



Defining Name Accessibility Using Scope Graphs

Aron Zwaan  

Delft University of Technology, The Netherlands

Casper Bach Poulsen  

Delft University of Technology, The Netherlands

Abstract

Many programming languages allow programmers to regulate *accessibility*; i.e., annotating a declaration with keywords such as `export` and `private` to indicate where it can be accessed. Despite the importance of name accessibility for, e.g., compilers, editor auto-completion and tooling, and automated refactorings, few existing type systems provide a formal account of name accessibility.

We present a declarative, executable, and language-parametric model for name accessibility, which provides a formal specification of name accessibility in Java, C#, C++, Rust, and Eiffel. We achieve this by defining name accessibility as a predicate on *resolution paths* through *scope graphs*. Since scope graphs are a language-independent model of name resolution, our model provides a uniform approach to defining different accessibility policies for different languages.

Our model is implemented in Statix, a logic language for executable type system specification using scope graphs. We evaluate its correctness on a test suite that compares it with the C#, Java, and Rust compilers, and show we can synthesize access modifiers in programs with holes accurately.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Language features; Theory of computation → Program constructs

Keywords and phrases access modifier, visibility, scope graph, name resolution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.47

Related Version *Extended Version*: <https://doi.org/10.48550/arXiv.2407.09320> [36]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: <https://doi.org/10.4230/DARTS.10.2.27>

Acknowledgements We thank Friedrich Steimann for challenging us to specify and formalize access modifiers using scope graphs, and the anonymous reviewers for their helpful comments.

1 Introduction

Many programming languages, especially object-oriented ones, support *information hiding*, i.e., regulating from which positions in a program a declaration can be accessed. Information hiding is used to enforce invariants of particular code units, implement design patterns (e.g. the singleton pattern), improve modularization, limit public APIs to offer guidance to library users and guarantee forward compatibility. Support for information hiding is usually provided using *access modifier keywords*¹ (*access modifiers* for short), such as `public`, `protected`, `internal` and `private`. Each of these corresponds with a particular accessibility policy that is validated by the type checker.

Although recent research has not paid much attention to access modifiers, there are still good reasons to study their semantics. First, understanding access modifiers is required to implement (alternative) compilers and editor services correctly. In particular, disregarding accessibility may result in incorrect name binding, and hence incorrect program behavior. Second, formalizing access modifiers enables reasoning about the meaning of programs.

¹ Other common names include “access specifier” or “visibility modifier”.



47:2 Defining Name Accessibility Using Scope Graphs

```

package p1;
class A {
  int x;
}

package p2;
class B extends p1.A { }

package p1;
class C extends p2.B {
  int y = x;
}
(a) Inheritance through Packages.

package p;
class A {
  protected int x;
}

class B extends A {
  private int x;
}

class C extends B {
  int y = x;
}
(b) Inaccessible or Shadowed?

package p;
class A {
  private int x = 0;
  protected int y = 1
}

class B {
  int x = 3;
  int y = 4;
  class C extends A {
    int z = x + y
  }
}
(c) Accessibility and Shadowing.

```

■ **Figure 1** Examples of intricate Access Modifier semantics. Classes are assumed to be public.

Finally, program transformation tools, such as automated refactorings, must handle the semantics of accessibility correctly. This is especially relevant for research on large-scale automated transformations, aimed at dealing with large (legacy) codebases. It is often infeasible to check transformations performed with such tools manually. Thus, the correctness of these transformations must be guaranteed through other means.

The meaning of access modifiers can be intricate in corner cases. We illustrate that using the examples in Figure 1. In Figure 1a, there is an inheritance chain, where class C extends class B, which itself extends A. Classes A and C reside in package p1, while B is in p2. Class A defines a package-accessible field x, which is accessed in C. The question here is whether that access is actually allowed. One could reason that it is correct, as the access occurs in the same package as the declaration, so a package-level declaration should be visible. On the other hand, one could consider x not inherited by B [11, §8.2], and thus not inherited by C either. In fact, the Java language designers chose the second option, rejecting this program [23, §4.2]. Using ((A) this).x is accepted however.

Something similar happens in Figure 1b. Here, one can consider the reference x in class C to be invalid, as the field in class B is inaccessible. Alternatively, under the assumption that B.x is *out of scope*, the reference can be valid, pointing to A.x. In this case, Java checks accessibility *after shadowing*, so this program is again rejected. However, in Figure 1c, accessibility does influence the binding. The reference x binds to the field of the *enclosing* class B, as the field inherited from class A is inaccessible. However, reference y binds to the field inherited from A. Thus, in this case, the *accessibility* of the inherited fields determines the resolution of x and y; i.e., accessibility is checked *before shadowing*. This shows that specifying accessibility is essential to defining the name binding of a language correctly.

