A Theory of Name Resolution

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

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

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

1 Introduction

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

In practice, the name resolution rules of real programming languages are usually described using *ad hoc* and informal mechanisms. Even when a language

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 205–231, 2015.
DOI: 10.1007/978-3-662-46669-8_9

is formalized, its resolution rules are typically encoded as part of static and dynamic judgments tailored to the particular language, rather than being presented separately using a uniform mechanism. This lack of modularity in language description is mirrored in the implementation of language tools, where the resolution rules are often encoded multiple times to serve different purposes, e.g., as the manipulation of a symbol table in a compiler, a use-to-definition display in an IDE, or a substitution function in a mechanized soundness proof. This repetition results in duplication of effort and risks inconsistencies. To see how much better this situation might be, we need only contrast it with the realm of syntax definition, where context-free grammars provide a well-established declarative formalism that underpins a wide variety of useful tools.

Formalizing Resolution. This paper describes a formalism that we believe can help play a similar role for name resolution in lexically-scoped languages. It consists of a *scope graph*, which represents the naming structure of a program, and a *resolution calculus*, which describes how to resolve references to declarations within a scope graph. The scope graph abstracts away from the details of a program AST, leaving just the information relevant to name resolution. Its nodes include name references, declarations, and "scopes," which (in a slight abuse of conventional terminology) we use to mean minimal program regions that behave uniformly with respect to name resolution. Edges in the scope graph associate references to scopes, declarations to scopes, or scopes to "parent" scopes (corresponding to lexical nesting in the original program AST). The resolution calculus specifies how to construct a path through the graph from a reference to a declaration, which corresponds to a possible resolution of the reference. Hiding of one definition by a "closer" definition is modeled by providing an ordering on resolution paths. Ambiguous references correspond naturally to multiple resolution paths starting from the same reference node; unresolved references correspond to the absence of resolution paths. To describe programs involving explicit name spaces, the scope graph also supports giving names to scopes, and can include "import" edges to make the contents of a named scope visible inside another scope. The calculus supports complex import patterns including transitive and cyclic import of scopes.

This language-independent formalism gives us clear, abstract definitions for concepts such as scope, resolution, hiding, and import. We build on these concepts to define generic notions of α-equivalence and valid renaming. We also give a practical algorithm for computing conventional static environments mapping bound identifiers to the AST locations of the corresponding declarations, which can be used to implement a deterministic, terminating resolution function that is consistent with the calculus. We expect that the formalism can be used as the basis for other language-independent tools. In particular, any tool that relies on use-to-definition information, such as an IDE offering code completion for identifiers, or a live variable analysis in a compiler, should be specifiable using scope graphs.

On the other hand, the construction of a scope graph from a given program is a language-*dependent* process. For any given language, the construction can be

specified by a conventional syntax-directed definition over the language grammar; we illustrate this approach for a small language in this paper. We would also like a more generic *binding specification language* which could be used to describe how to construct the scope graph for an arbitrary object language. We do not present such a language in this paper. However, the work described here was inspired in part by our previous work on NaBL [16], a DSL that provides high-level, non-algorithmic descriptions of name binding and scoping rules suitable for use by a (relatively) naive language designer. The NaBL implementation integrated into the Spoofax Language Workbench [14] automatically generates an incremental name resolution algorithm that supports services such as code completion and static analysis. However, the NaBL language itself is defined largely by example and lacks a high-level semantic description; one might say that it works well in practice, but not in theory. Because they are language-independent, scope graphs can be used to give a formal semantics for NaBL specifications, although we defer detailed exploration of this connection to further work.

Relationship to Related Work. The study of name binding has received a great deal of attention, focused in particular on two topics. The first is how to represent (already resolved) programs in a way that makes the binding structure explicit and supports convenient program manipulation "modulo α-equivalence" [7,20,3,10,4]. Compared to this work, our system is novel in several significant respects. (i) Our representation of program binding structure is *independent* of the underlying language grammar and program AST, with the benefits described above. (ii) We support representation of ill-formed programs, in particular, programs with ambiguous or undefined references; such programs are the normal case in IDEs and other front-end tools. (iii) We support description of binding in languages with explicit name spaces, such as modules or OO classes, which are common in practice.

A second well-studied topic is binding specification languages, which are usually enriched grammar descriptions that permit simultaneous specification of language syntax and binding structure [22,8,13,23,25]. This work is essentially complementary to the design we present here.

Specific Contributions.

- *Scope Graph and Resolution Calculus*: We introduce a language-independent framework to capture the relations among *references*, *declarations*, *scopes*, and *imports* in a program. We give a declarative specification of the resolution of references to declarations by means of a calculus that defines resolution paths in a scope graph (Section 2).
- *Variants*: We illustrate the modularity of our core framework design by describing several variants that support more complex binding schemes (Section 2.5).
- *Coverage*: We show that the framework covers interesting name binding patterns in existing languages, including various flavors of let bindings, qualified names, and inheritance in Java (Section 3).

- *Scope graph construction:* We show how scope graphs can be constructed for arbitrary programs in a simple example language via straightforward syntax-directed traversal (Section 4).
- *Resolution algorithm*: We define a deterministic and terminating resolution algorithm based on the construction of binding environments, and prove that it is sound and complete with respect to the calculus (Section 5).
- *α-equivalence and renaming*: We define a language-independent characterization of α-equivalence of programs, and use it to define a notion of valid renaming (Section 6).

  The extended version of this paper [19] presents the encoding of additional name binding patterns and the details of the correctness proof of the resolution algorithm.

2 Scope Graphs and Resolution Paths

Defining name resolution directly in terms of the abstract syntax tree leads to complex scoping patterns. In unary lexical binding patterns, such as lambda abstraction, the scope of the bound variable is the subtree dominated by the binding construct. However, in name binding patterns such as the sequential `let` in ML, or the variable declarations in a block in Java, the set of abstract syntax tree locations where the bindings are visible does not necessarily form a contiguous region. Similarly, the list of declarations of formal parameters of a function is contained in a subtree of the function definition that does not dominate their use positions. Informally, we can understand these name binding patterns by a conceptual mapping from the abstract syntax tree to an underlying pattern of *scopes*. However, this mapping is not made explicit in conventional descriptions of programming languages.

We introduce the language-independent concept of a *scope graph* to capture the scoping patterns in programs. A scope graph is obtained by a language-specific mapping from the abstract syntax tree of a program. The mapping collapses all abstract syntax tree nodes that behave uniformly with respect to name resolution into a single 'scope' node in the scope graph. In this paper, we do not discuss how to specify such mappings for arbitrary languages, which is the task of a binding specification language, but we show how it can be done for a particular toy language, first by example and then systematically. We assume that it should be possible to build a scope graph in a single traversal of the abstract syntax tree. Furthermore, the mapping should be *syntactic*; *no name resolution* should be necessary to construct the mapping.

Figures 1 to 3 define the full theory. Fig. 1 defines the structure of scope graphs. Fig. 2 defines the structure of *resolution paths*, a subset of resolution paths that are *well-formed*, and a *specificity ordering* on resolution paths. Finally, Fig. 3 defines the *resolution calculus*, which consists of the definition of *edges* between scopes in the scope graph and their transitive closure, the definition of *reachable* and *visible* declarations in a scope, and the *resolution* of references to declarations. In the rest of this section we motivate and explain this theory.

References and declarations

- $x_i^{\mathsf{D}}{:}S$: declaration with name x at position i and optional associated named scope S
- x_i^{R}: reference with name x at position i

Scope graph

- $\mathcal{G}$: scope graph
- $\mathcal{S}(\mathcal{G})$: scopes S in $\mathcal{G}$
- $\mathcal{D}(S)$: declarations $x_i^{\mathsf{D}}{:}S'$ in S
- $\mathcal{R}(S)$: references x_i^{R} in S
- $\mathcal{I}(S)$: imports x_i^{R} in S
- $\mathcal{P}(S)$: parent scope of S

Well-formedness properties

- $\mathcal{P}(S)$ is a partial function
- The parent relation is well-founded
- Each x_i^{R} and x_i^{D} appears in exactly one scope S

Fig. 1. Scope graphs

Resolution paths

$$s := \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{I}(x_i^{\mathsf{R}}, x_j^{\mathsf{D}}{:}S) \mid \mathbf{P}$$
$$p := [] \mid s \mid p \cdot p$$
(inductively generated)
$$[] \cdot p = p \cdot [] = p$$
$$(p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$$

Well-formed paths

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_, \_)^*$$

Specificity ordering on paths

$$\frac{}{\mathbf{D}(\_) < \mathbf{I}(\_, \_)} \quad (DI)$$

$$\frac{}{\mathbf{I}(\_, \_) < \mathbf{P}} \quad (IP)$$

$$\frac{}{\mathbf{D}(\_) < \mathbf{P}} \quad (DP)$$

$$\frac{s_1 < s_2}{s_1 \cdot p_1 < s_2 \cdot p_2} \quad (Lex1)$$

$$\frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2} \quad (Lex2)$$

Fig. 2. Resolution paths, well-formedness predicate, and specificity ordering

Edges in scope graph

$$\frac{\mathcal{P}(S_1) = S_2}{\mathbb{I} \vdash \mathbf{P} : S_1 \longrightarrow S_2} \quad (P)$$

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \setminus \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbb{I} \vdash \mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2} \quad (I)$$

Transitive closure

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

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

Reachable declarations

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

Visible declarations

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

Reference resolution

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

Fig. 3. Resolution calculus

```
program = decl*
   decl = module id { decl* } | import qid | def id = exp
    exp = qid | fun id { exp } | fix id { exp }
        | let bind* in exp | letrec bind* in exp | letpar bind* in exp
        | exp exp | exp ⊕ exp | int
    qid = id | id . qid
   bind = id = exp
```

Fig. 4. Syntax of LM

2.1 Example Language

To illustrate the scope graph framework we use the toy language LM, defined in Fig. 4, which contains a rather eclectic combination of features chosen to exhibit both simple and challenging name binding patterns. LM supports the following constructs for binding variables:

- Lambda and mu: The functional abstractions **fun** and **fix** represent lambda and mu terms, respectively; both have basic unary lexically scoped bindings.
- Let: The various flavors of let bindings (sequential **let**, **letrec**, and **letpar**) challenge the unary lexical binding model.
- Definition: A definition (**def**) declares a variable and binds it to the value of an initializing expression. The definitions in a module are not ordered (no requirement for 'def-before-use'), giving rise to mutually recursive definitions.

