A Theory of Name Resolution with Extended Coverage and Proofs

Pierre Neron, Andrew Tolmach, Eelco Visser, Guido Wachsmuth

Report TUD-SERG-2015-001
This technical report is an extended version of the paper:


The technical report adds appendices with further examples of coverage and proofs of the theorems in the papers.

@inproceedings{NeronTVW-ESOP-2015,
  title = {A Theory of Name Resolution},
  author = {Pierre Neron and Andrew P. Tolmach and Eelco Visser and Guido Wachsmuth},
  year = {2015},
  month = {April},
  note = {To appear},
  researcher = {http://researchr.org/publication/NeronTVW-ESOP-2015},
  series = {Lecture Notes in Computer Science},
  publisher = {Springer},
)}

© copyright 2015, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.
A Theory of Name Resolution

Pierre Neron¹, Andrew Tolmach², Eelco Visser¹, and Guido Wachsmuth¹

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

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

1 Introduction

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

In practice, the name resolution rules of real programming languages are usually described using ad hoc and informal mechanisms. Even when a language is formalized, its resolution rules are typically encoded as part of static
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 $\alpha$-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 gram-
mar; 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 [17], 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 [15] 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 $\alpha$-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, 14, 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 appendixes of this paper present 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
- $x_i^D:S$: 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
- $G$: scope graph
- $S(G)$: scopes $S$ in $G$
- $D(S)$: declarations $x_i^D:S'$ in $S$
- $R(S)$: references $x_i^R$ in $S$
- $I(S)$: imports $x_i^R$ in $S$
- $P(S)$: parent scope of $S$

Well-formedness properties
- $P(S)$ is a partial function
- The parent relation is well-founded
- Each $x_i^R$ and $x_i^D$ appears in exactly one scope $S$

Resolution paths

\[
\begin{align*}
  s := D(x_i^D) & | I(x_i^R, x_j^D, S) | P \\
  p := [] & | s | p \cdot p \\
\end{align*}
\]

(inductively generated)

\[
\begin{align*}
  [] \cdot p &= p \cdot [] = p \\
  (p_1 \cdot p_2) \cdot p_3 &= p_1 \cdot (p_2 \cdot p_3)
\end{align*}
\]

Well-formed paths

\[
WF(p) \Leftrightarrow p \in P^\ast \cdot I(\_, \_)^\ast
\]

Specificity ordering on paths

\[
\begin{align*}
  D(\_) &< I(\_, \_) \quad (D1) \\
  I(\_, \_) &< P \quad (IP) \\
  D(\_) &< P \quad (DP) \\
  s_1 < s_2 & \quad (Lex1) \\
  p_1 < p_2 & \quad (Lex2)
\end{align*}
\]

Edges in scope graph

\[
\begin{align*}
  P(S_1) &= S_2 \\
  \models P : S_1 \rightarrow S_2 \\
  y_i^R \in I(S_1) & \models I(p : y_i^R \rightarrow y_j^D, S_2) \\
  \models I(y_i^R, y_j^D) : S_1 \rightarrow S_2
\end{align*}
\]

Transitive closure

\[
\begin{align*}
  \models [] : A \rightarrow A \\
  \models s : A \rightarrow B & \models p : B \rightarrow C \\
  \models s \cdot p : A \rightarrow C
\end{align*}
\]

Reachable declarations

\[
\begin{align*}
  x_i^D &\in D(S') \models p : S \rightarrow S' & WF(p) \\
  \models p : D(x_i^D) : S \rightarrow x_i^D
\end{align*}
\]

Visible declarations

\[
\begin{align*}
  \models p : S \rightarrow x_i^D & \forall j, p' \models I(p' : S \rightarrow x_j^D \Rightarrow p') \\
  \models p : S \rightarrow x_i^D
\end{align*}
\]

Reference resolution

\[
\begin{align*}
  x_i^R \in R(S) & \{ x_i^R \} \cup I(p : S \rightarrow x_j^D) \\
  \models p : x_i^R \rightarrow x_j^D
\end{align*}
\]

Fig. 1. Scope graphs

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

Fig. 3. Resolution calculus
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 \textsf{fun} and \textsf{fix} represent lambda and mu terms, respectively; both have basic unary lexically scoped bindings.
- Let: The various flavors of let bindings (sequential \textsf{let}, \textsf{letrec}, and \textsf{letpar}) challenge the unary lexical binding model.
- Definition: A definition (\textsf{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 \textsf{module} to divide a program into separate units and a notion of \textsf{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 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 $x^R_i$ for a reference with name $x$ at position $i$ and $x^D_i$ 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 $G$, whose nodes are a finite set of scopes $S(G)$. Every program has at least one scope, the global or root scope. Each scope $S$ has an associated finite set $D(S)$ of declarations and finite set $R(S)$ 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 $x^R_i$ in a scope resolves to a declaration of the same variable $x^D_j$ 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 $R(G)$ and $D(G)$ for the (finite) sets of all references and all declarations, respectively, in the program with scope graph $G$. Name resolution is specified by a relation $\rightarrow \subseteq R(G) \times D(G)$ between references and corresponding declarations in $G$. In the absence of edges, this relation is very simple:

$$\frac{x^R_i \in R(S) \quad x^D_j \in D(S)}{x^R_i \rightarrow x^D_j} \quad (X_0)$$

That is, a reference $x^R_i$ resolves to a declaration $x^D_j$, if the scope $S$ in which $x^R_i$ is contained also contains $x^D_j$. We say that there is a resolution path from $x^R_i$ to $x^D_j$. 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 a...
Fig. 5. Declarations and references in global scope.

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 \( b_6 \) reference in Fig. 5 resolves to each of the two declarations \( b_2 \) and \( b_5 \). 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 [18]) 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 \( P(\_\_) \) maps a scope \( S \) to its parent scope \( P(S) \). 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 \texttt{fix} \( f_2 \), and the scope for \texttt{fun} \( n_3 \). The edges from scopes to scopes correspond to the parent.
def \( f_1 = \) 
  fix \( f_2 \) { 
    fun \( n_3 \) { 
      ifz \( n_4 \) then 1 
      else \( n_5 \cdot f_6(n_7 - 1) \) 
    } 
  } 
def \( n_8 = f_9 \)

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 \( \varepsilon_6 \) both declarations \( \varepsilon_1 \) and \( \varepsilon_2 \) are reachable, but from reference \( \varepsilon_9 \) only declaration \( \varepsilon_1 \) 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 \( \varepsilon_2 \) is visible from the scope of \( \varepsilon_6 \).

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

\[
\frac{x_i^R \in R(S_1) \quad p : S_1 \rightarrow S_2 \quad x_j^D \in D(S_2)}{p : x_i^R \mapsto x_j^D} (X_1)
\]

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 \( S_1 \rightarrow S_2 \). Reachability is defined in terms of the parent relation as follows:

\[
\frac{P(S_1) = S_2}{P : S_1 \rightarrow S_2} \quad \frac{p : A \rightarrow A}{s : A \rightarrow B \quad p : B \rightarrow C \quad s \cdot p : A \rightarrow C}
\]

The parent relation between scopes gives rise to a direct edge \( S_1 \rightarrow S_2 \) between child and parent scope, and \( A \rightarrow B \) 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 \( \varepsilon_6 \) can be resolved with path \( P \) to declaration \( \varepsilon_2 \) and with path \( P \cdot P \) to \( \varepsilon_1 \).