Unintuitive semantics of accessibility occurs in non-object-oriented languages as well. For example, the accessibility scheme of Agda seems simple: definitions are either public or module-private, and imported definitions can be re-exported. However, issue #5461² reports that re-exports in a private block are still exposed to the outside world. While this intuitively seems wrong to most commenters, an argument is made that this is actually the intended behavior. The discussion stalls shortly after a remark that talking about intended behavior is “meaningless without a specification”.

² <https://github.com/agda/agda/issues/5461>

These examples show that the meaning of access modifiers is not always obvious. Hence, language designers should define their semantics unambiguously. Ideally, that is done through *specifications* containing *inference rules*. Inference rules allow unambiguous interpretation of the meaning of programming language constructs, including name binding. However, perhaps surprisingly, a general model for defining access modifiers has never been proposed.

Perhaps closest is the work of Steimann and Thies [25] (later incorporated in the JRRT refactoring tool [23]). They propose a constraint-based approach to automating refactorings in Java, by collecting and solving *accessibility constraints*. These constraints are generated using *constraint generation rules*, which cover the access rules the Java compiler enforces. By solving these constraints, changes in accessibility implied by the refactoring can be inferred, yielding type- and behavior-preserving refactorings.

Steimann and Thies' work solves the problem of making refactorings in Java sound regarding accessibility. However, it does not yet give a high-level explanation of the meaning of access modifiers. This is partly because the constraint generation rules need several low-level details to catch some intricate corner cases, but also because the function that computes the minimal required accessibility level is not given, as it was “unpleasant to specify” and “of no theoretical interest” [25, §5.2]. Therefore, their work cannot easily be adapted to a different language or a different application (e.g., a type checker).

To advance the state of the art, we pursue the following goals:

- Explain the meaning of access modifiers.
- Explain the (subtle) differences between access modifiers in different languages.
- Provide a framework for experimenting with feature combinations that do not (yet) exist in other languages.

To this end, we do not fully formalize one particular language, but rather define a toy language that incorporates and combines a large number of accessibility features. To abstract over low-level name resolution details, we use *scope graphs* [18, 27, 22, 38]. In this paper, we demonstrate this is a natural fit, because accessibility can be expressed as a predicate over paths in a scope graph. The specification is written in the logic language Statix [27, 22], which has a well-defined declarative semantics and also supports generating executable type-checkers automatically.

We compare these executable type checkers with reference compilers of Java, C#, and Rust, showing that we accurately captured the semantics of access modifiers in some real-world languages. Moreover, using Statix/scope graphs as a basis for (*language-parametric*) refactorings is an active topic of research [16, 29, 15, 3]. We envision that this will provide accessibility-aware refactorings similar to Steimann et al., without requiring significant additional effort. This is substantiated by the fact that Statix-based code completion [19] proposes an access modifier if and only if it would not cause accessibility errors elsewhere in the program.

In summary, the contributions of this paper are as follows:

- We provide a systematic classification of accessibility features (Section 2);
- we apply our taxonomy to Java, C++, C#, Rust, and Eiffel (Section 2);
- we present a specification of (various versions of) accessibility on modules (Section 5), subclasses (Section 6), and their conjunctive and disjunctive combination (Section 7);
- we extend our specification with accessibility-restricting inheritance (Section 8);
- we prove some theorems about our model, showing it is well-behaved (Section 9); and
- we implement our specification in Statix, and compare it with the standard compilers of Java, C#, and Rust. Moreover, we show access modifiers can be synthesized accurately using Statix-based Code Completion [19] (Section 10).

This paper comes with an artifact that allows reproducing the evaluation [35], and appendices containing a full specification of the access modifiers and proofs of the stated theorems [36].

2 Access Modifiers in Real-World Languages

In this section, we explore the design space of access modifiers as they occur in real-world languages. We first motivate why languages have access modifiers (Section 2.1). After that, we discuss common accessibility features (Section 2.2), summarizing them in a feature model (Section 2.3).

2.1 Why Accessibility?

Most programming languages allow programmers to define entities (variables, functions, types, etc.), and assign a name to them. That name can then be used to refer to the introduced entity from other positions in the program. However, as there is typically a large number of entities within a software project, most languages offer a notion of modularization to group related definitions. Equally named definitions in different modules can be distinguished by qualifying them with the name of the module in which they reside. Unqualified (or partially qualified) names by default resolve within their enclosing module, or imported modules. Details of this scheme differ from language to language, but generally aim to make definitions easy to refer to (e.g., by minimizing the number of required qualifiers), while trying to be unambiguous to the compiler and the programmer.

However, these rules may often be too lenient with respect to the intention of the programmer. A definition may be accessible from scopes where it is not intended to be used. This can have detrimental effects on the quality of a software artifact. For example, exposing all internal definitions of a library makes it (1) less intuitive to its users, (2) prone to forward compatibility issues and technical debt (e.g. strong coupling).