Most programming languages have some notion of *module* to divide a program into separate units and a notion of *imports* that make elements of one module available in another. Modules change the standard lexical scoping model, since names can be declared either in the lexical parent or in an imported module. The modules of LM support the following features:

- Qualified names: Elements of modules can be addressed by means of a qualified name using conventional dot notation.
- Imports: All declarations in an imported module are made visible without the need for qualification.
- Transitive imports: The definitions imported into an imported module are themselves visible in the importing module.
- Cyclic imports: Modules can (indirectly) mutually import each other, leading to cyclic import chains.
- Nested modules: Modules may have sub-modules, which can be accessed using dot notation or by importing the containing module.

In the remainder of this section, we use LM examples to illustrate the basic features of our framework. In Section 3 and Appendix A of [19] we explore the expressive power of the framework by applying it to a range of name binding patterns from both LM and real languages. Section 4 shows how to construct scope graphs for arbitrary LM programs.

2.2 Declarations, References, and Scopes

We now introduce and motivate the various elements of the name binding framework, gradually building up to the full system described in Figures 1 to 3. The central concepts in the framework are *declarations*, *references*, and *scopes*. A *declaration* (also known as *binding occurrence*) *introduces* a name. For example, the **def** `x = e` and **module** `m { .. }` constructs in LM introduce names of variables and modules, respectively. (A declaration may or may not also *define* the name; this distinction is unimportant for name resolution—except in the case where the declaration defines a module, as discussed in detail later.) A *reference* (also known as *applied occurrence*) is the *use* of a name that refers to a declaration with the same name. In LM, the variables in expressions and the names in import statements (e.g. the `x` in **import** `x`) are references. Each reference and declaration is unique and is distinguished not just by its name, but also by its position in the program's AST. Formally, we write x_i^{R} for a reference with name x at position i and x_i^{D} for a declaration with name x at position i.

A *scope* is an abstraction over a group of nodes in the abstract syntax tree that behave uniformly with respect to name resolution. Each program has a *scope graph* $\mathcal{G}$, whose nodes are a finite set of scopes $\mathcal{S}(\mathcal{G})$. Every program has at least one scope, the global or *root* scope. Each scope S has an associated finite set $\mathcal{D}(S)$ of declarations and finite set $\mathcal{R}(S)$ of references (at particular program positions), and each declaration and reference in a program belongs to a unique scope. A scope is the atomic grouping for name resolution: roughly speaking, each reference x_i^{R} in a scope resolves to a declaration of the same variable x_j^{D} in the scope, if one exists. Intuitively, a single scope corresponds to a group of mutually recursive definitions, e.g., a **letrec** block, the declarations in a module, or the set of top-level bindings in a program. Below we will see that edges between nodes in a scope graph determine visibility of declarations in one scope from references in another scope.

Name Resolution. We write $\mathcal{R}(\mathcal{G})$ and $\mathcal{D}(\mathcal{G})$ for the (finite) sets of all references and all declarations, respectively, in the program with scope graph $\mathcal{G}$. Name resolution is specified by a relation $\longmapsto \subseteq \mathcal{R}(\mathcal{G}) \times \mathcal{D}(\mathcal{G})$ between references and corresponding declarations in $\mathcal{G}$. In the absence of edges, this relation is very simple:

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad x_j^{\mathsf{D}} \in \mathcal{D}(S)}{x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \qquad (X_0)$$

That is, a reference x_i^{R} resolves to a declaration x_j^{D}, if the scope S in which x_i^{R} is contained also contains x_j^{D}. We say that there is a *resolution path* from x_i^{R} to x_j^{D}. We will see soon that paths will grow beyond the one step relation defined by the rule above.

Scope Graph Diagrams. It can be illuminating to depict a scope graph graphically. In a scope graph diagram, a scope is depicted as a circle, a reference as a box with an arrow pointing *into* the scope that contains it, and a declaration as

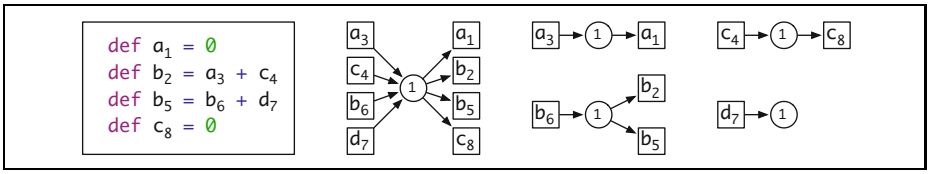

Fig. 5. Declarations and references in global scope

a box with an arrow *from* the scope that contains it. Fig. 5 shows an LM program consisting of a set of mutually-recursive global definitions; its scope graph; the resolution paths for variables `a`, `b`, and `c`; and an incomplete resolution path for variable `d`. In concrete example programs and scope diagrams we write both x_i^{R} and x_i^{D} as x_i, relying on context to distinguish references and declarations. For example, in Fig. 5, all occurrences $\mathtt{b}_i$ denote the *same name* `b` at *different positions*. In scope diagrams, the numbers in scope circles are arbitrarily chosen, and are just used to identify different scopes so that we can talk about them.

Duplicate Declarations. It is possible for a scope to contain multiple references and/or declarations with the same name. For example, scope 1 in Fig. 5 has two declarations of the variable `b`. While the existence of multiple references is normal, multiple declarations may give rise to multiple resolutions. For example, the $\mathtt{b}_6$ reference in Fig. 5 resolves to *each* of the two declarations $\mathtt{b}_2$ and $\mathtt{b}_5$.

Typically, correct programs will not declare the same identifier at two different locations in the same scope, although some languages have constructs (e.g. or-patterns in OCaml [17]) that are most naturally modeled this way. But even when the existence of multiple resolutions implies an erroneous program, we want the resolution calculus to identify *all* these resolutions, since IDEs and other front-end tools need to be able to represent erroneous programs. For example, a rename refactoring should support consistent renaming of identifiers, even in the presence of ambiguities (see Section 6). The ability of our calculus to describe ambiguous resolutions distinguishes it from systems, such as nominal logic [4], that inherently require unambiguous resolution of references.

2.3 Lexical Scope

We model lexical scope by means of the *parent* relation on scopes. In a well-formed scope graph, each scope has at most one parent and the parent relation is well-founded. Formally, the partial function $\mathcal{P}(\_)$ maps a scope S to its *parent* scope $\mathcal{P}(S)$. Given a scope graph with parent relation we can define the notion of *reachable* and *visible* declarations in a scope.

Fig. 6 illustrates how the parent relation is used to model common lexical scope patterns. Lexical scoping is typically presented through nested regions in the abstract syntax tree, as illustrated by the nested boxes in Fig. 6. Expressions in inner boxes may refer to declarations in surrounding boxes, but not vice versa. Each of the scopes in the program is mapped to a scope (circle) in the scope graph. The three scopes correspond to the global scope, the scope for **`fix`** $\mathtt{f}_2$, and the scope for **`fun`** $\mathtt{n}_3$. The edges from scopes to scopes correspond to the parent

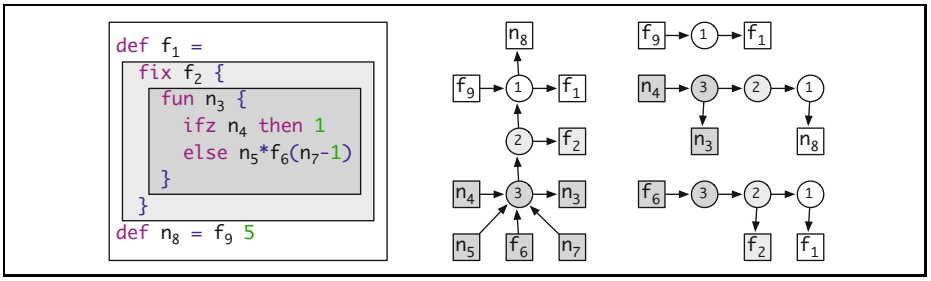

Fig. 6. Lexical scoping modeled by edges between scopes in the scope graph with example program, scope graph, and reachability paths for references

relation. The resolution paths on the right of Fig. 6 illustrate the consequences of the encoding. From reference `f6` both declarations `f1` and `f2` are *reachable*, but from reference `f9` only declaration `f1` is reachable. In languages with lexical scoping, the redeclaration of a variable inside a nested region typically *hides* the outer declaration. Thus, the duplicate declaration of variable `f` does not indicate a program error in this situation because only `f2` is *visible* from the scope of `f6`.

Reachability. The first step towards a full resolution calculus is to take into account reachability. We redefine rule (X_0) as follows:

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S_1) \quad p : S_1 \twoheadrightarrow S_2 \quad x_j^{\mathsf{D}} \in \mathcal{D}(S_2)}{p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \qquad (X_1)$$

That is, x_i^{R} in scope S_1 can be resolved to x_j^{D} in scope S_2, if S_2 is *reachable* from S_1, i.e. if $S_1 \twoheadrightarrow S_2$. Reachability is defined in terms of the parent relation as follows:

$$\frac{\mathcal{P}(S_1) = S_2}{\mathbf{P} : S_1 \longrightarrow S_2} \qquad \frac{}{[] : A \twoheadrightarrow A} \qquad \frac{s : A \longrightarrow B \quad p : B \twoheadrightarrow C}{s \cdot p : A \twoheadrightarrow C}$$

The parent relation between scopes gives rise to a direct edge $S_1 \longrightarrow S_2$ between child and parent scope, and $A \twoheadrightarrow B$ is the reflexive, transitive closure of the direct edge relation. In order to reason about the different ways in which a reference can be resolved, we record the resolution path p. For example, in Fig. 6 reference `f6` can be resolved with path $\mathbf{P}$ to declaration `f2` and with path $\mathbf{P} \cdot \mathbf{P}$ to `f1`.