**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*:

\[
\frac{x_i^D \in D(S') \quad p : S \rightarrow S'}{p \cdot D(x_i^D) : S \mapsto x_i^D} (R_2)
\]
That is, a declaration $x^D_i$ in $S'$ is reachable from scope $S$ ($S \rightarrow x^D_i$), if scope $S'$ is reachable from $S$.

Given multiple reachable declarations, which one should we prefer? A reachable declaration $x^D_i$ is visible in scope $S$ ($S \rightarrow x^D_i$) if there is no other declaration for the same name that is reachable through a more specific path:

$$\forall j, p'(S \rightarrow x^D_j) \implies \neg (p' < p) \quad (V_2)$$

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

$$D(\_ < P \quad s_1 < s_2 \quad p_1 < p_2 \quad s_1 \cdot p_1 < s_2 \cdot p_2$$

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.

$$x^R_i \in \mathcal{R}(S) \quad p : S \rightarrow x^D_j \quad p : x^R_i \rightarrow x^D_j \quad (X_2)$$

Example. In Fig. 6 the scope (labeled 3) containing reference $\varepsilon_6$ can reach two declarations for $\varepsilon$: $P \cdot D(\varepsilon^D_2) : S_3 \rightarrow \varepsilon^D_2$ and $P \cdot P \cdot D(\varepsilon^D_0) : S_3 \rightarrow \varepsilon^D_1$. Since the first path is more specific than the second path, only $\varepsilon_2$ is visible, i.e. $P \cdot D(\varepsilon^D_2) : S_3 \rightarrow \varepsilon^D_2$. Therefore $\varepsilon_6$ resolves to $\varepsilon_2$, i.e. $P \cdot D(\varepsilon^D_2) : \varepsilon^R_6 \rightarrow \varepsilon^D_2$.

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 $D(S)$ is reachable from that reference, or no declaration in $D(S)$ is reachable from that reference.
- Every declaration in $D(S)$ 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 $x^R_i$ in the set of imports $I(S)$ of a scope $S$. (Imports are also always references and included in some $\mathcal{R}(S')$, but not
necessarily in the same scope in which they are imports.) We model a *module* by associating a scope \( S \) with a declaration \( x_i^D;S \). 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 \( S_1 \rightarrow S_2 \) between a scope \( S_1 \) and its parent scope \( S_2 \) in the scope graph. Similarly, an import induces an edge \( S_1 \rightarrow S_2 \) between a scope \( S_1 \) and the scope \( S_2 \) associated with a declaration imported into \( S_1 \):

\[
\text{def } c_2 = 4 \\
\text{module } A_2 \{ \\
\text{import } B_3 \\
\text{def } a_4 = b_5 + c_6 \\
\} \\
\text{module } B_7 \{ \\
\text{import } C_8 \\
\text{def } b_9 = 0 \\
\} \\
\text{module } C_10 \{ \\
\text{def } b_{11} = 1 \\
\text{def } c_{12} = b_{13} \\
\}
\]

\[
\begin{align*}
&y_i^R \in I(S_1), \quad p : y_i^R \mapsto y_j^D;S_2 \\
&I(y_i^R, y_j^D;S_2) : S_1 \rightarrow S_2 \quad (I_3)
\end{align*}
\]

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 \( \lambda_2 \).) 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

\[
\text{def } c_1 = ... \\
\]
imports the contents of the scope associated to the resolution of reference $B_3$; note that since $B_3$ 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 $B_3$ resolves to declaration $B_7$, which has associated scope 3.) The paths at the bottom right of the figure illustrate that the scope (labeled 2) containing reference $c_6$ can reach two declarations for $c_1: P \cdot D(c_1^D): S_2 \rightarrow c_1^D$ and $I(c_3^R, B_1^P; S_3) \cdot I(c_3^R, c_1^D) \cdot D(c_1^D): S_2 \rightarrow c_{12}^D$, making use of the subsidiary resolutions $c_5^R \rightarrow B_1^P$ and $c_8 \rightarrow c_{10}^D$.

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 $B_{10}$ we can reach declaration $B_7$ with path $D(B_7^D)$ and declaration $B_4$ with path $I(A_6^R, A_2^P; S_A) \cdot D(B_7^D)$ (where $S_A$ is the scope named by declaration $A_2$). We resolve this conflict by extending the specificity order with the rule $D(\_ \_ ) < I(\_ \_ )$. That is, local declarations override imported declarations. Similarly, in the scope of reference $A_8$ we can reach declaration $a_3$ with path $P \cdot D(a_1^D)$ and declaration $a_3$ with path $I(A_6^R, A_2^P; S_A) \cdot D(a_3^P)$. We resolve this conflict by extending the specificity order with the rule $I(\_ \_ ) < P$. That is, resolution through imports is preferred over resolution through parents. In other words, declarations in imported modules override declarations in lexical parents.

The next issue is illustrated in Fig. 9. In the scope of reference $A_8$ we can reach declaration $a_4$ with path $P \cdot D(a_4^P)$ and declaration $a_1$ with path $P \cdot D(a_1^P)$. 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 $a_1$, via the path $I(a_6^R, B_1^P; S_B) \cdot P \cdot D(a_1^P)$. 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 $P^* \cdot I(\_ \_ )^*$, 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:

\[
\frac{x^D \in D(S') \quad p : S \rightarrow S' \quad WF(p)}{p \cdot D(x^D) : S \rightarrow x^D_i} \quad (R_3)
\]
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 \( a_3 \) reachable in the scope of reference \( a_6 \)? This reduces to the question whether the import of \( A_4 \) can resolve to module \( A_2 \). 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 \( A_4^R \rightarrow A_2^D : S_{A_2} \). This conclusion is obtained by using the import at \( A_4 \) to conclude at step (*) that \( S_{root} \rightarrow S_{A_1} \), i.e. that the body of module \( A_1 \) is reachable! In other words, the import of \( A_4 \) is used in its own resolution. Intuitively, this is nonsensical.

To rule out this kind of behavior we extend the calculus to keep track of the set of seen imports \( I \) using judgements of the form \( I \vdash p : x^R \rightarrow x^D_j \). We need to extend all rules to pass the set \( I \), but only the rules for resolution and import are truly affected:

\[
\frac{x^R_i \in R(S) \quad \{ x^R_j \} \cup I \vdash p : S \rightarrow x^D_j}{I \vdash p : x^R_i \rightarrow x^D_j} \quad (X)
\]

\[
\frac{y^R_i \in I(S_1) \setminus I \quad I \vdash p : y^R_i \rightarrow y^D_j : S_2 \quad I \vdash I(y^R_i, y^D_j : S_2) : S_1 \rightarrow S_2}{I \vdash I(y^R_i, y^D_j : S_2)} \quad (I)
\]

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 \( I \) of seen imports (via the \( X \) rule); since the set \( R(G) \) is finite, \( I \) 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

![Fig. 10. Derivation for \( A_4^R \rightarrow A_2^D : S_{A_2} \) in a calculus without import tracking.](image-url)
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 \( \rightarrow, \twoheadrightarrow, \twoheadrightarrow \) and \( \mapsto \rightarrow \) 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

\[
WF(p) \iff p \in P^* \cdot I(\_, \_)?
\]

where \( ? \) denotes the \textit{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:

\[
WF(p) \iff p \in P^* \cdot I(\_, \_)^* \cdot I(x_R, \_), x_{TR}^* \cdot I(x_{TR}, \_)?
\]