For these reasons, many programming languages provide constructs that give *the programmer* control over the regions of code where a definition can be accessed. For example, in many object-oriented languages, a class can access fields from its ancestor classes by default (language-controlled). However, if the programmer does not want a field to be accessible from subclasses, they can add a `private` access modifier. This modifier *prevents* access from all other classes (programmer-controlled). Although many constructs that provide access control to the programmer can be envisioned, most languages settle on a limited set of keywords that can be attached to a definition. In practice, this relatively simple scheme has proven powerful enough to cover most use cases.

2.2 Accessibility in Practice

Next, we explore how languages typically provide modularization and accessibility features.

Modules. A common feature that provides modularization is *modules* (also called “package” or “namespace”). A module is a syntactic construct that introduces a named collection of definitions. Members of modules can be accessed using the name of the module, for example in a preceding import statement, or as a qualifier to the name of the member that is accessed.

Hiding a definition from other modules is the simplest accessibility restriction that can be applied with respect to modules. For example, Java declarations without an access modifier can only be accessed within the same package. Rust items without a modifier behave similarly, except that declarations can still be accessed from submodules.

Some languages have multiple notions of modularization. For example, C# has assemblies, namespaces, and files, where a namespace can comprise multiple files, and/or a file can contain multiple namespaces. The `internal` keyword in C# restricts accessibility to the *assembly*, and the `file` keyword (introduced in C# 11 [32]) to the current file. Similarly, Java 9 introduces *modules* [21], with features to restrict access from external modules.

```

mod outer {
  mod inner {
    pub x = 42;
  }
  pub use inner::x;
}

fn main() {
  // ERROR: inner is inaccessible:
  // let x = outer::inner::x;
  let x = outer::x;
  println!("{}", x)
}

```

■ **Figure 2** Re-exports can change Accessibility.

Some languages give some more control over *which* modules a declaration can be accessed from. For example, Rust has the `pub(in path)` access modifier, where *path* refers to some enclosing module. This enables programmers to expose items to an arbitrary ancestor.

Imports usually do not affect the visibility of a declaration. A notable exception to this rule is *re-exporting* (e.g., as implemented in Rust), which can actually *change* the visibility of a declaration, as shown in Figure 2. In this program, the module `inner` is accessible in `outer`, but not in its parent (the root scope). Therefore, the function `main` cannot access its field `x`. However, `outer` re-exports `inner::x`, which gives rise to a new definition `outer::x`. As `outer` is accessible in the root scope, so is this definition. Hence, via the re-export, `main` can access `x`, although the original declaration was hidden.

From an accessibility point of view, re-exporting can typically be considered as a combination of an import and a declaration, where the declaration always points to the imported member. The re-exported item (`inner::x` in the example) should be accessible from the location of the *re-export*. References to the re-export should have access to the location of the re-export, but not necessarily to the location of the original declaration. In fact, for any access path, it does not matter whether the declaration is a re-export or not.

Classes. A special modularization concept is the notion of *classes*, which represent composite data types with associated operations (methods). Where simple modules only have a static interpretation, an arbitrary number of class instances can exist at runtime.³ While modules can implicitly be related to each other by their relative position, such a relation does not exist for classes. However, classes can extend other classes, ensuring the subclass inherits the fields of its parent class. This creates an inheritance hierarchy orthogonal to the module hierarchy.

Object-oriented languages usually provide modifiers to control accessibility over the inheritance chain. For example, Java and C# have a `private` keyword, which prevents access outside the defining class. Additionally, the `protected` keyword allows access from subclasses, but prevents access from any other location.

In Java and C#, the accessibility level is inherited with the field. That means, if a field in the superclass is `protected`, it will be protected in the subclass as well. However, C++ allows restricting the accessibility of members of the parent class. A `private` modifier on extends-clauses will make all inherited public/protected members private on instances of the subclass. Similarly, a `protected` modifier will make all inherited public members protected.

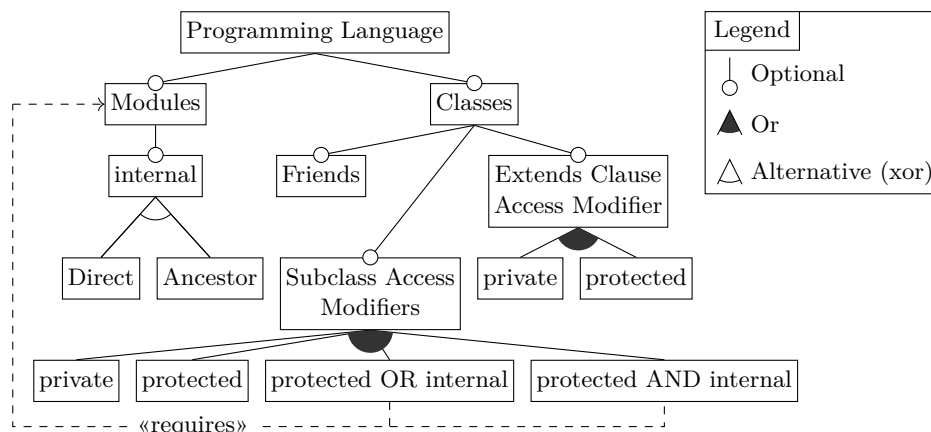
Finally, some languages allow specifying “friend” classes, which grant the friend access to its members. This enables fine-grained access control, independent from module and class hierarchies. While discouraged in C++, Eiffel provides only this access control mechanism.