Visibility. Under lexical scoping, multiple possible resolutions are not problematic, as long as the declarations reached are not declared in the same scope. A declaration is *visible* unless it is shadowed by a declaration that is 'closer by'. To formalize visibility, we first extend reachability of scopes to *reachability of declarations*:

$$\frac{x_i^{\mathsf{D}} \in \mathcal{D}(S') \quad p : S \twoheadrightarrow S'}{p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \qquad (R_2)$$

That is, a declaration x_i^{D} in S' is reachable from scope S ($S \rightarrowtail x_i^{\mathsf{D}}$), if scope S' is reachable from S.

Given multiple reachable declarations, which one should we prefer? A reachable declaration x_i^{D} is *visible* in scope S ($S \longmapsto x_i^{\mathsf{D}}$) if there is no other declaration for the same name that is reachable through a *more specific* path:

$$\frac{p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p'(p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{p : S \longmapsto x_i^{\mathsf{D}}} \qquad (V_2)$$

where the *specificity ordering* $p' < p$ on paths is defined as

$$\frac{}{\mathbf{D}(\_) < \mathbf{P}} \qquad \frac{s_1 < s_2}{s_1 \cdot p_1 < s_2 \cdot p_2} \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2}$$

That is, a path with fewer parent transitions is more specific than a path with more parent transitions. This formalizes the notion that a declaration in a "nearer" scope shadows a declaration in a "farther" scope.

Finally, a reference resolves to a declaration if that declaration is visible in the scope of the reference.

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad p : S \longmapsto x_j^{\mathsf{D}}}{p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \qquad (X_2)$$

Example. In Fig. 6 the scope (labeled 3) containing reference $\mathtt{f}_6$ can reach two declarations for $\mathtt{f}$: $\mathbf{P} \cdot \mathbf{D}(\mathtt{f}_2^{\mathsf{D}}) : S_3 \rightarrowtail \mathtt{f}_2^{\mathsf{D}}$ and $\mathbf{P} \cdot \mathbf{P} \cdot \mathbf{D}(\mathtt{f}_1^{\mathsf{D}}) : S_3 \rightarrowtail \mathtt{f}_1^{\mathsf{D}}$. Since the first path is more specific than the second path, only $\mathtt{f}_2$ is visible, i.e. $\mathbf{P} \cdot \mathbf{D}(\mathtt{f}_2^{\mathsf{D}}) : S_3 \longmapsto \mathtt{f}_2^{\mathsf{D}}$. Therefore $\mathtt{f}_6$ resolves to $\mathtt{f}_2$, i.e. $\mathbf{P} \cdot \mathbf{D}(\mathtt{f}_2^{\mathsf{D}}) : \mathtt{f}_6^{\mathsf{R}} \longmapsto \mathtt{f}_2^{\mathsf{D}}$.

Scopes, Revisited. Now that we have defined the notions of reachability and visibility, we can give a more precise description of the sense in which scopes "behave uniformly" with respect to resolution. For every scope S:

- Each declaration in the program is either visible at every reference in $\mathcal{R}(S)$ or not visible at any reference in $\mathcal{R}(S)$.
- For each reference in the program, either every declaration in $\mathcal{D}(S)$ is reachable from that reference, or no declaration in $\mathcal{D}(S)$ is reachable from that reference.
- Every declaration in $\mathcal{D}(S)$ is visible at every reference in $\mathcal{R}(S)$.

2.4 Imports

Introducing modules and imports complicates the name binding picture. Declarations are no longer visible only through the lexical context, but may be visible through an import as well. Furthermore, resolving a reference may require first resolving one or more imports, which may in turn require resolving further imports, and so on.

We model an *import* by means of a reference x_i^{R} in the set of imports $\mathcal{I}(S)$ of a scope S. (Imports are also always references and included in some $\mathcal{R}(S')$, but not

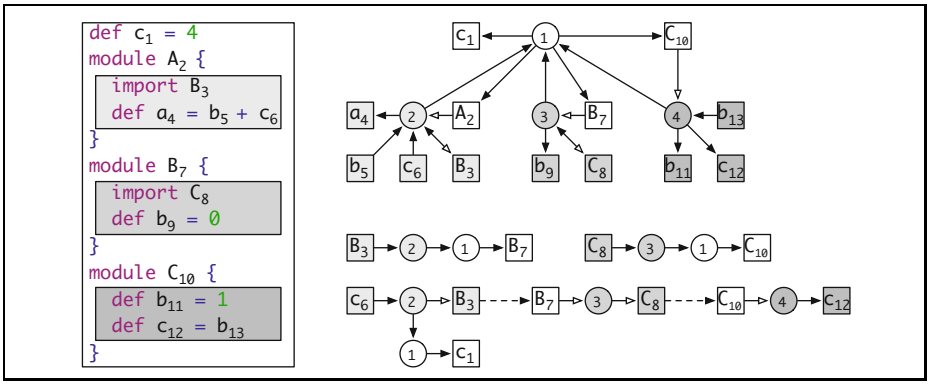

Fig. 7. Modules and imports with example program, scope graph, and reachability paths for references

necessarily in the same scope in which they are imports.) We model a *module* by associating a scope S with a declaration $x_i^{\mathsf{D}}{:}S$. This associated *named scope* (i.e., named by x) represents the declarations introduced by, and encapsulated in, the module. (We write the $:S$ only in rules where it is required; where we omit it, the declaration may or may not have an associated scope.) Thus, *importing* entails resolving the import reference to a declaration and making the declarations in the scope associated with that declaration available in the importing scope.

Note that 'module' is not a built-in concept in our framework. A module is any construct that (1) is named, (2) has an associated scope that encapsulates declarations, and (3) can be imported into another scope. Of course, this can be used to model the module systems of languages such as ML. But it can be applied to constructs that are not modules at first glance. For example, a class in Java encapsulates class variables and methods, which are imported into its subclasses through the 'extends' clause. Thus, a class plays the role of module and the extends clause that of import. We discuss further applications in Section 3.

Reachability. To define name resolution in the presence of imports, we first extend the definition of reachability. We saw above that the parent relation on scopes induces an edge $S_1 \longrightarrow S_2$ between a scope S_1 and its parent scope S_2 in the scope graph. Similarly, an import induces an edge $S_1 \longrightarrow S_2$ between a scope S_1 and the scope S_2 associated with a declaration imported into S_1:

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \quad p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2} \qquad (I_3)$$

Note the recursive invocation of the resolution relation on the name of the imported scope.

Figure 7 illustrates extensions to scope graphs and paths to describe imports. Association of a name to a scope is indicated by an open-headed arrow from the name declaration box to the scope circle. (For example, scope 2 is associated to declaration $\mathtt{A}_2$.) An import into a scope is indicated by an open-headed arrow from the scope circle to the import name reference box. (For example, scope 2

imports the contents of the scope associated to the resolution of reference $\mathtt{B}_3$; note that since $\mathtt{B}_3$ is also a reference within scope 2, there is also an ordinary arrow in the opposite direction, leading to a double-headed arrow in the scope graph.) Edges in reachability paths representing the resolution of imported scope names to their definitions are drawn dashed. (For example, reference $\mathtt{B}_3$ resolves to declaration $\mathtt{B}_7$, which has associated scope 3.) The paths at the bottom right of the figure illustrate that the scope (labeled 2) containing reference $\mathtt{c}_6$ can reach two declarations for c: $\mathbf{P} \cdot \mathbf{D}(\mathtt{c}_1^{\mathsf{D}}) : S_2 \rightarrowtail \mathtt{c}_1^{\mathsf{D}}$ and $\mathbf{I}(\mathtt{B}_3^{\mathsf{R}}, \mathtt{B}_7^{\mathsf{D}}{:}S_3) \cdot \mathbf{I}(\mathtt{C}_8^{\mathsf{R}}, \mathtt{C}_{10}^{\mathsf{D}}{:}S_4) \cdot \mathbf{D}(\mathtt{c}_{12}^{\mathsf{D}}) : S_2 \rightarrowtail \mathtt{c}_{12}^{\mathsf{D}}$, making use of the subsidiary resolutions $\mathtt{B}_3^{\mathsf{R}} \longmapsto \mathtt{B}_7^{\mathsf{D}}$ and $\mathtt{C}_8^{\mathsf{R}} \longmapsto \mathtt{C}_{10}^{\mathsf{D}}$.

Visibility. Imports cause new kinds of ambiguities in resolution paths, which require extension of the visibility policy.