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

Similarly, we can modify the rule to handle the \textit{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:

\[
WF(p) \iff p \in P^* \cdot I(x_R, \_)^* \cdot I(x_{ER}, \_)?
\]

**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 \texttt{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

\begin{verbatim}
module A1 {
  def x2 = 3 
}
module B3 {
  include A1;
  def x5 = 6;
  def x6 = x7
}
\end{verbatim}

Fig. 13. Include
former should not hide the latter. To handle this situation, we can drop the rule $D(\_ \_ \_I(\_\_\_\_\_))$ 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 [15,17], 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 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. 14. Example LM programs with sequential, recursive, and parallel let, and their encodings as scope graphs.](image-url)
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 \( (f_5) \) is a reference in the anonymous scope. We add the qualifying scope name \( (D_4) \) 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, \( g_{10} \) correctly resolves to \( g_5 \). Since local declarations hide imported declarations, \( f_{12} \) also refers correctly to the local declaration \( f_9 \), which hides the transitively
Fig. 17. Scope graph construction for LM via syntax-directed AST traversal.

imported $\xi_2$. 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 [17], 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 \([\text{pattern}]_{\text{args}} \). When \( f \) is
invoked on a term, the clause whose \text{pattern} matches the term is executed. Func-
tions may also take additional arguments \text{args}. Each clause body consists of a
sequence of statements separated by semicolons. Functions can optionally return
a value using \text{ret}(). The \text{let} statement binds a metavariable in the remainder of
the clause body. An empty clause body is written ()

The algorithm is initiated by invoking \( [\_]^{\text{prog}} \) on an entire LM program. Its
net effect is to produce a scope graph via a sequence of imperative operations.
The construct \text{new}\( P \) creates a new scope \( S \) with parent \( P \) (or no parent if \( p = \bot \))
and empty sets \( D(S) \), \( R(S) \), and \( I(S) \). 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 \text{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 \text{environment} is just a set of declarations \( x^D \). 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 decla-
rations resulting from the parent and import relations. The key operator for
combining environments is \text{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 envi-
ronment. More formally:

\text{Definition 1 (Shadowing).} For any environments \( E_1, E_2 \), we write:
\[ E_1 \triangleleft E_2 := E_1 \cup \{ x^D_i \in E_2 \mid \nexists x^D_i \in E_1 \} \]
\[\begin{align*}
Res[I](x^R_i) & := \{x^D_j \mid \exists S \text{ s.t. } x^R_i \in R(S) \land x^D_j \in EnvV\{\{x^R_i\} \cup I, \emptyset\}(S)\} \\
EnvV[I,S](S) & := EnvL[I,S](S) \triangleleft EnvP[I,S](S) \\
EnvL[I,S](S) & := EnvD[I,S](S) \triangleleft EnvL[I,S](S) \\
EnvD[I,S](S) & := \emptyset \text{ if } S \in S \\
EnvP[I,S](S) & := \emptyset \text{ if } S \in S \\
EnvP[I,S](S) & := \{\emptyset \text{ if } S \in S, EnvL[I,S](S) \cup S(S) \mid y^R_i \in I(S) \land I \land y^D_j \in Res[I,y^R_i]\} \\
EnvP[I,S](S) & := \{\emptyset \text{ if } S \in S, EnvV[I,S](S) \cup S(S) \mid y^D_j \in EnvP[I,S](S)\}
\end{align*}\]

**Fig. 18.** Resolution algorithm

Figure 18 specifies an algorithm \(Res[I](x^R_i)\) for resolving a reference \(x^R_i\) to a set of corresponding declarations \(x^D_j\). Like the calculus, the algorithm avoids trying to use an import to resolve itself by maintaining a set \(I\) of “already seen” imports. The algorithm works by computing the full environment \(EnvV[I,S](S)\) of declarations that are visible in the scope \(S\) containing \(x^R_i\), and then extracting just the declarations for \(x\). The full environment, in turn, is built from the more basic environments \(EnvD\) of immediate declarations, \(EnvL\) of imported declarations, and \(EnvP\) 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 \((|\mathcal{R}(G) \setminus I|, |\mathcal{S}(G) \setminus S|)\). Termination is straightforward by unfolding the calls to \(Res\) in \(EnvL\) and then inlining the definitions of \(EnvV\) and \(EnvL\); 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.** \(\forall I, x^R_i, j, (x^D_j \in Res[I](x^R_i)) \iff (\exists p \text{ s.t. } I \vdash p : x^R_i \rightarrow x^D_j).\)

We sketch the proof of this theorem here; details of the supporting lemmas and proofs are in Appendix B. To begin with, we must deal with the 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 \( \Gamma \vdash p : x_i^R \rightarrow x_j^D \) 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 \( S \) 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.** \( \forall \Gamma, S, x_i^P, (\exists p \text{ s.t. } \Gamma, S \vdash p : S \rightarrow x_i^P) \iff (\exists p \text{ s.t. } \Gamma, S \vdash p : S \rightarrow x_i^P) \).

Hereinafter, we can work with the primed system.

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

**Definition 2 (Path Sets).**

\[
\begin{align*}
\mathcal{P}_D[\Gamma, S](S) & := \{ p \mid \exists x_i^P \text{ s.t. } p = D(x_i^P) \land \Gamma, S \vdash p : S \rightarrow x_i^P \} \\
\mathcal{P}_P[\Gamma, S](S) & := \{ p \mid \exists p' \text{ s.t. } p = P \cdot p' \land \Gamma, S \vdash p : S \rightarrow x_i^P \land \Gamma, S \vdash p' : P(S) \rightarrow x_i^D \} \\
\mathcal{P}_I[\Gamma, S](S) & := \{ p \mid \exists p' \text{ s.t. } p = I(g_i^R, g_i^D, S') \land \Gamma, S \vdash p : S \rightarrow x_i^P \land \Gamma, S \vdash p' : P(S') \rightarrow x_i^D \} \\
\mathcal{P}_L[\Gamma, S](S) & := \{ p \mid \exists x_i^D \text{ s.t. } \Gamma, S \vdash p : S \rightarrow x_i^D \land p \in I(\_ , \_ )^* \cdot D(\_ ) \} \\
\mathcal{P}_V[\Gamma, S](S) & := \{ p \mid \exists x_i^D \text{ s.t. } \Gamma, S \vdash p : S \rightarrow x_i^D \}
\end{align*}
\]
These sets are designed to correspond to the various classes of environments $Env_C$, $P_D$, $P_P$, and $P_I$ contain all reachability derivations starting with a $D(\_\_)$, $P$, or $I(\_\_)$ respectively, with the further condition that the tail of each derivation is a visibility derivation (i.e., is most specific among all reachability derivations). $P_V$ describes the set of all visibility derivations. ($P_L$ is similar, but omits paths including $P$ 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 $p$, $\delta(p) := x_1^D$ iff $\exists p' \ s.t. \ p = p' \cdot D(x_1^D)$ and for any set of paths $P$, $\Delta(P) := \{ \delta(p) \mid p \in P \}$.