³ At this point, we slightly over-simplify the reality. For example, neither parameterized modules (ML) nor objects (e.g. Scala/Kotlin) fit in this scheme. We made this choice deliberately, to cover the most prevalent cases. We conjecture that the techniques we develop for classes can be applied to parameterized modules (and vice versa for modules and objects) but leave explicating that to future work.

Interaction. Accessibility restrictions on modules and classes be combined. This is very explicit in C#, which has `protected internal` and `private protected` as additional modifiers. The former permits access from within the assembly (similar to `internal`) and to subclasses (similar to `protected`), even if they live outside the assembly. Analogously, `private protected` grants access to subclasses in the same assembly only, which is equivalent to the conjunction of `internal` and `protected`.

2.3 Classification

These concepts are organized and related in the feature model in Figure 3. Following the previous discussion, the main features are modules and classes. We have only a single feature for modules, because the different variants are (apart from C#'s files and namespaces) typically not mutually nested. The `internal` keyword can either relate to the containing module (Direct) or an arbitrary parent module (Ancestor). We explore this further in Section 5.



■ **Figure 3** Feature Model for Access Control.

■ **Table 1** Languages classified according to the feature model in Figure 3.

	Java	C#	C++	Eiffel	Rust
Modules	✓	✓	✓		✓
Internal	Direct	Direct ⁴	Direct		Ancestor
Classes	✓	✓	✓	✓	
Friends			✓	✓	
Subclass Acc. Mod.	✓	✓	✓	✓	
<i>private</i>	✓	✓	✓	✓	
<i>protected</i>		✓	✓		
<i>protected internal</i>	✓	✓			
<i>protected & internal</i>		✓	✓		
Extends Clause Acc. Mod.			✓		
<i>private</i>			✓		
<i>protected</i>			✓		

In the Classes category, the three subfeatures denote the three mechanisms for access control: Friends allow access to other classes by name, Subclass Access Modifiers are access modifiers on definitions that determine how it is accessible within the class hierarchy (Sections 6 and 7), and Extends Clause Access Modifiers (Section 8) are access modifiers on extends clauses, as seen in C⁺⁺. The latter two have subfeatures for each concrete keyword associated with the access control mechanism. For that reason, `private` and `protected` occur twice: once on definitions and once on extends clauses. Table 1 classifies several languages according to this scheme. In the remainder of this paper, we develop AML (Access Modifier Language), a language that covers all features. To this end, we first introduce scope graphs (Section 3), and a base language for AML (Section 4).

3 Using Scope Graphs to Model Name Binding in Programs

In the previous section, we sketched the landscape of access modifiers. This discussion was based largely on prose specifications as well as experiments with compiler implementations. No language specification we are aware of provides a more rigorous model of accessibility (or even non-lexical name binding). In this section, we introduce *scope graphs* [18, 27, 22, 38], and argue that they provide a suitable framework for such a model. Section 4 introduces AML (Access Modifier Language), a toy language with a type system defined using scope graphs. Sections 5–8 will extend this language with all accessibility features from Figure 3.

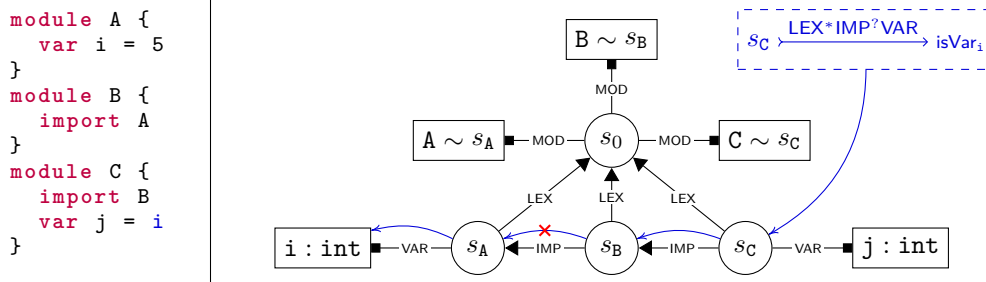
3.1 Scope Graphs as A Model for Name Binding

From a name binding perspective, classes and modules have some similarities. Each of these constructs can be thought of as introducing a “scope” (region of code), in which declarations live, and in which names can be resolved. Scopes are related to each other in various ways. First, modules are related according to their relative position in the abstract syntax tree. In addition, imports and extends clauses relate arbitrary modules and classes, respectively. Resolving a reference corresponds to finding a matching declaration in a scope that is reachable from the scope of the reference. For example, a reference may resolve to a declaration if it lives in a lexically enclosing scope, or in a module that is imported in an enclosing scope.