The first issue is illustrated by Fig. 8. In the scope of reference $\mathtt{b}_{10}$ we can reach declaration $\mathtt{b}_7$ with path $\mathbf{D}(\mathtt{b}_7^{\mathsf{D}})$ and declaration $\mathtt{b}_4$ with path $\mathbf{I}(A_6^{\mathsf{R}}, A_2^{\mathsf{D}} : S_A) \cdot \mathbf{D}(\mathtt{b}_4^{\mathsf{D}})$ (where S_A is the scope named by declaration A_2). We resolve this conflict by extending the specificity order with the rule $\mathbf{D}(\_) < \mathbf{I}(\_, \_)$. That is, local declarations override imported declarations. Similarly, in the scope of reference $\mathtt{a}_8$ we can reach declaration $\mathtt{a}_1$ with path $\mathbf{P} \cdot \mathbf{D}(\mathtt{a}_1^{\mathsf{D}})$ and declaration $\mathtt{a}_3$ with path $\mathbf{I}(A_6^{\mathsf{R}}, A_2^{\mathsf{D}}{:}S_A) \cdot \mathbf{D}(\mathtt{a}_3^{\mathsf{D}})$. We resolve this conflict by extending the specificity order with the rule $\mathbf{I}(\_, \_) < \mathbf{P}$. That is, resolution through imports is preferred over resolution through parents. In other words, declarations in imported modules override declarations in lexical parents.

```
def a1 = ...
module A2 {
  def a3 = ...
  def b4 = ...
}
module C5 {
  import A6
  def b7 = a8
  def c9 = b10
}
```

Fig. 8. Parent vs Import

```
def a1 = ...
module B2 {
}
module C3 {
  def a4 = ...
  module D5 {
    import B6
    def e7 = a8
  }
}
```

Fig. 9. Parent of import

The next issue is illustrated in Fig. 9. In the scope of reference $\mathtt{a}_8$ we can reach declaration $\mathtt{a}_4$ with path $\mathbf{P} \cdot \mathbf{D}(\mathtt{a}_4^{\mathsf{D}})$ and declaration $\mathtt{a}_1$ with path $\mathbf{P} \cdot \mathbf{P} \cdot \mathbf{D}(\mathtt{a}_1^{\mathsf{D}})$. The specificity ordering guarantees that only the first of these is visible, giving the resolution we expect. However, with the rules as stated so far, there is another way to reach $\mathtt{a}_1$, via the path $\mathbf{I}(\mathtt{B}_6^{\mathsf{R}}, \mathtt{B}_2^{\mathsf{D}}{:}S_B) \cdot \mathbf{P} \cdot \mathbf{D}(\mathtt{a}_1^{\mathsf{D}})$. That is, we first import module B, and then go to its lexical parent, where we find the declaration. In other words, when importing a module, we import not just its declarations, but all declarations in its lexical context. This behavior seems undesirable; to our knowledge, no real languages exhibit it. To rule out such resolutions, we define a well-formedness predicate $WF(p)$ that requires paths p to be of the form $\mathbf{P}^* \cdot \mathbf{I}(\_, \_)^*$, i.e. forbidding the use of parent steps after one or more import steps. We use this predicate to restrict the reachable declarations relation by only considering scopes reachable through a well-formed path:

$$\frac{x_i^{\mathsf{D}} \in \mathcal{D}(S') \quad p : S \twoheadrightarrow S' \quad WF(p)}{p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \qquad (R_3)$$

$$\dfrac{\dfrac{\dfrac{A_2^{\mathsf{D}}{:}S_{A_2} \in \mathcal{D}(S_{A_1}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{I}(S_{root}) \qquad \dfrac{A_4^{\mathsf{R}} \in \mathcal{R}(S_{root}) \qquad A_1^{\mathsf{D}}{:}S_{A_1} \in \mathcal{D}(S_{root})}{A_4^{\mathsf{R}} \longmapsto A_1^{\mathsf{D}}{:}S_{A_1}}}{S_{root} \longrightarrow S_{A_1} \quad (*)}}{S_{root} \rightarrowtail A_2^{\mathsf{D}}{:}S_{A_2}} \qquad A_4^{\mathsf{R}} \in \mathcal{R}(S_{root})}{S_{root} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}} \qquad A_4^{\mathsf{R}} \in \mathcal{R}(S_{root})}{A_4^{\mathsf{R}} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}}$$

Fig. 10. Derivation for $A_4^{\mathsf{R}} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}$ in a calculus without import tracking

The complete definition of well-formed paths and specificity order on paths is given in Fig. 2. In Section 2.5 we discuss how alternative visibility policies can be defined by just changing the well-formedness predicate and specificity order.

Seen Imports. Consider the example in Fig. 11. Is declaration $\mathtt{a}_3$ reachable in the scope of reference $\mathtt{a}_6$? This reduces to the question whether the import of $\mathtt{A}_4$ can resolve to module $\mathtt{A}_2$. Surprisingly, it can, in the calculus as discussed so far, as shown by the derivation in Fig. 10 (which takes a few shortcuts). The conclusion of the derivation is that $A_4^{\mathsf{R}} \longmapsto A_2^{\mathsf{D}}{:}S_{A_2}$. This conclusion is obtained by *using the import at* $\mathtt{A}_4$ to conclude at step (*) that $S_{root} \longrightarrow S_{A_1}$, i.e. that the body of module $\mathtt{A}_1$ is reachable! In other words, the import of $\mathtt{A}_4$ is used in its own resolution. Intuitively, this is nonsensical.

```
module A1 {
  module A2 {
    def a3 = ...
  }
}
import A4
def b5 = a6
```

Fig. 11. Self import

To rule out this kind of behavior we extend the calculus to keep track of the set of *seen imports* $\mathbb{I}$ using judgements of the form $\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$. We need to extend all rules to pass the set $\mathbb{I}$, but only the rules for resolution and import are truly affected:

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

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \setminus \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbb{I} \vdash \mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2} \qquad (I)$$

```
module A1 {
  module B2 {
    def x3 = 1
  }
}
module B4 {
  module A5 {
    def y6 = 2
  }
}
module C7 {
  import A8
  import B9
  def z10 = x11
          + y12
}
```

Fig. 12. Anomalous resolution

With this final ingredient, we reach the full calculus in Fig. 3. It is not hard to see that the resolution relation is well-founded. The only recursive invocation (via the I rule) uses a strictly larger set $\mathbb{I}$ of seen imports (via the X rule); since the set $\mathcal{R}(G)$ is finite, $\mathbb{I}$ cannot grow indefinitely.

Anomalies. Although the calculus produces the desired resolutions for a wide variety of real language constructs, its behavior can be surprising on corner cases. Even with the "seen imports" mechanism, it is still possible for a single derivation

to resolve a given import in two different ways, leading to unintuitive results. For example, in the program in Fig. 12, x_{11} can resolve to x_3 and y_{12} can resolve to y_6. (Derivations left as an exercise to the curious reader!) In our experience, phenomena like this occur only in the presence of mutually-recursive imports; to our knowledge, no real language has these (perhaps for good reason). We defer deeper exploration of these anomalies to future work.

2.5 Variants

The resolution calculus presented so far reflects a number of binding policy decisions. For example, we enforce imports to be transitive and local declarations to be preferred over imports. However, not every language behaves like this. We now present how other common behaviors can easily be represented with slight modifications of the calculus. Indeed, the modifications do not have to be done on the calculus itself (the $\longrightarrow$, $\twoheadrightarrow$, $\rightarrowtail$ and $\longmapsto$ relations) but can simply be encoded in the *WF* predicate and the $<$ ordering on paths.

Reachability policy. Reachability policies define how a reference can access a particular definition, i.e. what rules can be used during the resolution. We can change our reachability policy by modifying the *WF* predicate. For example, if we want to rule out transitive imports, we can change *WF* to be

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_, \_)?$$

where ? denotes the *at most one* operation on regular expressions. Therefore, an import can only be used once at the end of the chain of scopes.

For a language that supports both transitive and non-transitive imports, we can add a label on references corresponding to imports. If x^{R} is a reference representing a non-transitive import and x^{TR} a reference corresponding to a transitive import, then the *WF* predicate simply becomes:

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_^{\mathsf{TR}}, \_)^* \cdot \mathbf{I}(\_^{\mathsf{R}}, \_)?$$

Now no import can occur after the use of a non-transitive one.

Similarly, we can modify the rule to handle the *Export* declaration in Coq, which forces transitivity (a resolution can always use an exported module even after importing from a non-transitive one). Assume x^{R} is a reference representing a non-transitive import and x^{ER} a reference corresponding to an export; then we can use the following predicate:

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_^{\mathsf{R}}, \_)? \cdot \mathbf{I}(\_^{\mathsf{ER}}, \_)^*$$

```
module A1 {
  def x2 = 3
}
module B3 {
  include A4;
  def x5 = 6;
  def z6 = x7
}
```

Fig. 13. Include

Visibility policy. We can modify the visibility policy, i.e. how resolutions shadow each other, by changing the definition of the specificity ordering. For example, we might want imports to act like textual inclusion, so the declarations in the included module have the same precedence as local declarations. This is similar to Standard ML's **include** mechanism. In the program in Fig. 13, the reference x_7 should be treated as having duplicate resolutions, to either x_5 or x_2; the

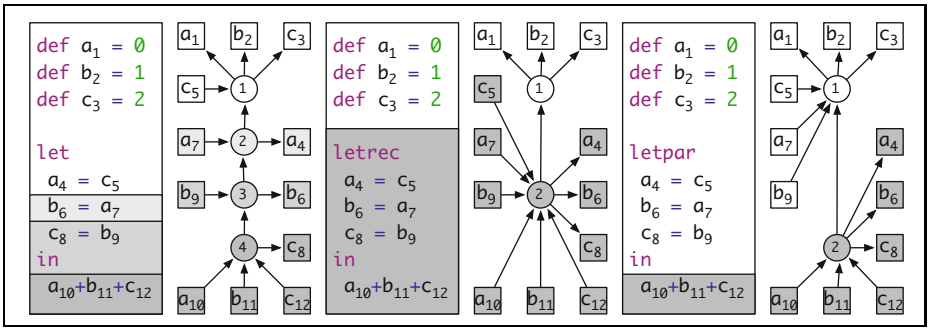

Fig. 14. Example LM programs with sequential, recursive, and parallel **let**, and their encodings as scope graphs

former should not hide the latter. To handle this situation, we can drop the rule $\mathbf{D}(\_) < \mathbf{I}(\_, \_)$ so that definitions and references will get the same precedence, and a definition will not shadow an imported definition. To handle both **include** and ordinary imports, we can once again differentiate the references, and define different ordering rules depending on the reference used in the import step.

