) \) as:

\[
\{ p \cdot D(d) \mid S' \subseteq I, S \cup \{ S \} \vdash \exists \phi \mid p : S \Rightarrow S' \land S \subseteq S' \}
\]

given a path \( p \) such that \( p = p' \cdot D(d) \), \( \Delta(p) \) 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

\[
\neg S \triangleleft \{ p \cdot D(d) \mid S \subseteq I, S \cup \{ S \} \vdash \exists \phi \mid p : S \Rightarrow S' \land S \subseteq S' \}
\]

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

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

\[
R[I][x^R] = \Delta (\{ p \mid \exists d, \exists r : p : x^R \Rightarrow d \})
\]

\[
Env_v[I, S](S) = \begin{cases} \emptyset & \text{if } S \subseteq S \\Delta (\neg S \triangleleft (p : D(d) \in \mathbb{P}[I, S](S) \mid p \in r e)) & \text{otherwise} \end{cases}
\]

\[
Env^v_p[I, S](S) = \Delta (\{ d : S \subseteq I \land p \in \mathbb{P}[I, S](S) \})
\]

\[
Env^p[I, S](S) = \Delta (\{ \{ d : S \subseteq I \land p \in I \} \})
\]

\[
IS^v[I](S) = \{ S' \mid \exists y^R, \exists S \subseteq I \mid E(S, y^R, S') : S \Rightarrow S' \}
\]

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 \( I \vdash s : S \Rightarrow S' \) then its tail \( p \) is also minimal from \( S' \), due to the lexicographic ordering.

\[\square\]

#### 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 \( G \), 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 \( G \):

\[
\vdash G x^R \mathbin{\xrightarrow{\phi}} x^D_i \quad \forall \phi, \vdash_{\phi(G)} x^R \mathbin{\xrightarrow{\phi}} x^D_i \quad (\Phi)
\]

where we write \( \vdash_{\phi} \) for the resolution relation for graph \( G \) and \( \phi(G) \) is the ground scope graph corresponding to the application of substitution \( \phi \) to variables in \( G \). Similarly a declaration \( x^D_i \) is visible from scope \( S \) in an incomplete scope graph \( G \) 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 \( G \) is stable for a reference or a scope \( o \), denoted \( G \downarrow o \), if all the resolutions in all its ground instances are the same:

\[
G \downarrow o \overset{\Delta}{=} \forall \phi, \phi' \vdash_{\phi(G)} o \mathbin{\xrightarrow{\phi}} x^D_i \Rightarrow_{\phi'} \vdash_{\phi'(G)} o \mathbin{\xrightarrow{\phi'}} x^D_i
\]

**Soundness**

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

**Lemma 2.** For any incomplete graph \( G \):

\[
x^D_i \in R_G(x^R) \Rightarrow \vdash G x^R \mathbin{\xrightarrow{\phi}} x^D_i \land G \downarrow x^R
\]

where \( R_G(x^R) \) denotes the top-level resolution function \( R[\emptyset](x^R) \) for the graph \( G \).
Lemma 1 states that this property holds when the graph $G$ is ground. We next prove that if the resolution on an incomplete graph $G'$ terminates with a total flag $T$ then for any graph $G'$ that is an instance of $G$, the result is the same:

$$\text{Env}_{v}[\varnothing,\mathcal{S}](x^R)_{G'} = (T, D) \implies \text{Env}_{v}[\varnothing,\mathcal{S}](x^R)_{G} = (T, D) \quad (i)$$

**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 $G$ be an incomplete scope graph and $G'$ one of its instances. If a resolution on $G$ contains a set of declarations for a given name then the resolution on $G'$ contains the same declarations for this name:

$$\text{Env}_{v}[\varnothing,\mathcal{S}](S)_{G} = (\varnothing, D) \implies \text{Env}_{v}[\varnothing,\mathcal{S}](S)_{G'} = (\varnothing, D') \implies \forall x, \{x^D \in D\} \neq \emptyset \implies \{x^D \in D'\} \quad (ii)$$

**Proof.** We prove this result along with similar result for all the other functions by induction on the termination order of the algorithm, using $\varnothing$.

Finally, we can prove Lemma 2.