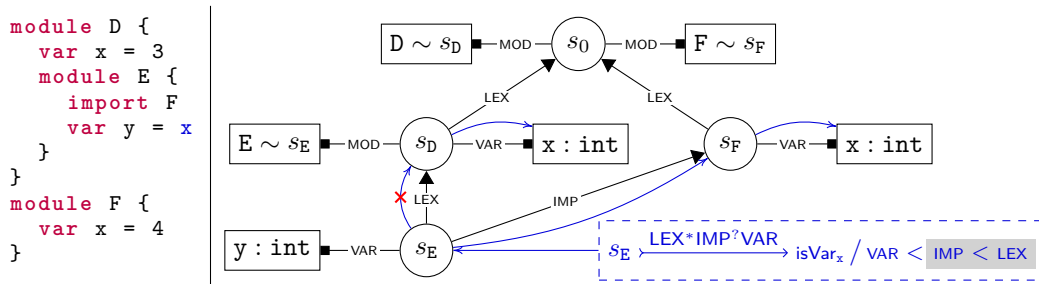
Scope graphs [18, 27, 22, 38] make this more precise. In this model, the name binding structure of a program is represented by a graph. Figure 4 (adapted from Poulsen et al. [20, Fig. 1]) gives an example program and its corresponding scope graph. A scope is represented by a circular node in the graph. For example, s_0 represents the global scope, and s_A , s_B and s_C represent the bodies of modules A, B, and C, respectively. Scopes are related using labeled, directed edges. For example, s_A is lexically enclosed by s_0 , and thus the graph contains an edge from s_A to s_0 with label LEX. Similarly, s_B imports s_A , and thus the graph contains an edge $s_B \xrightarrow{\text{IMP}} s_A$. Finally, scope graphs contain declarations. For example, a declaration of i in scope s_C is represented by the $s_C \xrightarrow{\text{VAR}} i : \text{int}$ edge/node pair. Similarly, the modules are declared in the root scope (e.g., $s_0 \xrightarrow{\text{MOD}} A \sim s_A$). The language specification determines which data is included in the declaration. Similarly, the labels for edges and declarations can be chosen to match the (binding) constructs of the language.

⁴ Either the most direct enclosing *file* (**file**), or most directly enclosing *assembly* (**internal**), possibly bypassing some namespaces.

47:8 Defining Name Accessibility Using Scope Graphs



■ **Figure 4** Reachability example. The IMP^2 part in the regular expression prevents traversal over the second IMP edge.



■ **Figure 5** Shadowing example. The highlighted label order causes the edge to s_F to have priority.

Reachability. To resolve a reference, a *query* is executed to find a valid path in the scope graph from the scope of the reference to a matching declaration. Queries give specification writers several options to filter paths, to retain only valid paths. First, a unary predicate selects valid declarations. Usually, this predicate matches declarations with the name of the reference. Second, a *regular expression* over labels is used to select valid paths. This regular expression can, for example, be used to prevent transitive imports, or accessing members in a lexical parent of an imported module.

Figure 4 illustrates this with the query for i in module C (dashed blue box). The parameter on the arrow ($\text{LEX}^*\text{IMP}^2\text{VAR}$), is a regular expression that defines which paths to declarations are valid. The LEX^* indicates that a path may traverse an arbitrary number of LEX -edges. This corresponds to looking for variables in enclosing scopes. Next, the IMP^2 part indicates that zero or one IMP -edges can be traversed. Finally, the regular expression ends with VAR to ensure all paths resolve in variable declarations only, excluding e.g. modules. The isVar_i parameter matches all variable definitions with name i (isVar is defined in the next section). The candidate path (shown as blue edges) does not match this regular expression. Because IMP -labeled edges may only be traversed one time, the step to s_A cannot be made. In other words: the declaration of i in A is not *reachable* from C .

Visibility. Not every declaration that is reachable (i.e., for which a valid access path exists) can actually be referenced, due to *shadowing*. For example, in most languages, local definitions have higher priority than imported ones. We call reachable declarations that are not shadowed by any other declaration *visible*.

In scope graphs, visibility can be encoded using a partial order on labels. For example, an order $\text{VAR} < \text{IMP}$ encodes that (local) variable declarations shadow imported declarations. This is illustrated in Figure 5. The reference x in module F can refer to the declaration in

module D as well as the one in module E. Because the label order (third argument) indicates that imports shadow lexically enclosing scopes ($\text{IMP} < \text{LEX}$). Thus, the variable resolves to the declaration in s_F . Alternatively, if $\text{LEX} < \text{IMP}$, it would resolve to x in s_D . Finally, if neither $\text{LEX} \not< \text{IMP}$ nor $\text{IMP} \not< \text{LEX}$, both paths would be included in the query result.