3 Coverage

To what extent does the scope graph framework cover name binding systems that live in the world of real programming languages? It is not possible to *prove* complete coverage by the framework, in the sense of being able to encode all possible name binding systems that exist or may be designed in the future. (Indeed, given that these systems are typically implemented in compilers with algorithms in Turing-complete programming languages, the framework is likely *not* to be complete.) However, we believe that our approach handles many lexically-scoped languages. The design of the framework was informed by an investigation of a wide range of name binding patterns in existing languages, their (attempted) formalization in the NaBL name binding language [14,16], and their encoding in scope graphs. In this section, we discuss three such examples: **let** bindings, qualified names, and inheritance in Java. This should provide the reader with a good sense of how name binding patterns can be expressed using scope graphs. Appendix A of [19] provides further examples, including definition-before-use, compilation units and packages in Java, and namespaces and partial classes in C#.

Let Bindings. The several flavors of **let** bindings in languages such as ML, Haskell, and Scheme do not follow the unary lexical binding pattern in which the binding construct dominates the abstract syntax tree that makes up its scope. The LM language from Fig. 4 has three flavors of **let** bindings: sequential, recursive, and parallel **let**, each with a list of bindings and a body expression. Fig. 14 shows the encoding into scope graphs for each of the constructs and makes precise how the bindings are interpreted in each flavour. In the recursive **letrec**, the bindings are visible in all initializing expressions, so a single scope

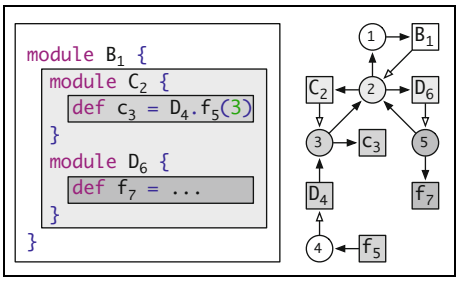

Fig. 15. Example LM program with partially-qualified name

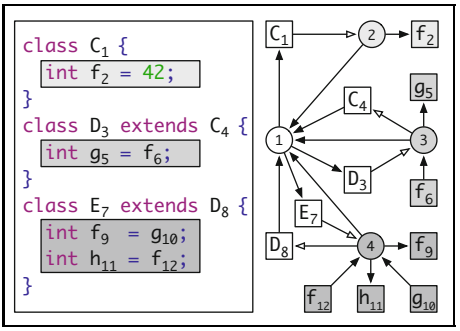

Fig. 16. Class inheritance in Java modeled by import edges

suffices for the whole construct. In the sequential **`let`**, each binding is visible in the *subsequent* bindings, but not in its own initializing expression. This requires the introduction of a new scope for each binding. In the parallel **`letpar`**, the variables being bound are not visible in any of the initializing expressions, but only in the body. This is expressed by means of a single scope (2) in which the bindings are declared; any references in the initializing expressions are associated to the parent scope (1).

Qualified Names. Qualified names refer to declarations in named scopes outside the lexical scoping. They can be either used as simple references or as imports. For example, fully-qualified names of Java classes can be used to refer to (or import) classes from other packages. While fully-qualified names allow navigating named scopes from the root scope, partially-qualified names give access to lexical subscopes, which are otherwise hidden from lexical parent scopes.

The LM program in Fig. 15 uses a partially-qualified name `D.f` to access function `f` in submodule `D`. We can model this pattern using an anonymous scope (4), which is not linked to the lexical context. The relative name ($\texttt{f}_5$) is a reference in the anonymous scope. We add the qualifying scope name ($\texttt{D}_4$) as an import in the anonymous scope.

Inheritance in Java. We can model inheritance in object-oriented languages with named scopes and imports. For example, Fig. 16 shows a hierarchy of three Java classes. Class `C` declares a field `f`. Class `D` extends `C` and inherits its field `f`. Class `E` extends `D`, inheriting the fields of `C` and `D`. Each class name is a declaration in the same package scope (1), and associated with the scope of its class body. Inheritance is modeled with imports: a subclass body scope contains an import referring to its super class, making the declarations in the super class reachable from the body. In the example, the scope (4) representing the body of class `E` contains an import referring to its super class `D`. Using this import, $\texttt{g}_{10}$ correctly resolves to $\texttt{g}_5$. Since local declarations hide imported declarations, $\texttt{f}_{12}$ also refers correctly to the local declaration $\texttt{f}_9$, which hides the transitively

$$
\begin{array}{rcl}
[\![\texttt{ds}]\!]^{prog} & := & \mathsf{let}\ S := \mathsf{new}_{\bot}\ \mathsf{in}\ [\![\texttt{ds}]\!]^{recd}_{S} \\
[\![\texttt{d ds}]\!]^{recd}_{S} & := & [\![\texttt{d}]\!]^{dec}_{S}; [\![\texttt{ds}]\!]^{recd}_{S} \\
[\![]\!]^{recd}_{S} & := & () \\
[\![\textbf{module}\ \texttt{x}_i\texttt{\{ds\}}]\!]^{dec}_{S} & := & \mathsf{let}\ S' := \mathsf{new}_{S}\ \mathsf{in}\ \mathcal{D}(S) \mathrel{+}= x_i^{\mathsf{D}}{:}S'; [\![\texttt{ds}]\!]^{recd}_{S'} \\
[\![\textbf{import}\ \texttt{xs}]\!]^{dec}_{S} & := & [\![\texttt{xs}]\!]^{rqid}_{S}; [\![\texttt{xs}]\!]^{iqid}_{S} \\
[\![\textbf{def}\ \texttt{x}_i\ \texttt{=}\ \texttt{e}]\!]^{dec}_{S} & := & \mathcal{D}(S) \mathrel{+}= x_i^{\mathsf{D}}; [\![\texttt{e}]\!]^{exp}_{S} \\
[\![\texttt{xs}]\!]^{exp}_{S} & := & [\![\texttt{xs}]\!]^{rqid}_{S} \\
[\![\texttt{(}\textbf{fun}\ \texttt{|}\ \textbf{fix}\texttt{)}\ \texttt{x}_i\texttt{\{e\}}]\!]^{exp}_{S} & := & \mathsf{let}\ S' := \mathsf{new}_{S}\ \mathsf{in}\ \mathcal{D}(S') \mathrel{+}= x_i^{\mathsf{D}}; [\![\texttt{e}]\!]^{exp}_{S'} \\
[\![\textbf{letrec}\ \texttt{bs}\ \textbf{in}\ \texttt{e}]\!]^{exp}_{S} & := & \mathsf{let}\ S' := \mathsf{new}_{S}\ \mathsf{in}\ [\![\texttt{bs}]\!]^{recb}_{S'}; [\![\texttt{e}]\!]^{exp}_{S'} \\
[\![\textbf{letpar}\ \texttt{bs}\ \textbf{in}\ \texttt{e}]\!]^{exp}_{S} & := & \mathsf{let}\ S' := \mathsf{new}_{S}\ \mathsf{in}\ [\![\texttt{bs}]\!]^{parb}_{(S,S')}; [\![\texttt{e}]\!]^{exp}_{S'} \\
[\![\textbf{let}\ \texttt{bs}\ \textbf{in}\ \texttt{e}]\!]^{exp}_{S} & := & \mathsf{let}\ S' := [\![bs]\!]^{seqb}_{S}\ \mathsf{in}\ [\![\texttt{e}]\!]^{exp}_{S'} \\
[\![\texttt{e}_1\ \texttt{e}_2]\!]^{exp}_{S} & := & [\![\texttt{e}_1]\!]^{exp}_{S}; [\![\texttt{e}_2]\!]^{exp}_{S} \\
[\![\texttt{e}_1 \oplus \texttt{e}_2]\!]^{exp}_{S} & := & [\![\texttt{e}_1]\!]^{exp}_{S}; [\![\texttt{e}_2]\!]^{exp}_{S} \\
[\![\texttt{n}]\!]^{exp}_{S} & := & () \\
[\![\texttt{x}_i\texttt{.xs}]\!]^{rqid}_{S} & := & \mathcal{R}(S) \mathrel{+}= x_i^{\mathsf{R}}; \mathsf{let}\ S' := \mathsf{new}_{\bot}\ \mathsf{in}\ \mathcal{I}(S') \mathrel{+}= x_i^{\mathsf{R}}; [\![xs]\!]^{rqid}_{S'} \\
[\![\texttt{x}_i]\!]^{rqid}_{S} & := & \mathcal{R}(S) \mathrel{+}= x_i^{\mathsf{R}} \\
[\![\texttt{x}_i\texttt{.xs}]\!]^{iqid}_{S} & := & [\![xs]\!]^{iqid}_{S} \\
[\![\texttt{x}_i]\!]^{iqid}_{S} & := & \mathcal{I}(S) \mathrel{+}= x_i^{\mathsf{R}} \\
[\![\texttt{x}_i\ \texttt{=}\ \texttt{e;}\ \texttt{bs}]\!]^{recb}_{S} & := & \mathcal{D}(S) \mathrel{+}= x_i^{\mathsf{D}}; [\![\texttt{e}]\!]^{exp}_{S}; [\![\texttt{bs}]\!]^{recb}_{S} \\
[\![]\!]^{recb}_{S} & := & () \\
[\![\texttt{x}_i\ \texttt{=}\ \texttt{e;}\ \texttt{bs}]\!]^{parb}_{(S,S')} & := & \mathcal{D}(S') \mathrel{+}= x_i^{\mathsf{D}}; [\![\texttt{e}]\!]^{exp}_{S}; [\![\texttt{bs}]\!]^{parb}_{(S,S')} \\
[\![]\!]^{parb}_{(S,S')} & := & () \\
[\![\texttt{x}_i\ \texttt{=}\ \texttt{e;}\ \texttt{bs}]\!]^{seqb}_{S} & := & [\![\texttt{e}]\!]^{exp}_{S}; \mathsf{let}\ S' := \mathsf{new}_{S}\ \mathsf{in}\ \mathcal{D}(S') \mathrel{+}= x_i^{\mathsf{D}}; \mathsf{ret}(S') \\
[\![]\!]^{seqb}_{S} & := & \mathsf{ret}(S)
\end{array}
$$

Fig. 17. Scope graph construction for LM via syntax-directed AST traversal

imported $\texttt{f}_2$. Note that since a scope can contain several imports, encoding multiple inheritance uses exactly the same principle.