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

$$\forall \ I \ S, Env[I, S](S) = \Delta(P_C[I, S](S))$$

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

$$\forall \ I \ S \ s \ p \ S \ x_1^D, (I, S) \vdash s \cdot p : D(x_1^D) : S \rightarrow x_1^D \implies I, \{ S \} \cup S \vdash s : S \rightarrow S' \implies I, \{ S \} \cup S \vdash p : S' \rightarrow x_1^D \ (\Diamond)$$

$$\forall \ I \ S \ s \ p \ S \ x_1^D, (I, S) \vdash s \cdot p : S \rightarrow x_1^D \implies I, \{ S \} \cup S \vdash p : S' \rightarrow x_1^D \ (\heartsuit)$$

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

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

**Proof.** Fix $I$, $x_i^R$, and $j$. Given $S$, the (unique) scope such that $x_i^R \in R(S)$:

$$x_1^D \in Res[x_i^R](I) \iff x_1^D \in Env[I, \{ x_i^R \} \cup I, \emptyset](S)$$

By the $V$ case of Lemma 3 and the definition of $P_S$, this is equivalent to

$$\exists p \ s.t. \ \{ x_i^R \} \cup I, \emptyset \vdash p : S \rightarrow x_1^D$$

which, by Lemma 2 and rule $X$, is equivalent to $\exists p \ s.t. \ I \vdash p : x_i^R \rightarrow x_j^D$. \qed

### 6 $\alpha$-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 $\alpha$-equivalence, but definitions of $\alpha$-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 $\alpha$-equivalence in a language-independent way.
Free variables. A free variable is a reference that does not resolve to any declaration ($x^R_i$ is free if $\not\exists j, p$ s.t. $I \vdash p : x^R_i \rightarrow x^D_j$); a bound variable has at least one declaration. For uniformity, we introduce for each possibly free variable $x$ a program-independent artificial declaration $x^D_{\bar{x}}$ 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:

$$
I \vdash \top : S \rightarrow x^D_{\bar{x}} \quad p \neq \top \\
p < \top
$$

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 $\alpha$-Equivalence

We now define $\alpha$-equivalence using scope graphs. Except for the leaves representing identifiers, two $\alpha$-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 $\alpha$-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 $\mathcal{P}$ for the set of positions corresponding to references and declarations and $\mathcal{P}X$ for $\mathcal{P}$ extended with the artificial positions (e.g. $\bar{x}$). We define the $\sim$ equivalence relation between elements of $\mathcal{P}X$ as the reflexive symmetric and transitive closure of the resolution relation.

Definition 4 (Position equivalence).

$$
\frac{\vdash p : x^R_i \rightarrow x^D_{i'}}{i \sim i'} \\
\frac{i' \sim i''}{i' \sim i''} \quad \frac{i \sim i'}{i \sim i'}
$$

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. $\forall x^D_i, i \sim \bar{x} \implies i = \bar{x}$

Proof. Detailed proof is in Appendix B. We first prove:

$\forall x^R_i, (I \vdash \top : x^R_i \rightarrow x^D_{i'}) \implies \forall p i', I \vdash p : x^R_i \rightarrow x^D_{i'} \implies i' = \bar{x} \land p = \top$

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 $\alpha$-equivalent if the identifiers at identical positions refer to the same entity, that belong to the same equivalence class:

**Definition 5 ($\alpha$-equivalence).** Two programs $P_1$ and $P_2$ are $\alpha$-equivalent (denoted $P_1 \approx P_2$) when they are similar and have the same $\sim$-equivalence classes:

$$P_1 \approx \alpha P_2 \iff P_1 \simeq P_2 \land \forall i, i' : i \sim i' \iff i' \sim \alpha i$$

**Remark 1.** $\approx$ is an equivalence relation since $\simeq$ and $\iff$ are equivalence relations.

**Free variables.** The $\sim$ equivalence classes corresponding to free variables $x$ also contain the artificial position $\bar{x}$. Since the equivalence classes of two equivalent programs $P_1$ and $P_2$ have to be exactly the same, every element equivalent to $\bar{x}$ (i.e. a free reference) in $P_1$ is also equivalent to $\bar{x}$ in $P_2$. Therefore the free references of $\alpha$-equivalent programs have to be identical.

**Duplicate declarations.** The definition allows us to also capture $\alpha$-equivalence of programs with duplicate declarations. Assume that a reference $x_{i_1}$ resolves to two definitions $x_{i_2}^D$ and $x_{i_3}^D$; then $i_1$, $i_2$ and $i_3$ belong to the same equivalence class. Thus all $\alpha$-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 $\alpha$-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 $(t_i \rightarrow t' \mid F)$ 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 $\bar{x}$-equivalent to $i$ are replaced by $y$:

$$[x_i := y]P \triangleq (x_{i'} \rightarrow y \mid i' \bar{x} i)P$$
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 $\alpha$-equivalent program, i.e. $[x_i := y]P \approx P$.

**Remark 2.** This definition prevents the renaming of free variables since $\alpha$-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 $\alpha$-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 $\lambda$-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 $\alpha$-equivalence and functions such as capture-avoiding substitution that can be proven correct in the chosen proof assistant modulo $\alpha$-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 [14,13] introduce static binding specifications to formalize a notion of $\alpha$-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.
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 [16], to explicit or implicit environment propagation [21,19,6], 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 $\alpha$-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


A More Name Binding Patterns

In this appendix we discuss more name binding patterns and their encoding using scope graphs.

A.1 Definition-before-use

Our scopes are sets of mutually recursive declarations. This makes it easy to model a construct such as letrec as we have seen in Section 3. However, the scope constructs in many languages have a ‘definition before use’ policy in which bindings are only visible to references appearing ‘later’ in the program. For example, the variable declarations in a block in C or Java are only visible to the subsequent statements in the block. While colloquially such a block is thought of as a single scope, in fact each variable declaration opens a new scope, much like the bindings in the sequential let construct. Fig. 20 shows how the policy is encoded for blocks in Java by placing each declaration in a new scope that dominates the references in subsequent statements.

A.2 Compilation Units and Packages in Java

Scopes do not always correspond to single, continuous program fragments. For example, in Java [11], the scope of a top-level type name is all type declarations in the package in which it is declared. Those type declarations can be spread over multiple, discontinuous program fragments, since a package can consist of a number of compilation units.

A compilation unit can be modeled as a scope, which imports other scopes associated with the same package name. For example, Fig. 21 shows two compilation units contributing to the same package p. Each compilation unit declares package p in the root scope and associates the declaration with a scope for top-level type declarations. Types c and d are declared in separate scopes, but are reachable in both scopes via imports from scopes associated with p. This approach follows the Java terminology, where each compilation unit declares its package.

Import Declarations. Fig. 22 shows a compilation unit with two import declarations. Java supports two kinds of type-import declarations in compilation units. A single-type-import declaration imports a single type. A type-import-on-demand declaration allows all accessible types of a package or type to be imported as needed. Thereby, types imported by single-import declarations shadow types of
the same name imported by import-on-demand declarations, as well as top-level
types of the same name declared in other compilation units of the same package.
The scope of an imported type name is all type declarations in the compilation
unit in which the import declaration appears.