In summary, scope graphs model the name binding structure of a program using nodes for scopes and declarations, and edges for relations between those. Queries can be used to model reference resolution. A query selects a declaration when (1) it matches some predicate, and (2) there exists a path to it of which the labels match a regular expression, and (3) no other paths that traverse labels with higher priority exist. The result of a query is a set of paths that lead to these matching declarations.

Accessibility. We can model extensibility using plain scope graphs by including accessibility information in the *declaration*. In other words, a declaration of a variable in a scope graph contains not only a name and a type, but also its accessibility level. *After resolution*, we check if the path that the query returns is actually valid according to the accessibility level of the declaration. For example, if a variable is private, but an EXT -edge (for class *extension*) is traversed, an error is emitted. With this pattern, we can model all accessibility features.

Notation. Figures 4 and 5 introduce the graphical notation of scope graphs. In text, variable s ranges over scopes, and S over sets of scopes. Moreover, we use the following notation for assertions on scope graphs: $s_1 \xrightarrow{L} s_2 \in \mathcal{G}$ means “scope graph \mathcal{G} has an L-labeled edge from s_1 to s_2 ”, and $s \xrightarrow{D} d \in \mathcal{G}$ means that \mathcal{G} has a declaration with data d under label D in scope s . Moreover, we write queries in the following way:

$$\text{query}_{\mathcal{G}} s \xrightarrow{R} \mathcal{P} / \mathcal{O} \mapsto R$$

where \mathcal{G} is the scope graph in which the query is resolved, s is the scope in which the resolution starts, R is the regular expression that paths must adhere to, and \mathcal{P} is the predicate that declarations must match. \mathcal{O} is the strict partial order on labels used for shadowing. It is usually written as $L_1 < L_2 < \dots < L_n$. We omit the label order when there is no shadowing. R is the result set containing tuples of paths and declarations. When we expect a single result, we use $\{ \langle p, d \rangle \}$ to match on the value in the set. Paths are alternating sequences of scopes and labels, written as $s_1 \xrightarrow{L_1} s_2 \dots s_m$. Paths do not include the declaration it resolved to, but stop at the scope in which the declaration occurs. The functions $\text{src}(p)$, $\text{tgt}(p)$ refer to the source and target scope of a path, respectively. $\text{scopes}(p)$ denotes all scopes in a path.

4 AML: The Base Language

In the next sections, we show how scope graphs support intuitive formalization of accessibility. We will do so by defining *AML* (Access Modifier Language). The base syntax (which will be extended later) is given in Figure 6. In AML, a program consists of a list of modules. Each module can define other modules, import other modules, and contain class definitions. A class can optionally extend another class, and contains a list of field declarations. Each field has an access modifier, and is initialized by some expression. Possible expressions include references, integer constants, class instance creation, field access, and binary operations.

At the right-hand side of Figure 6, the scope graph parameters are shown. There are three labels that connect scopes. LEX denotes lexical scoping, IMP denotes imports, and EXT class extension. The other three labels are used for declarations. MOD is used for module declarations, CLS for classes, and VAR for variables/fields. Next, we assume that each *module*

47:10 Defining Name Accessibility Using Scope Graphs

$\langle prog \rangle ::= \langle mod \rangle^*$ $\langle mod \rangle ::= \text{module } \langle x \rangle \{ \langle md \rangle^* \}$ $\langle md \rangle ::= \langle mod \rangle \mid \text{import } \langle x \rangle \mid \langle cls \rangle$ $\langle cls \rangle ::= \text{class } \langle x \rangle (: \langle acc \rangle \langle x \rangle)^? \{ \langle cd \rangle^* \}$ $\langle cd \rangle ::= \langle acc \rangle \text{ var } \langle x \rangle = \langle e \rangle \mid \langle cls \rangle$ $\langle acc \rangle ::= \text{public} \mid \dots$ $\langle e \rangle ::= \langle n \rangle \mid \langle x \rangle \mid \text{new } \langle x \rangle () \mid \langle e \rangle . \langle x \rangle \mid \dots$	$\langle l \rangle ::= \text{LEX} \mid \text{IMP} \mid \text{EXT}$ $\quad \mid \text{MOD} \mid \text{CLS} \mid \text{VAR}$ $\quad \mid \text{THIS}_M \mid \text{THIS}_C$ $\langle d \rangle ::= \text{mod } \langle x \rangle : \langle s \rangle$ $\quad \mid \text{cls } \langle x \rangle : \langle s \rangle$ $\quad \mid \text{var } \langle x \rangle : \langle T \rangle @ \langle A \rangle$ $\quad \mid \langle s \rangle$ $\langle T \rangle ::= \text{int} \mid \text{inst } \langle s \rangle$ $\langle A \rangle ::= \text{PUB} \mid \dots$
---	--

■ **Figure 6** Syntax of AML. The highlighted positions indicate extensions in later sections. The syntax of the complete language can be found in Appendix A [35].