4 Scope Graph Construction

The preceding sections have illustrated scope graph construction by means of examples corresponding to various language features. Of course, to apply our formalism in practice, one must be able to construct scope graphs systematically. Ultimately, we would like to be able to specify this process for arbitrary languages using a generic binding specification language such as NaBL [16], but that remains future work. Here we illustrate systematic scope graph construction for arbitrary programs in a *specific* language, LM (Fig. 4), via straightforward syntax-directed traversal.

Figure 17 describes the construction algorithm. For clarity of presentation, the algorithm traverses the program's concrete syntax; a real implementation would traverse the program's AST. The algorithm is presented in an *ad hoc* imperative

language, explained here. The traversal is specified as a collection of (potentially) mutually recursive functions, one or more for each syntactic class of LM. Each function f is defined by a set of clauses $[\![pattern]\!]^f_{args}$. When f is invoked on a term, the clause whose *pattern* matches the term is executed. Functions may also take additional arguments $args$. Each clause body consists of a sequence of statements separated by semicolons. Functions can optionally return a value using ret(). The let statement binds a metavariable in the remainder of the clause body. An empty clause body is written ().

The algorithm is initiated by invoking $[\![\_]\!]^{prog}$ on an entire LM program. Its net effect is to produce a scope graph via a sequence of imperative operations. The construct new_P creates a new scope S with parent P (or no parent if $p = \bot$) and empty sets $\mathcal{D}(S)$, $\mathcal{R}(S)$, and $\mathcal{I}(S)$. These sets are subsequently populated using the $+=$ operator, which extends a set imperatively. The program scope graph is simply the set of scopes that have been created and populated when the traversal terminates.

5 Resolution Algorithm

The calculus of Section 2 gives a precise definition of resolution. In principle, we can search for derivations in the calculus to answer questions such as "Does this variable reference resolve to this declaration?" or "Which variable declarations does this reference resolve to?" But automating this search process is not trivial, because of the need for back-tracking and because the paths in reachability derivations can have cycles (visiting the same scope more than once), and hence can grow arbitrarily long.

In this section we describe a deterministic and terminating *algorithm* for computing resolutions, which provides a practical basis for implementing tools based on scope graphs, and prove that it is sound and complete with respect to the calculus. This algorithm also connects the calculus, which talks about resolution of a single variable at a time, to more conventional descriptions of binding which use "environments" or "contexts" to describe *all* the visible or reachable declarations accessible from a program location.

For us, an *environment* is just a set of declarations x^{D}_i. This can be thought of as a function from identifiers to (possible empty) sets of declaration positions. (In this paper, we leave the representation of environments abstract; in practice, one would use a hash table or other dictionary data structure.) We construct an atomic environment corresponding to the declarations in each scope, and then combine atomic environments to describe the sets of reachable and visible declarations resulting from the parent and import relations. The key operator for combining environments is *shadowing*, which returns the union of the declarations in two environments restricted so that if a variable x has any declarations in the first environment, no declarations of x are included from the second environment. More formally:

Definition 1 (Shadowing). *For any environments E_1, E_2, we write:*

$$E_1 \triangleleft E_2 := E_1 \cup \{x^{\mathsf{D}}_i \in E_2 \mid \nexists\, x^{\mathsf{D}}_{i'} \in E_1\}$$

$$
\begin{aligned}
Res[\mathbb{I}](x_i^{\mathsf{R}}) &:= \{x_j^{\mathsf{D}} \mid \exists S \; s.t. \; x_i^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_j^{\mathsf{D}} \in Env_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](S)\} \\
Env_V[\mathbb{I}, \mathbb{S}](S) &:= Env_L[\mathbb{I}, \mathbb{S}](S) \triangleleft Env_P[\mathbb{I}, \mathbb{S}](S) \\
Env_L[\mathbb{I}, \mathbb{S}](S) &:= Env_D[\mathbb{I}, \mathbb{S}](S) \triangleleft Env_I[\mathbb{I}, \mathbb{S}](S) \\
Env_D[\mathbb{I}, \mathbb{S}](S) &:= \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \mathcal{D}(S) \end{cases} \\
Env_I[\mathbb{I}, \mathbb{S}](S) &:= \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \bigcup \left\{ Env_L[\mathbb{I}, \{S\} \cup \mathbb{S}](S_y) \mid y_i^{\mathsf{R}} \in \mathcal{I}(S) \setminus \mathbb{I} \wedge y_j^{\mathsf{D}} : S_y \in Res[\mathbb{I}](y_i^{\mathsf{R}}) \right\} \end{cases} \\
Env_P[\mathbb{I}, \mathbb{S}](S) &:= \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ Env_V[\mathbb{I}, \{S\} \cup \mathbb{S}](\mathcal{P}(S)) \end{cases}
\end{aligned}
$$

Fig. 18. Resolution algorithm

Figure 18 specifies an algorithm $Res[\mathbb{I}](x_i^{\mathsf{R}})$ for resolving a reference x_i^{R} to a set of corresponding declarations x_j^{D}. Like the calculus, the algorithm avoids trying to use an import to resolve itself by maintaining a set $\mathbb{I}$ of "already seen" imports. The algorithm works by computing the full environment $Env_V[\mathbb{I}, \mathbb{S}](S)$ of declarations that are visible in the scope S containing x_i^{R}, and then extracting just the declarations for x. The full environment, in turn, is built from the more basic environments Env_D of immediate declarations, Env_I of imported declarations, and Env_P of lexically enclosing declarations, using the shadowing operator. The order of construction matches both the *WF* restriction from the calculus, which prevents the use of parent after an import, and the path ordering $<$, which prefers immediate declarations over imports and imports over declarations from the parent scope. (Note that the algorithm does *not* work for the variants of *WF* and $<$ described in Section 2.5.) A key difference from the calculus is that the shadowing operator is applied at each stage in environment construction, rather than applying the visibility criterion just once at the "top level" as in calculus rule V. This difference is a natural consequence of the fact that the algorithm computes sets of declarations rather than full derivation paths, so it does not maintain enough information to delay the visibility computation.

Termination The algorithm is terminating using the well-founded lexicographic measure $(|\mathcal{R}(\mathcal{G}) \setminus \mathbb{I}|, |\mathcal{S}(\mathcal{G}) \setminus \mathbb{S}|)$. Termination is straightforward by unfolding the calls to Res in Env_I and then inlining the definitions of Env_V and Env_L: this gives an equivalent algorithm in which the measure strictly decreases at every recursive call.

5.1 Correctness of Resolution Algorithm

The resolution algorithm is sound and complete with respect to the calculus.

Theorem 1. $\forall\, \mathbb{I}, x_i^{\mathsf{R}}, j, (x_j^{\mathsf{D}} \in Res[\mathbb{I}](x_i^{\mathsf{R}})) \iff (\exists p \; s.t. \; \mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}})$.

We sketch the proof of this theorem here; details of the supporting lemmas and proofs are in Appendix B of [19]. To begin with, we must deal with the

Transitive closure

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

$$\frac{\mathbb{I} \vdash s : A \longrightarrow B \quad B \notin \mathbb{S} \quad \mathbb{I}, \{B\} \cup \mathbb{S} \vdash p : B \twoheadrightarrow C}{\mathbb{I},\mathbb{S} \vdash s \cdot p : A \twoheadrightarrow C} \quad (T')$$

Reachable declarations

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

Visible declarations

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

Reference resolution

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \quad (X')$$

Fig. 19. "Primed" resolution calculus with "seen scopes" component

fact that the calculus can generate reachability derivations with cycles, but the algorithm does not follow cycles. In fact, *visibility* derivations cannot have cycles:

Lemma 1. *If* $\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$ *then* p *is cycle-free.*

We therefore begin by defining an alternative version of the calculus that prevents construction of cyclic paths. This alternative calculus consists of the original rules $(P), (I)$ from Figure 3 together with the new rules $(N'), (T'), (R'), (V'), (X')$ from Figure 19. The new rules describe transitions that include a "seen scopes" component $\mathbb{S}$ which is used to enforce acyclicity of paths. By inspection, this is the only difference between the "primed" system and original one. Thus, by Lemma 1, we have

Lemma 2. $\forall \mathbb{I}, \mathbb{S}, x_i^{\mathsf{D}}, (\exists p \; s.t. \; \mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}) \iff (\exists p \; s.t. \; \mathbb{I}, \emptyset \vdash p : S \longmapsto x_i^{\mathsf{D}})$.

Hereinafter, we can work with the primed system.

Next we define a family of sets $\mathbb{P}$ of derivable paths in the (primed) calculus.