We model visibility of imported declarations by three lexical scopes per com-
pilation unit. The first scope handles all import-on-demand declarations. The
second scope imports type declarations from other compilation units of the same
package, and declares aliases for single-type-import declarations. Those aliases
act as a declaration and as a reference to the imported name. According to
the default visibility policy, the alias declarations shadow imported declarations
from the second and first scopes. The third scope is associated with the package
name and declares the types of the compilation unit. Those local declarations
shadow declarations from other compilation units or packages.

Example. The scope graph diagram in Fig. 22 shows a root scope for package
names and three lexical scopes for the compilation unit. Scope 2 imports from
a package \( r \) and scope 3 imports from other compilation units from package \( p \).
The single-import declaration \( q.E \) is modeled as a qualified name resolution in
the global scope and as a declaration in scope 3. Scope 4 declares the top-level
types \( C \) and \( D \).

Accessibility. In Java, the reachability of type declarations can be influenced by
access modifiers. In the example, \( C \) and \( D \) can be accessed in other compilation
units of \( p \), but \( D \) cannot be accessed in other packages. To model accessibility,
we need to distinguish public from package-private classes, and imports of
compilation units of the same package from regular imports in the scope graph.
We can then define a Java-specific well-formedness predicate which rejects paths
with regular imports of package-private classes.

A.3 Namespaces and Partial Classes in C#

While Java packages can be declared in multiple compilation units, C# names-
paces are open-ended, allowing also for multiple declarations of a namespace in
the same compilation unit. In a similar way, C# supports open-ended class declarations. Fig. 23 shows two declarations of namespace N, both partially declaring a class C. The left part declares a field f, while the right part is referring to this field. Similar to Java import declarations, the using directive in the left part only affects the left namespace declaration.

We can model those partial scopes similarly to packages in Java. Each namespace declaration corresponds to two lexical scopes, where local declarations in the second scope hide imported declarations in the first scope. The second scope also imports from other namespace declarations for the same namespace. The same pattern can be applied for partial class declarations. However, C# supports proper nesting of namespaces and partially qualified names. Thus, we can no longer resolve to other partial declarations of the same name in global scope. With the default reachability policy, we might find other partial declarations for the same name in lexical scope. Instead, we need to make sure to resolve only to parts in the same surrounding namespace (which might again be declared in several parts). This can be achieved by distinguishing imports of parts from regular imports, and a reachability policy for the former, which rejects parent paths.
B Proofs
We often write just $x^R$ (resp. $x^D$) for a reference (resp. declaration) at some particular but unnamed position.

B.1 Termination of resolution algorithm
Figure 24 presents the inlined resolution algorithm that only use strictly decreasing recursive calls.

\[
\begin{align*}
Res[\emptyset](x^R) & := \{ x^D \mid \exists S \text{ s.t. } x^R \in R(S) \land x^D \in \{ EnvD[{x^R} \cup I, \emptyset](S) \} \\
Envp[I, S](S) & := \{ 0 \text{ if } S \in S \} \\
Envp[I, S](S) & := \bigcup \left\{ \begin{array}{l}
EnvD[I, \{ S \} \cup \emptyset](S_y) \\
\cup EnvP[I, \{ S \} \cup \emptyset](S_y)
\end{array} \right\} \\
Envp[I, S](S) & := \{ 0 \text{ if } S \in S \} \\
Envp[I, S](S) & := \{ EnvD[I, \{ S \} \cup \emptyset](P(S)) \} \\
Envp[I, S](S) & := \{ EnvP[I, \{ S \} \cup \emptyset](P(S)) \}
\end{align*}
\]

Fig. 24. Inlined resolution algorithm

B.2 Well-formed Paths and Specificity Ordering
We can give a simple grammatical characterization of well-formed paths, and use it to define a measure on paths that is compatible with the specificity ordering.

Lemma 5 (Well-formed path measure). For all well-formed paths $p$, there exist $i, j \geq 0$ such that $p = P^i \cdot I(\_, \_)^j$. We write $m(p)$ for the corresponding pair $(i, j)$ and write $\prec$ for the lexicographic ordering on these pairs.

Lemma 6 (WF path measure and specificity ordering are compatible). For all well-formed paths $p, p'$,

\[ p \cdot D(\_) < p' \cdot D(\_) \iff m(p) < m(p') \]

Proof. Let $m(p) = (i, j)$ and $m(p') = (i', j')$. Let $q = p \cdot D(\_)$ and $q' = p' \cdot D(\_),$ and write $q_k, q'_k$ for the $k$-th steps of $q$ and $q'$, respectively.

$(\Rightarrow)$ Assume $p < p'$. Then in the first step $k$ where $q$ and $q'$ differ (i.e., the first application of rule $Lex1$), we have three possible cases:
(\rightarrow \leftarrow) Suppose m(p) < m(p'). Then there are two cases:

- i < i': then \forall k \leq i, q_k = q'_k, q_{i+1} = I(_, _) or D(_, _) and q'_{i+1} = P; hence p < p'.
- i = i' and j < j': then \forall k \leq i + j, q_k = q'_k, q_{i+j+1} = I(_, _); hence p < p'.

\qed

B.3 Cycles in Resolution Paths

To enforce the termination of the resolution algorithm, we only look for cycle-free paths in this algorithm. Looking for cycle-free paths is sufficient:

Definition 9 (Cycle). A path p contains a cycle if there exist p_1, p_2, p_3 such that p = p_1 \cdot p_2 \cdot p_3, p_2 \neq [], and there exist A, B and S such that \mathopen| p_1 : A \rightarrow S, \mathopen| p_2 : S \rightarrow S, and \mathopen| p_3 : S \rightarrow B.

Lemma 7 (Lemma 1: Resolution paths are cycle-free.). If

\mathopen| p : x^R \rightarrow x^D

then p is cycle-free.

Proof. By rule (X), we have \mathopen| p : S_r \rightarrow x^D where S = \{x^R\} \cup \mathopen| S and S_r is the (unique) scope such that x^R \in \mathcal{R}(S_r). By rule (V) we have \mathopen| p : S_r \rightarrow x^D and

\mathopen| p : x^R \rightarrow S_r \rightarrow x^D and p' < p. (\star)

By rule (R) we have \mathopen| p_0 : S_r \rightarrow S_d where p = p_0 \cdot D(x^D) and S_d is the (unique) scope such that x^D \in \mathcal{D}(S_d); moreover, WF(p_0) holds, so by Lemma 5, p_0 = P' \cdot I(_, _) for some i, j \geq 0. We claim p_0 contains no cycle. For suppose it does, with p_1, p_2, p_3 and S as in Definition 9, and consider the shrunken path p' = p_1 \cdot p_3. Then, since \mathopen| p_1 : S_r \rightarrow S and \mathopen| p_3 : S \rightarrow S_d, we have \mathopen| p : S_r \rightarrow S_d. Also, p' must have the form P' \cdot I(_, _)', where, depending on the position of p_2 in p_0, either (i) i' < i or (ii) i' = i and j' < j. Therefore WF(p') holds, and we have \mathopen| p' : D(x^D) : S_r \rightarrow x^D. But since m(p') = (i', j') < (i, j) = m(p_0), then by Lemma 6, p' \cdot D(x^D) < p_0 \cdot D(x^D), contradicting (\star). So p_0 has no cycle, and hence neither does p. \qed
B.4 Shadowing and Resolution

Lemma 8 (Shadowing preserves resolution). If there is a reachability path from a scope to the declaration of a variable, then there is a visibility path, at least as specific, from that scope to some declaration of that variable.