Data Matching Predicates

$\mathcal{P}(d)$

$$\begin{array}{ll} \text{isMod}_x(\text{mod } x' : s) \Leftarrow x = x' & \text{isCls}_x(\text{cls } x' : s) \Leftarrow x = x' \\ \text{isVar}_x(\text{var } x' : T @ A) \Leftarrow x = x' & \text{isScope}_s(s') \Leftarrow s = s' \end{array}$$

Class Members

$s \vdash_{\mathcal{G}} cd \text{ OK}$

$$\text{D-DEF} \frac{s \vdash_{\mathcal{G}} e : T \quad s \vdash_{\mathcal{G}} acc \Rightarrow A \quad s \xrightarrow{\text{VAR}} \blacksquare (\text{var } x : T @ A) \in \mathcal{G}}{s \vdash_{\mathcal{G}} acc \text{ var } x = e \text{ OK}}$$

Type of Expression

$s \vdash_{\mathcal{G}} e : T$

$$\text{T-VAR} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{EXT}^* \text{VAR}} \text{isVar}_x / \text{VAR} < \text{EXT} < \text{LEX} \mapsto \{ \langle p, \text{var } x : T @ A \rangle \}}{s \vdash_{\mathcal{G}} x : T} \quad s \vdash_{\mathcal{G}} p ! A$$

$$\text{T-FLD} \frac{\text{query}_{\mathcal{G}} s_c \xrightarrow{\text{EXT}^* \text{VAR}} \text{isVar}_x / \text{VAR} < \text{EXT} \mapsto \{ \langle p, \text{var } x : T @ A \rangle \} \quad s \vdash_{\mathcal{G}} e : \text{inst } s_c \quad s \vdash_{\mathcal{G}} p ! A}{s \vdash_{\mathcal{G}} e.x : T}$$

Access Modifier

$s \vdash_{\mathcal{G}} acc \Rightarrow A$

Access Policy

$s \vdash_{\mathcal{G}} p ! A$

$$\text{A-PUB} \frac{}{s \vdash_{\mathcal{G}} \text{public} \Rightarrow \text{PUB}}$$

$$\text{AP-PUB} \frac{}{s \vdash_{\mathcal{G}} p ! \text{PUB}}$$

Module and Class References

$s \vdash_{\mathcal{G}} x \overset{M}{\rightsquigarrow} s_m \quad s \vdash_{\mathcal{G}} x \overset{C}{\rightsquigarrow} s_c$

$$\text{Q-MOD} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{MOD}} \text{isMod}_x / \text{MOD} < \text{LEX} \mapsto \{ \langle p, \text{mod } x : s_m \rangle \}}{s \vdash_{\mathcal{G}} x \overset{M}{\rightsquigarrow} s_m}$$

$$\text{Q-CLS} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{IMP}^? \text{CLS}} \text{isCls}_x / \text{CLS} < \text{IMP} < \text{LEX} \mapsto \{ \langle p, \text{cls } x : s_c \rangle \}}{s \vdash_{\mathcal{G}} x \overset{C}{\rightsquigarrow} s_c}$$

■ **Figure 7** Typing Rules of AML. Accessibility is integrated at the highlighted positions. The full type system specification can be found in Appendix A [35].

scope has a THIS_M edge pointing to itself, and similarly, each class has a THIS_C scope pointing to itself. This will be used to resolve enclosing classes or modules. The sort $\langle d \rangle$ denotes the data that can be associated with scopes. Modules and classes are characterized by their name and the scope of their body. A field has a name, a type $\langle T \rangle$, and an accessibility level $\langle A \rangle$. Scopes that are not declarations implicitly map to themselves. To query declarations, we use the four predicates shown at the top of Figure 7, which each match a single kind of declaration. Depending on the type of access control we formalize, different access modifiers will be used. Therefore, we have left the $\langle acc \rangle$ and $\langle A \rangle$ productions partially unspecified. Each section will instantiate those appropriately.

Typing Rules. Figure 7 presents some typing rules of AML. The rules are written in a declarative style, where a scope graph \mathcal{G} that models the program is assumed. Constraints over the scope graph are used as premises. The highlighted premises show where accessibility is integrated into the type system. We now discuss each of the presented rules.

The **D-DEF** rule asserts a declaration is well-typed if the initialization expression e has some type T (first premise), the access modifier acc corresponds to some accessibility policy A (second premise), and an appropriate declaration exists in the scope graph (third premise). The accessibility policy is included in the declaration, which enables us to validate accessibility when type checking references.