Definition 2 (Path Sets).

$$\begin{aligned}
\mathbb{P}_D[\mathbb{I},\mathbb{S}](S) &:= \{p \mid \exists \; x_i^{\mathsf{D}} \; s.t. \; p = \mathbf{D}(x_i^{\mathsf{D}}) \wedge \mathbb{I},\mathbb{S} \vdash p : S \rightarrowtail x_i^{\mathsf{D}}\} \\
\mathbb{P}_P[\mathbb{I},\mathbb{S}](S) &:= \{p \mid \exists \; p' \; x_i^{\mathsf{D}} \; s.t. \; p = \mathbf{P} \cdot p' \wedge \\
&\qquad \mathbb{I},\mathbb{S} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \wedge \mathbb{I}, \{S\} \cup \mathbb{S} \vdash p' : \mathcal{P}(S) \longmapsto x_i^{\mathsf{D}}\} \\
\mathbb{P}_I[\mathbb{I},\mathbb{S}](S) &:= \{p \mid \exists \; p' \; x_i^{\mathsf{D}} \; y_j^{\mathsf{R}} \; y_{j'}^{\mathsf{D}}{:}S' \; s.t. \; p = \mathbf{I}(y_j^{\mathsf{R}}, y_{j'}^{\mathsf{D}}{:}S') \cdot p' \wedge \\
&\qquad \mathbb{I},\mathbb{S} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \wedge \mathbb{I}, \{S\} \cup \mathbb{S} \vdash p' : S' \longmapsto x_i^{\mathsf{D}}\} \\
\mathbb{P}_L[\mathbb{I},\mathbb{S}](S) &:= \{p \mid \exists \; x_i^{\mathsf{D}} \; s.t. \; \mathbb{I},\mathbb{S} \vdash p : S \longmapsto x_i^{\mathsf{D}} \wedge p \in \mathbf{I}(_,_)^* \cdot \mathbf{D}(_)\} \\
\mathbb{P}_V[\mathbb{I},\mathbb{S}](S) &:= \{p \mid \exists \; x_i^{\mathsf{D}} \; s.t. \; \mathbb{I},\mathbb{S} \vdash p : S \longmapsto x_i^{\mathsf{D}}\}
\end{aligned}$$

These sets are designed to correspond to the various classes of environments Env_C. $\mathbb{P}_D$, $\mathbb{P}_P$, and $\mathbb{P}_I$ contain all reachability derivations starting with a $\mathbf{D}(\_)$, $\mathbf{P}$, or $\mathbf{I}(\_, \_)$ respectively, with the further condition that the *tail* of each derivation is a visibility derivation (i.e. is most specific among all reachability derivations). $\mathbb{P}_V$ describes the set of all visibility derivations. ($\mathbb{P}_L$ is similar, but omits paths including $\mathbf{P}$ steps, because well-formedness prevents using these steps after an import step.) For compactness, we state the key result uniformly over all classes of sets:

Definition 3. *For any path p, $\delta(p) := x_i^{\mathsf{D}}$ iff $\exists p'$ s.t. $p = p' \cdot \mathbf{D}(x_i^{\mathsf{D}})$ and for any set of paths P, $\Delta(P) := \{\delta(p) \mid p \in P\}$.*

Lemma 3. *For each class $C \in \{V, L, D, I, P\}$:*

$$\forall\ \mathbb{I}\ \mathbb{S}\ S, Env_c[\mathbb{I}, \mathbb{S}](S) = \Delta(\mathbb{P}_C[\mathbb{I}, \mathbb{S}](S))$$

Proof. We first prove two auxiliary lemmas about reachability and visibility after one step:

$$\forall\ \mathbb{I}\ \mathbb{S}\ s\ p\ S\ x_i^{\mathsf{D}}, (\mathbb{I}, \mathbb{S} \vdash s \cdot p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}} \Longrightarrow \mathbb{I}, \{S\} \cup \mathbb{S} \vdash s : S \longrightarrow S' \Longrightarrow \mathbb{I}, \{S\} \cup \mathbb{S} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S' \rightarrowtail x_i^{\mathsf{D}}) \quad (\Diamond)$$

$$\forall\ \mathbb{I}\ \mathbb{S}\ s\ p\ S\ x_i^{\mathsf{D}}, (\mathbb{I}, \mathbb{S} \vdash s \cdot p : S \longmapsto x_i^{\mathsf{D}} \Longrightarrow \mathbb{I}, \{S\} \cup \mathbb{S} \vdash s : S \longrightarrow S' \Longrightarrow \mathbb{I}, \{S\} \cup \mathbb{S} \vdash p : S' \longmapsto x_i^{\mathsf{D}}) \quad (\blacklozenge)$$

Then we proceed by three nested inductions, the outer one on $\mathbb{I}$ (or, more strictly, on $|\mathcal{R}(\mathcal{G}) \setminus \mathbb{I}|$, the number of references *not* in $\mathbb{I}$), the second one on $\mathbb{S}$ (more strictly, on $|\mathcal{S}(\mathcal{G}) \setminus \mathbb{S}|$, the number of scopes *not* in $\mathbb{S}$) and the third one on the class C with the order $V > L > P, I, D$. Then we conclude using $\Diamond$ and $\blacklozenge$ and a number of other technical results. Details are in Appendix B of [19]. □

With these lemmas in hand we proceed to prove Theorem 1.

Proof. Fix $\mathbb{I}$, x_i^{R}, and j. Given S, the (unique) scope such that $x_i^{\mathsf{R}} \in \mathcal{R}(S)$:

$$x_j^{\mathsf{D}} \in Res[x_i^{\mathsf{R}}](\mathbb{I}) \Leftrightarrow x_j^{\mathsf{D}} \in Env_V[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](S)$$

By the V case of Lemma 3 and the definition of $\mathbb{P}_S$, this is equivalent to

$$\exists p\ s.t.\ \{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset \vdash p : S \longmapsto x_j^{\mathsf{D}}$$

which, by Lemma 2 and rule X, is equivalent to $\exists p\ s.t.\ \mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$. □

6 α-equivalence and Renaming

The choice of a particular name for a bound identifier should not affect the meaning of a program. This notion of name irrelevance is usually referred to as α-equivalence, but definitions of α-equivalence exist only for some languages and are language-specific. In this section we show how the scope graph and resolution calculus can be used to specify α-equivalence in a language-independent way.

Free variables. A free variable is a reference that does not resolve to any declaration (x_i^{R} is free if $\nexists\, j, p \; s.t. \; \mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$); a bound variable has at least one declaration. For uniformity, we introduce for each possibly free variable x a program-independent artificial declaration $x_{\bar{x}}^{\mathsf{D}}$ with an artificial position $\bar{x}$. These declarations do not belong to any scope but are reachable through a particular well-formed path $\top$, which is less specific than any other path, according to the following rules:

$$\frac{}{\mathbb{I} \vdash \top : S \rightarrowtail x_{\bar{x}}^{\mathsf{D}}} \qquad \frac{p \neq \top}{p < \top}$$

This path representing the resolution of a free reference is shadowed by any existing path leading to a concrete declaration; therefore the resolution of bound variables is unchanged.

6.1 α-Equivalence

We now define α-equivalence using scope graphs. Except for the leaves representing identifiers, two α-equivalent programs must have the same abstract syntax tree. We write `P` $\simeq$ `P'` (pronounced "`P` and `P'` are similar") when the ASTs of `P` and `P'` are equal up to identifiers. To compare two programs we first compare their AST structures; if these are similar then we compare how identifiers behave in these programs. Since two potentially α-equivalent programs are similar, the identifiers occur at the same positions. In order to compare the identifiers' behavior, we define equivalence classes of positions of identifiers in a program: positions in the same equivalence class are declarations of, or references to, the same entity. The abstract position $\bar{x}$ identifies the equivalence class corresponding to the free variable x.

Given a program `P`, we write $\mathbb{P}$ for the set of positions corresponding to references and declarations and $\mathbb{PX}$ for $\mathbb{P}$ extended with the artificial positions (e.g. $\bar{x}$). We define the $\overset{\mathsf{P}}{\sim}$ equivalence relation between elements of $\mathbb{PX}$ as the reflexive symmetric and transitive closure of the resolution relation.

Definition 4 (Position equivalence).

$$\frac{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_{i'}^{\mathsf{D}}}{i \overset{\mathsf{P}}{\sim} i'} \qquad \frac{i' \overset{\mathsf{P}}{\sim} i}{i \overset{\mathsf{P}}{\sim} i'} \qquad \frac{i \overset{\mathsf{P}}{\sim} i' \quad i' \overset{\mathsf{P}}{\sim} i''}{i \overset{\mathsf{P}}{\sim} i''} \qquad \frac{}{i \overset{\mathsf{P}}{\sim} i}$$

In this equivalence relation, the class containing the abstract free variable declaration cannot contain any other declaration. So the references in a particular class are either all free or all bound.

Lemma 4 (Free variable class). *The equivalence class of a free variable does not contain any other declaration, i.e.* $\forall\, x_i^{\mathsf{D}}, i \overset{\mathsf{P}}{\sim} \bar{x} \Longrightarrow i = \bar{x}$

Proof. Detailed proof is in appendix B of [19]. We first prove:
$\forall\, x_i^{\mathsf{R}},\ (\mathbb{I} \vdash \top : x_i^{\mathsf{R}} \longmapsto x_{\bar{x}}^{\mathsf{D}}) \Longrightarrow \forall\, p\ i',\ \mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_{i'}^{\mathsf{D}} \Longrightarrow i' = \bar{x} \wedge p = \top$
and then proceed by induction on the equivalence relation.

The equivalence classes defined by this relation contain references to or declarations of the same entity. Given this relation, we can state that two programs are α-equivalent if the identifiers at identical positions refer to the same entity, that belong to the same equivalence class:

Definition 5 (α-equivalence). *Two programs* P1 *and* P2 *are α-equivalent (denoted* $\texttt{P1} \overset{\alpha}{\approx} \texttt{P2}$*) when they are similar and have the same $\sim$-equivalence classes:*

$$\texttt{P1} \overset{\alpha}{\approx} \texttt{P2} \triangleq \texttt{P1} \simeq \texttt{P2} \wedge \forall\ i\ i',\ i \overset{\texttt{P1}}{\sim} i' \Leftrightarrow i \overset{\texttt{P2}}{\sim} i'$$

Remark 1. $\overset{\alpha}{\approx}$ is an equivalence relation since $\simeq$ and $\Leftrightarrow$ are equivalence relations.