∀ I, S ⊢ p : S  x_i^S \implies (∃ q, j \text{ s.t. } I, S \vdash q : S \mapsto x_j^S \land q \leq p).

Proof. Fix I, S, p, and x_i^S. Let W := \{p' | \exists i' \text{ s.t. } I, S \vdash p' : S \mapsto x_i^S \land p' \leq p\} be the set of reachability paths from S to x that are at least as specific as p. We need to show that W contains a minimal reachable path, i.e., an element q such that: ∀i' p', I, S \vdash p' : S \mapsto x_i^S \Rightarrow \neg(p' < q). W is not empty (it contains p) and the set of associated measures of its members, i.e.

m(W) = \{m(p') | \exists i' \text{ s.t. } I, S \vdash p' : S \mapsto x_i^S \land p' \cdot D(x_i^S) \leq p\}

is a set of pairs of natural numbers, on which the lexicographic ordering is well-founded. Thus it has at least one smallest element, with associated path q and declaration position j, and using lemma 6, q is minimal for < in W. We claim that q is minimal among all reachable paths. For suppose not; then there is a path q' < q and an i' such that

I, S \vdash q' : S \mapsto x_j^S.

But since q is in W then q \leq p and < is transitive so q' < p and then q' ∈ W, contradicting the minimality of q in W. Therefore we have an element q and position j such that:

I, S \vdash q : S \mapsto x_j^S \land q \leq p

\square

B.5 Correctness of resolution algorithm

We now proceed to prove the correctness of the resolution algorithm. We first address some auxiliary lemmas that we will need in the proof. First, the path uniquely encodes the sequence of scopes in the resolution:

Lemma 9 (Transition unique).

I, S \vdash s : S \mapsto S_1 \implies I, S \vdash s : S \mapsto S_2 \implies S_1 = S_2

Proof. Trivial by case analysis on s: import steps record the imported scope S_y to which they step, and parent steps go to the uniquely defined parent. \square

Then we state that we can cut a resolution path for the reachability relation, i.e., if a declaration is reachable from a scope S then it is also reachable from any scope appearing in the resolution.
Lemma 10 (Reachable tail). If a definition is reachable from a scope $S$ through scope $S'$ then it is also reachable from $S'$:

$$\forall I S s P D, I, \{S\} \cup \{S\} \vdash s : S \rightarrow S' \implies
I, S \vdash s \cdot P (x^D) : S \rightarrow x^D \implies
I, \{S\} \cup \{S\} \vdash P (x^D) : S' \rightarrow x^D$$

Proof. If $s \cdot P$ is well formed then $P$ is also well formed. By inversion of:

$$I, S \vdash s \cdot P (x^D) : S' \rightarrow x^D$$

and using lemma 9, we get:

1) $I, \{S, S\} \cup \{S\} \vdash P : S' \rightarrow S_d$
2) $S \notin S$
3) $S' \notin S$
4) $x^D \in D(S_d)$

Therefore, using 1), 3) and 4), we get by rule $(R')$:

$$I, \{S\} \cup \{S\} \vdash P (x^D) : S' \rightarrow x^D$$

\[\square\]

Similarly, if a declaration is visible from a reference then it is visible all along the path.

Lemma 11 (Visible tail). If a definition is visible from a scope $S$ through scope $S'$ then it is also visible from $S'$:

$$\forall I S p S D, I, \{S\} \cup \{S\} \vdash s : S \rightarrow S' \implies
I, S \vdash s \cdot P (x^D) : S \rightarrow x^D \implies
I, \{S\} \cup \{S\} \vdash P (x^D) : S' \rightarrow x^D$$

Proof. By inversion on $(ii)$ we get:

$$(*) I, S \vdash s \cdot P (x^D) : S \rightarrow x^D$$

$$(**) \forall p' x^D, s.t. I, S \vdash p' : S \rightarrow x^D \implies \neg p' < s \cdot p$$

By lemma 10 and $(*)$ we have:

$(\bigcirc) I, \{S\} \cup \{S\} \vdash P (x^D) : S' \rightarrow x^D$

Suppose $p$ is not minimal. Then there is a $p'$ such that:

$(Hr) I, \{S\} \cup \{S\} \vdash p' (x^D) : S' \rightarrow x^D$

$(<) p' (x^D) < p (x^D)$
If \( s = \mathbf{P} \) then, since inversion on \((\mathbf{Hr})\) gives us \( \text{WF}(p') \), we obtain \( \text{WF}(s \cdot p') \).

Alternatively, if \( s = \mathbf{I}(\_, \_) \) then, since inversion on \((\ast)\) gives us \( \text{WF}(s \cdot p) \), we know \( p \) has the form \( \mathbf{I}(\_, \_)^* \). Then by \((\prec)\), we must have that \( p' \) also has the form \( \mathbf{I}(\_, \_)^* \), so we again obtain \( \text{WF}(s \cdot p') \). Therefore, by inversion of \((\mathbf{Hr})\) and using \((i)\) we have:

\[
\mathbf{I}, \mathbf{S} \vdash s \cdot p' \cdot \mathbf{D}(x_i^D) : S \rightarrow x_i^D
\]

which contradicts \((\ast\ast)\). Therefore \( p \) is minimal and using \((\Diamond)\) we have:

\[
\mathbf{I}, \{S\} \cup \mathbf{S} \vdash p \cdot \mathbf{D}(x_i^D) : S' \rightarrow x_i^D.
\]

\[\Box\]

The elements in the \( \mathbb{P}_C \) sets are incomparable under the specificity ordering.

**Lemma 12 (Path set minima).**

\[
\forall C \quad \mathbf{I} \mathbf{S} \mathbf{S} \quad \forall p, p' \in \mathbb{P}_C[\mathbf{I}, \mathbf{S}](S), \quad \neg p < p'.
\]

**Proof.** Assume for contradiction that \( p < p'(\ast) \). Proceed by case analysis on \( C \):

- Case \( I \): by definition of \( \mathbb{P}_I \):
  - \( i \) \( p = \mathbf{I}(y^R, y^D, S_y) \cdot \bar{p} \)
  - \( ii \) \( p' = \mathbf{I}(z^R, z^D, S_z) \cdot \bar{p}' \)
  - \( iii \) \( \mathbf{I}, \{S\} \cup \mathbf{S} \vdash \bar{p} : S_y \rightarrow x_i^D \)
  - \( iv \) \( \mathbf{I}, \{S\} \cup \mathbf{S} \vdash \bar{p}' : S_z \rightarrow x_i^D \)

By inversion of \((\ast)\), \( \mathbf{I}(y^R, y^D, S_y) = \mathbf{I}(z^R, z^D, S_z) \) and hence \( S_y = S_z \), so we cannot have \( \bar{p} < \bar{p}' \) since both \( \bar{p} \) and \( \bar{p}' \) are minimal.

- Case \( P \): Proof is similar to case \( I \): both \( \bar{p} \) and \( \bar{p}' \) are resolutions from \( \mathbb{P}(S) \).

- Cases \( L, V, D \): trivial. \[\Box\]

We now proceed to prove that the different functions of the algorithm compute definitions corresponding to paths in the \( \mathbb{P}_C \) sets.