**Proof.** Let $S_e = R_e(x^R)$ and pick $x^D_e \in S_e$. To prove that $x^R$ resolves to $x^D_e$ in $G$, let $G'$ be an arbitrary ground instance of $G$. Using (ii) we have $x^D_e \in R_{G'}(x^R)$ and by Lemma 1 we have $G_{G'} x^R \rightarrow x^D_e$. By (ii) we get that $G_{G'} x^R \rightarrow x^D_e$.

To prove stability, let $G_1$ and $G_2$ be ground instances of $G$. Then using (ii), we have $R_{G_1}(x^R) = R_{G_2}(x^R) = S_e$, so by definition we have $G_{G_1} x^R = G_{G_2} x^R$.

### 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 $G$ and a scope $S$, we compute name collections as:

$$
\begin{align*}
N_G(T(S)) &= \pi(T_G(S)) \\
N_G(T_{\mathcal{R}}(S)) &= \pi(\mathcal{R}_G(S)) \\
N_G(T(S,E)) &= \pi(x^D \mid E, \text{Env}_{v}[\varnothing,\mathcal{S}](S) = (T, E \land x^D_e \in E))
\end{align*}
$$

**Lemma 3 (Name computation soundness).** If the computation of a name collection $E$ terminates on an incomplete graph $G$, its results is the semantics of the name collection for any graph $G'$ that is an instance of $G$:

$$N_G(E) = M \implies [E]_{G'} = M.$$ 

### 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 $(C, G, \psi)$ 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 and 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 $(C, G, \psi)$ 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 $G$ and filling $\psi$.

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 $x^R \rightarrow \delta$ using the resolution algorithm from Fig. 11. If a resolution is found, it is substituted for the variable $\delta$. 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 $x^D \rightarrow \varsigma$ by looking up the scope $S$ associated with ground declaration $x^b$ in the scope graph. By substituting $S$ for $\varsigma$, 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 $U(T_1, T_2)$, as described in 11. 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. ($A(x)$ is the multiplicity of $x$ in $A$).
- Rule S-SUBNAME solves $N_1 \subseteq N_2$ 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 $x^D : T$. The rule considers two cases. When no type assignment is declared for $x^D$ in $\psi$ (i.e. the first time that it is encountered) the assignment is added to the typing environment $\psi$. When a type assignment is declared (i.e. for subsequent encounters), the type $T$ from the constraint is unified with the type $\psi(x^D)$ 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:

$$
(C, G, \psi) \rightarrow^* \text{True, } G', \psi' \implies \\
\exists \phi, \phi(G) = G' \land \forall \sigma, \sigma G', \sigma \psi' \models \sigma(\phi(C))
$$

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

**Unification:** If $U(t_1, t_2) = \sigma$ then $\sigma t_1 = \sigma t_2 \land \sigma \sigma = \sigma$. See 11 for a survey on unification problem and unification algorithms for first order terms.

**Resolution Soundness:** Now we can prove the Lemma 3 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:

$$
\begin{align*}
&\forall(C_1, G_1, \psi_1, (C_2, G_2, \psi_2)) \rightarrow (C_2, G_2, \psi_2) \Rightarrow \\
&\exists \sigma', \sigma' G_1 = G_2 \land \\
&\forall \sigma, (\sigma G_1, \sigma \psi_1) \models \sigma(C_2) \Rightarrow \\
&\sigma G_2, \sigma \psi_2 \models \sigma(\phi(C_1))
\end{align*}
$$

(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.
that TS cannot be used to express languages requiring type inference to a proof assistant such as Coq. In particular, the semantics of no-language definitions [18] requires a semantics that can be explained a problem for further development. The aim to verify properties of checking, reference resolution, and code completion.

Eclipse IDE support for editor services such as name and type error type-dependent name resolution. NaBL and TS are implemented by ports in an abstract syntax tree without recourse to environments or sketched in [18].) Rules in TS are similar to traditional typing judgements. (The design of TS is not formally published, but it is componenting of static semantic analysis is by means of attribute grammars 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].

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-DLs [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 judgements, 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.

Conclusions 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 [15] 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. [3]. 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 gram-
There are many directions for future work. One to reuse most of our existing name resolution theory, which gives us some interaction between naming and typing, but we are still able to reuse our declarative incremental computation of name and type resolution. On a pragmatic front, more analysis and implementation experiments are needed to determine if our approach will scale to many applications for semantic analysis (e.g. in IDEs) requiring 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