Free variables. The $\overset{\texttt{P}}{\sim}$ equivalence classes corresponding to free variables x also contain the artificial position $\bar{x}$. Since the equivalence classes of two equivalent programs P1 and P2 have to be exactly the same, every element equivalent to $\bar{x}$ (i.e. a free reference) in P1 is also equivalent to $\bar{x}$ in P2. Therefore the free references of α-equivalent programs have to be identical.

Duplicate declarations. The definition allows us to also capture α-equivalence of programs with duplicate declarations. Assume that a reference $x^{\mathsf{R}}_{i_1}$ resolves to two definitions $x^{\mathsf{D}}_{i_2}$ and $x^{\mathsf{D}}_{i_3}$; then i_1, i_2 and i_3 belong to the same equivalence class. Thus all α-equivalent programs will have the same ambiguities.

6.2 Renaming

Renaming is the substitution of a bound variable by a new variable throughout the program. It has several practical applications such as rename refactoring in an IDE, transformation to a program with unique identifiers, or as an intermediate transformation when implementing capture-avoiding substitution.

A valid renaming should respect α-equivalence classes. To formalize this idea we first define a generic transformation scheme on programs that also depends on the position of the sub-term to rewrite:

Definition 6 (Position dependent rewrite rule). *Given a program* P*, we denote by $(t_i \to t' \mid F)$ the transformation that replaces the occurrences of the sub-term t at positions i by t' if the condition F is true. (T)*P *denotes the application of the transformation T to the program* P*.*

Given this definition we can now define the renaming transformation that replaces the identifier corresponding to an entire equivalence class:

Definition 7 (Renaming). *Given a program* P *and a position i corresponding to a declaration or a reference for the name x, we denote by $[x_i{:=}y]$*P *the program* P' *corresponding to* P *where all the identifiers x at positions $\overset{\texttt{P}}{\sim}$-equivalent to i are replaced by y:*

$$[x_i := y]\texttt{P} \triangleq (x_{i'} \to y \mid i' \overset{\texttt{P}}{\sim} i)\texttt{P}$$

However, not every renaming is acceptable: a renaming might provoke variable captures and completely change the meaning of a program.

Definition 8 (Valid renamings). *Given a program* P*, renaming* $[x_i := y]$ *is valid only if it produces an* α*-equivalent program, i.e.* $[x_i := y]\mathtt{P} \overset{\alpha}{\approx} \mathtt{P}$

Remark 2. This definition prevents the renaming of free variables since α-equivalent programs have exactly the same free variables.

Intuitively, valid renamings are those that do not accidentally "capture" variables. Since the capture of a reference resolution also depends on the seen-import context in which this resolution occurs, a precise characterization of capture in our general setting is complex and we leave it for future work.

7 Related Work

Binding-sensitive Program Representations. There has been a great deal of work on representing program syntax in ways that take explicit note of binding structure, usually with the goal of supporting program transformation or mechanized reasoning tools that respect α-equivalence by construction. Notable techniques include de Bruijn indexing [7], Higher-Order Abstract Syntax (HOAS) [20], locally nameless representations [3], and nominal sets [10]. (Aydemir, et al. [2] give a survey in the context of mechanized reasoning.) However, most of this work has concentrated on simple lexical binding structures, such as single-argument λ-terms. Cheney [4] gives a catalog of more interesting binding patterns and suggests how nominal logic can be used to describe many of them. However, he leaves treatment of module imports as future work.

Binding Specification Languages. The *Ott* system [22] allows definition of syntax, name binding and semantics. This tool generates language definitions for theorem provers along with a notion of α-equivalence and functions such as capture-avoiding substitution that can be proven correct in the chosen proof assistant modulo α-equivalence. Avoiding capture is also the basis of hygienic macros in Scheme. Dybvig [8] gives an algorithmic description of what hygiene means. Herman and Wand [13,12] introduce static binding specifications to formalize a notion of α-equivalence that does not depend on macro expansion. Stansifer and Wand's Romeo system [23] extends these specifications to somewhat more elaborate binding forms, such as sequential **`let`**. *Unbound* [25] is another recent domain specific language for describing bindings that supports moderately complex binding forms. Again, none of these systems treat modules or imports.

Language Engineering. In language engineering approaches, name bindings are often realized using a random-access symbol table such that multiple analysis and transformation stages can reuse the results of a single name resolution pass [1]. Another approach is to represent the result of name resolution by means

of *reference attributes*, direct pointers from the uses of a name to its definition [11]. However these representations are usually built using an implementation of a language-specific resolution algorithm. Erdweg, et al. [9] describe a system for defining capture-free transformations, assuming resolution algorithms are provided for the source and target languages. The approach represents the result of name resolution using 'name graphs' that map uses to definitions (references to declarations in our terminology) and are language independent. This notion of 'name graph' inspired our notion of 'scope graph'. The key difference is that the results of name resolution generated by the resolution calculus are *paths* that extend a use-def pair with the *language-independent evidence* for the resolution.

Semantics Engineering. Semantics engineering approaches to name binding vary from first-order representation with substitution [15], to explicit or implicit environment propagation [21,18,6], to HOAS [5]. Identifier bindings represented with environments are passed along in derivation rules, rediscovering bindings for each operation. This approach is inconvenient for more complex patterns such as mutually recursive definitions.

8 Conclusion and Future Work

We have introduced a generic, language-independent framework for describing name binding in programming languages. Its theoretical basis is the notion of a scope graph, which abstracts away from syntax, together with a calculus for deriving resolution paths in the graph. Scope graphs are expressive enough to describe a wide range of binding patterns found in real languages, in particular those involving modules or classes. We have presented a practical resolution algorithm, which is provably correct with respect to the resolution calculus. We can use the framework to define generic notions of α-equivalence and renaming.

As future work, we plan to explore and extend the theory of scope graphs, in particular to find ways to rule out anomalous examples and to give precise characterizations of variable capture and substitution. On the practical side, we will use our formalism to give a precise semantics to the NaBL DSL, and verify (using proof and/or testing) that the current NaBL implementation conforms to this semantics.

Our broader vision is that of a complete language designer's workbench that includes NaBL as the domain-specific language for name binding specification and also includes languages for type systems and dynamic semantics specifications. In this setting, we also plan to study the interaction of name resolution and types, including issues of dependent types and name disambiguation based on types. Eventually we aim to derive a complete mechanized meta-theory for the languages defined in this workbench and to prove the correspondence between static name binding and name binding in dynamics semantics as outlined in [24].

Acknowledgments. We thank the many people who reacted to our previous work on NaBL by asking "but what is its semantics?"; this paper provides our

answer. 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. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, pp. 3–15. ACM (2008)
3. Charguéraud, A.: The locally nameless representation. Journal of Automated Reasoning 49(3), 363–408 (2012)
4. Cheney, J.: Toward a general theory of names: binding and scope. In: Pollack, R. (ed.) ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized Reasoning About Languages with Variable Binding, MERLIN 2005, Tallinn, Estonia, pp. 33–40. ACM (September 30, 2005)
5. Chlipala, A.J.: A verified compiler for an impure functional language. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, pp. 93–106. ACM (2010)
6. Churchill, M., Mosses, P.D., Torrini, P.: Reusable components of semantic specifications. In: Binder, W., Ernst, E., Peternier, A., Hirschfeld, R. (eds.) 13th International Conference on Modularity, MODULARITY 2014, Lugano, Switzerland, April 22-26, pp. 145–156. ACM (2014)
7. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 34(5), 381–392 (1972)
8. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in scheme. Higher-Order and Symbolic Computation 5(4), 295–326 (1992)
9. Erdweg, S., van der Storm, T., Dai, Y.: Capture-avoiding and hygienic program transformations. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 489–514. Springer, Heidelberg (2014)
10. Gabbay, M., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Asp. Comput. 13(3-5), 341–363 (2002)
11. Hedin, G., Magnusson, E.: Jastadd–an aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
12. Herman, D.: A Theory of Hygienic Macros. PhD thesis, Northeastern University, Boston, Massachusetts (May 2010)
13. Herman, D., Wand, M.: A theory of hygienic macros. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 48–62. Springer, Heidelberg (2008)
14. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, Reno/Tahoe, Nevada, pp. 444–463. ACM (2010)

15. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run your research: on the effectiveness of lightweight mechanization. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, pp. 285–296. ACM (2012)
16. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Declarative name binding and scope rules. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 311–331. Springer, Heidelberg (2013)
17. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system (release 4.00): Documentation and user's manual. Institut National de Recherche en Informatique et en Automatique (July 2012)
18. Mosses, P.D.: Modular structural operational semantics. Journal of Logic and Algebraic Programming 61-61, 195–228 (2004)
19. Neron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A theory of name resolution with extended coverage and proofs. Technical Report TUD-SERG-2015-001, Software Engineering Research Group. Delft University of Technology, Extended version of this paper (January 2015)
20. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Wexelblat, R.L. (ed.) Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, pp. 199–208. ACM (1988)
21. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
22. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. Journal of Functional Programming 20(1), 71–122 (2010)
23. Stansifer, P., Wand, M.: Romeo: A system for more flexible binding-safe programming. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, Gothenburg, Sweden, September 1-3, pp. 53–65. ACM (2014)
24. Visser, E., Wachsmuth, G., Tolmach, A.P., Neron, P., Vergu, V.A., Passalaqua, A., Konat, G.D.P.: A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In: Black, A.P., Krishnamurthi, S., Bruegge, B., Ruskiewicz, J.N. (eds.) Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH 2014, Portland, OR, USA, October 20-24, pp. 95–111. ACM (2014)
25. Weirich, S., Yorgey, B.A., Sheard, T.: Binders unbound. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, pp. 333–345. ACM (2011)