**Lemma 13 (Lemma 3). For each class \( C \in \{V, L, D, I, P\} \):**

\[
\forall \quad \mathbf{I} \mathbf{S} \mathbf{S} \quad \mathbf{Env}_C[\mathbf{I}, \mathbf{S}](S) = \Delta(\mathbb{P}_C[\mathbf{I}, \mathbf{S}](S))
\]

**Proof.** The proof is by three nested inductions, the outer one on \( I \) (or, more strictly, on the size of \( |\mathcal{R}(G)\setminus I| \), the references not in \( I \)) and the second one on \( S \) (more strictly, the size of \( |S(G)\setminus S| \), the scopes not in \( S \)) and the third one on the class \( C \), with the order \( V > L > D, I, P \). We then pick an \( x_i^D \), and proceed according to the class \( C \):

**Case D:**

\[
\begin{align*}
x_i^D \in \mathbf{Env}_D[\mathbf{I}, \mathbf{S}](S) & \iff \text{(by definition of } \mathbf{Env}_D) \\
x_i^D \in \mathbf{D}(S) \land S \notin \mathbf{S} & \iff \text{(by rule } (R')) \\
\mathbf{I}, \mathbf{S} \vdash \mathbf{D}(x_i^D) : S \rightarrow x_i^D & \iff \text{(by definition of } \Delta \text{ and } \mathbb{P}_D)
\end{align*}
\]
\[ x^D \in \Delta(P_D[I, S](S)) \]

**Case P:**
\[ x^D \in Env_P[I, S](S) \iff (by \ definition \ of \ Env_P) \]
\[ x^D \in Env_V[I, \{S\} \cup S](P(S)) \land S \notin S \iff (by \ induction \ on \ S) \]
\[ \exists p \ s.t. \ I, \{S\} \cup S \vdash p \cdot D(x^D) : P(S) \longrightarrow x^D \land S \notin S \]
\[ \iff (\Rightarrow \ by \ rule \ (P) \ and \ rules \ (V', R')); \iff \ by \ weakening) \]
\[ \exists p \ s.t. \ I, \{S\} \cup S \vdash p \cdot D(x^D) : P(S) \longrightarrow x^D \land S \notin S \]
\[ \iff (by \ rules \ (R') \ and \ (T')) \]
\[ x^D \in \Delta(P_D[I, S](S)) \]

**Case I:**
\[ x^D \in Env_I[I, S](S) \iff (definition \ of \ Env_I) \]
\[ x^D \in \bigcup \left\{ Env_I[I, \{S\} \cup S](S_y) \mid y^R \in I(S) \setminus I \land y^D : S_y \in Res[I](y^R) \right\} \land S \notin S \]
\[ \iff (by \ unfolding \ of \ Res[I](y^R) \ and \ induction \ on \ I) \]
\[ x^D \in \bigcup \left\{ Env_I[I, \{S\} \cup S](S_y) \right\} \]
\[ \exists y^R \ s.t. \ y^R \in I(S) \setminus I \land \exists S', y^R \in R(S') \land y^D : S_y \in \Delta(P_V[I, \{y^R\} \cup S](S_y)) \]
\[ \iff (by \ definition \ of \ P_V \ and \ rule \ (X')) \]
\[ x^D \in \bigcup \left\{ Env_I[I, \{S\} \cup S](S_y) \right\} \]
\[ \exists y^R \ s.t. \ I \vdash q : y^R \longrightarrow y^D : S_y \]
\[ \iff (by \ L \ case \ of \ induction \ on \ S) \]
\[ \exists y^R \ y^D : S_y \ s.t. \ y^R \in I(S) \setminus I \land \exists S', y^R \in R(S') \land y^D : S_y \]
\[ \iff (by \ definition \ of \ P_L) \]
\[ \exists y^R \ y^D : S_y \ s.t. \ y^R \in I(S) \setminus I \land \exists S', y^R \in R(S') \land y^D : S_y \]
\[ \iff (by \ rule \ (I)) \]
\[ \exists x \exists y \exists D \in S_y \ni \] 

\[ n \in I(\_,\_) \ast \land S \notin S \]

\[ \iff \text{(by (T'), (I) and WF definition)} \]

\[ \exists S \exists y \exists R_1 \ni \]

\[ I(\_,\_) \ast \land S \notin S \]

\[ \iff \text{(by (R'))} \]

\[ \exists y \exists y \exists D \in S_y \ni \]

\[ I(\_,\_) \ast \land S \notin S \]

\[ \iff \text{(by definition of P_I)} \]

\[ \exists y \exists y \exists D \in S_y \ni \]

\[ I(\_,\_) \ast \land S \notin S \]

\[ \iff \text{(by definition of (\_) and \( D \))} \]

\[ x \in \Delta(P_I[S]) \]

\[ \text{Case L: We split the two directions:} \]

\[ \text{Case L. (\( \Rightarrow \))} : \]

\[ x \in \text{Env}_L[S] \implies \text{(by definition of Env}_L[S] \]

\[ x \in \text{Env}_D[S] \land \text{Env}_L[S] \]

By case on the set \( x \) comes from:

1) If \( x \in \text{Env}_D[S] \), then by the induction on the class (since \( D < L \) we get:

\[ x \in \Delta(P_D[S](S)) \implies \text{(by definition of \( \Delta \) and \( P_D \))} \]

\[ S \ni x \in \Delta(P_D[S](S)) \implies \text{(by definition of \( D \) and \( P_D \))} \]

\[ S \ni x \in \Delta(P_D[S](S)) \implies \text{(by definition of \( D \) and \( P_D \))} \]

2) If \( \forall i, x \in \text{Env}_D[S] \) and \( x \in \text{Env}_L[S] \) then by the induction on class (since \( I < L \) we get that: \( x \in \Delta(P_I[S]) \)

By definition of \( P_I \) there exist \( p \) and \( y \) such that \( p = I(y, y \cdot D) \cdot p' \)

and:

\[ (\ast) \ni S \ni x \implies x \in \Delta(P_I[S](S)) \]

\[ (\ast \ast) \ni S \ni x \implies x \in \Delta(P_I[S](S)) \]

From (\( \ast \)) we have \( WF(I(y, y \cdot D) \cdot p') \), which trivially implies that \( p \in I(\_,\_) \ast \cdot D(\_). \)

Thus, to prove that \( x \in \Delta(P_I[S](S)) \) it is sufficient to prove that \( I, S \vdash p : S \implies x \in \Delta(P_I[S](S)) \) and given (\( \ast \)) we need only prove that \( p \) is minimal:

Assume for contradiction that there is a path \( p' \) such that:

\[ (\circ) \ni S \ni \bar{p} : S \implies x \in \Delta(P_I[S](S)) \]
and \( \overline{p} < I(y^R, y^D; S') \cdot p' \). Then we have 2 cases:

- \( \overline{p} = D(x^D_i) \):
  then by definition of \( P_D \) it contains a resolution path for \( x \) and by induction hypothesis on the class \( (D < L) \) there is a definition \( x^D_i \in Env_D[I, S](S) \): contradiction.

- \( \overline{p} = I(y^R, y^D; S') \cdot p' \land \overline{p} < p' \):
  then by \( (\circ) \) and lemma 10, we have that:
  \( I, \{ S \} \cup S \vdash p' : S' \rightarrow x^D_i \)
  which contradicts \( (**) \) since \( p' < p' \).

\textbf{Case L. (\( \leq \))}: Assume \( x^D \in \Delta(P_L[I, S](S)) \). Then there is a path \( p \) such that:

\begin{align*}
(\ast) & \quad I, S \vdash p : S \rightarrow x^D \\
(\ast\ast) & \quad p \in I(\_ , \_ )^* \cdot D(\_ )
\end{align*}

By rule \( V' \) we have that:

\begin{align*}
(i) & \quad I, S \vdash p : S \rightarrow x^D \\
(ii) & \quad p \text{ is a minimal path for } x \text{ under } I, S.
\end{align*}

By case analysis on the first step of \( p \):

1) Case \( p = D(x^D) \):
\( x^D \) is trivially in \( Env_D[I, S](S) \) and thus in \( Env_L[I, S](S) \).

2) Case \( p = I(y^R, y^D; S') \cdot p' \):
By lemma 11 and \( (\ast) \) we have:
\( I, \{ S \} \cup S \vdash p' : S' \rightarrow x^D \)
Combining this with \( (i) \) we have \( p \in P_L[I, S](S) \) so by induction on the class, \( x^D \in Env_L[I, S](S) \). Moreover, there is no \( i \) with \( x^D_i \in Env_D[I, S](S) \): if there were, we would have a path \( q = D(x^D) \) with \( q < p \) and \( I, S \vdash q : S \rightarrow x^D \); contradicting \( (ii) \). So \( x^D \in Env_L[I, S](S) \).

\textbf{Case V}: We split the two directions:

\textbf{Case V. (\( \Rightarrow \))}:
\( x^D \in Env_V[I, S](S) \Rightarrow \) (by definition of \( Env_L \))
\( x^D \in Env_L[I, S](S) \land Env_V[I, S](S) \)

By case on the set \( x^D \) comes from:

1) If \( x^D \in Env_L[I, S](S) \) then by induction on the class (since \( L < V \) ) we get:
\( x^D \in \Delta(P_L[I, S](S)) \)
By definition of \( P_L \) there is a path \( p \) in \( I(\_ , \_ )^* \cdot D(\_ ) \) such that:
\( I, S \vdash p : S \rightarrow x^D \)
And we have the conclusion:
\( x^D \in \Delta(P_V[I, S](S)) \)

2) If \( \forall i, x^D_i \notin Env_V[I, S](S) \) and \( x^D \in Env_P[I, S](S) \) then by the induction on the class (since \( P < V \) ) we get that: \( x^D \in \Delta(P_P[I, S](S)) \).
By definition of \( P_P \) there exists \( p' \) such that \( p = P \cdot p' \) and:
Given (\(\ast\)), to prove that \(x^D \in \Delta(PV[I\{S\}](S))\) it is sufficient to prove that \(p\) is minimal:

Assume for contradiction that there is a path \(\bar{p}\) such that:

\[(\circ) \ I, S \vdash \bar{p} : S \rightarrow x^D\]

and \(\bar{p} < P : p'\). Then we have 2 cases:

- \(\bar{p} \in I(\_,\_)^* \cdot D(\_)\):
  
  Then by (\(\circ\)) and Lemma 8 there is a path \(\bar{p}'\) and a definition \(x^D_{\bar{p}'}\) such that:

  \[(i) \ I, S \vdash \bar{p}' : S \rightarrow x^D_{\bar{p}'}\]
  \n  (ii) \(\bar{p}' \leq \bar{p}\)

From (ii) we know \(\bar{p}' \in I(\_,\_)^* \cdot D(\_)\) and thus \(\bar{p}' \in P_L[I, S](S)\). By induction (\(L < V\)) we get that \(x^D_{\bar{p}'} \in Env_L[I, S](S)\): contradiction.

- \(\bar{p} = P \cdot p' \wedge p' < p'\):
  
  then by (\(\circ\)) and Lemma 10, we have that:

  \[I, \{S\} \cup S \vdash \bar{p}' : P(S) \rightarrow x^D_{\bar{p}'}\]

which contradicts (\(\ast\ast\)) since \(\bar{p}' < p'\).

Case V. (\(\rightarrow\)): Assume \(x^D \in \Delta(PV[I\{S\}](S))\) then there is a path \(p\) such that:

\[(\ast) \ I, S \vdash p : S \rightarrow x^D\]

By rule (\(V'\)) we have that:

\[(i) \ I, S \vdash p : S \rightarrow x^D\]

(ii) \(p\) is a minimal path for \(x\) under \(I, S\).

By case analysis on the first step on \(p\):

1) Case \(p \in I(\_,\_)^* \cdot D(\_)\): then \(x^D\) is trivially in \(\Delta(P_L[I, S](S))\) and thus in \(Env_L[I, S](S)\) by induction on class and thus in \(Env_P[I, S](S)\).

2) Case \(p = P \cdot p'\):

By Lemma 11 and (\(\ast\)) we have:

\[I, \{S\} \cup S \vdash p' : P(S) \rightarrow x^D\]

From this and (i) we know \(p \in P_P[I, S](S)\) and by induction on the class, 
\(x^D \in Env_P[I, S](S)\). To show \(x^D \in Env_P[I, S](S)\), it suffices to show that 
\(Env_L[I, S](S)\) cannot contain a definition \(x^D_{\bar{p}'}\). Suppose for contradiction that it does. Then by induction on class there is a resolution path \(\bar{p}\) for \(x^D_{\bar{p}'}\) in \(P_L[I, S](S)\). By definition of \(P_L\), \(\bar{p}\) has the form \(I(\_,\_)^* \cdot D(\_)\), so \(\bar{p} < p\), contradicting (ii).

This last case concludes the proof. \(\square\)
B.6 α-equivalence

We prove Lemma 4 about the absence of declaration in the equivalence class of a free variable:

Lemma 14 (Lemma 4). The equivalence class of a free variable can not contain any other declaration:

\[ \forall x_i^D, i \vdash x_i \implies i = \bar{x} \]

Proof. First, since \( \top \) is the only path to the free variable definition we have:

\[ \forall x_i^R p, (\Pi \vdash p : x_i^R \longrightarrow x_i^D) \implies p = \top \]

Then since \( \top \) is less specific than any other path and using the \((V)\) rule we have:

\[ \forall x_i^R, (\Pi \vdash \top : x_i^R \longrightarrow x_i^D) \implies \forall p i', \Pi \vdash p : x_i^R \longrightarrow x_i^D \implies i' = \bar{x} \]

So if there is resolution to the free variable declaration, then it is the only possible resolution for the reference. Finally, we prove by induction on the equivalence relation \( i \sim i' \) that any term equivalent to \( \bar{x} \) resolves to \( \bar{x} \) or is \( \bar{x} \) itself:

\[ \forall i j, i \vdash j \implies \\
( (j = \bar{x} \lor \Pi \vdash \top : x_i^R \longrightarrow x_i^D) \implies (i = \bar{x} \lor \Pi \vdash \top : x_i^R \longrightarrow x_i^D) ) \land \\
( (i = \bar{x} \lor \Pi \vdash \top : x_i^R \longrightarrow x_i^D) \implies (j = \bar{x} \lor \Pi \vdash \top : x_j^R \longrightarrow x_j^D) ) \]

Instantiating \( j \) with \( \bar{x} \) finishes the proof. \( \square \)