Next, rule **T-VAR** defines how references are type checked in a current scope s . First, it performs a query that looks into the lexical context (LEX^*), parent classes (EXT^*), and eventually resolves to a variable declaration (VAR). It matches only variables with the same name as the reference (isVar_x). Regarding shadowing, it prefers local variables over variables from a parent class ($\text{VAR} < \text{EXT}$), and variables from parent classes over variables from enclosing classes ($\text{EXT} < \text{LEX}$). The query should return a single result, as the name would otherwise be ambiguous. From this result, the access path p , type T , and accessibility policy A are extracted. The path and the accessibility policy are used in the second (highlighted) premise ($s \vdash_{\mathcal{G}} p!A$), which asserts that “accessibility policy A grants access via path p in scope s ”. In future sections, we will define new accessibility policy rules, that may prohibit access of a variable, even if the query premise resolved properly.

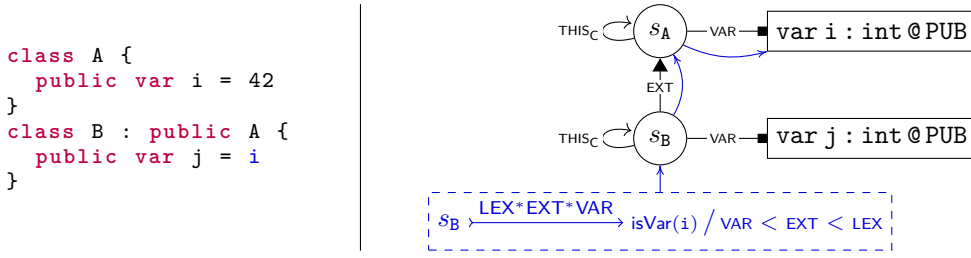
Note that, by having accessibility separated from the resolution, we do not capture the interaction between accessibility as shown in Figure 1c. We made this choice because the place where accessibility is integrated does not influence the access rules themselves, and this presentation allows more concise derivations, which makes the explanations more accessible. Appendix A.1 [35] shows how to integrate accessibility in the shadowing policy of a query, and is incorporated in the evaluation (Section 10).

For this base language, we only have the **public** access modifier. The **A-PUB** rule shows that this keyword corresponds to the **PUB** policy. The meaning of this policy is that access is allowed from any location, with any access path. This is encoded in the **AP-PUB** rule, which has no premises.

Finally, the last two rules define how references to classes and modules are resolved. Rule **Q-MOD** indicates that module reference x resolves to scope s_m if that scope is included in the closest module declaration with name x in the lexical context. Similarly, a class reference resolves to the scope of the closest class declaration s_c , preferring (non-transitively) imported classes over classes in the lexical context (**Q-CLS**).

Example. The example in Figure 8 shows two classes A and B. Both classes have a THIS_C -edge pointing to itself. Class B extends class A, which is represented by the $s_B \xrightarrow{\text{EXT}} s_A$ edge in the scope graph. Class A has a public field **i** with type **int**. The type as well as the corresponding

47:12 Defining Name Accessibility Using Scope Graphs



(a) Example program and (partial) scope graph.

$$\frac{\text{query}_{\mathcal{G}} s_B \rightsquigarrow \text{isVar}_i / \dots \mapsto \{ \langle s_B \xrightarrow{\text{EXT}} s_A, \text{var } i : \text{int} @ \text{PUB} \rangle \}}{s_B \vdash_{\mathcal{G}} i : \text{int}} \quad \boxed{s_B \vdash_{\mathcal{G}} s_B \xrightarrow{\text{EXT}} s_A ! \text{PUB}}$$

(b) Part of typing derivation that shows how access is granted by the PUB accessibility policy.

■ **Figure 8** Example AML program demonstrating the scope graph structure and name resolution with accessibility checking.

Enclosing Modules

$$\boxed{\vdash_{\mathcal{G}} s \uparrow_M S \quad \vdash_{\mathcal{G}} s \uparrow_M s}$$

$$\text{ENC-M} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{THIS}_M} \top \mapsto R \quad S_M = \{s_m \mid \langle p_m, s_m \rangle \in R\}}{\vdash_{\mathcal{G}} s \uparrow_M S_M}$$

$$\text{ENC-MI} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{THIS}_M} \top / \text{THIS}_M < \text{LEX} \mapsto \{ \langle p, s_m \rangle \}}{\vdash_{\mathcal{G}} s \uparrow_M s_m}$$

Enclosing Classes

$$\boxed{\vdash_{\mathcal{G}} s \uparrow_C S \quad \vdash_{\mathcal{G}} s \uparrow_C s}$$

$$\text{ENC-C} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{THIS}_C} \top \mapsto R \quad S_C = \{s_c \mid \langle p_c, s_c \rangle \in R\}}{\vdash_{\mathcal{G}} s \uparrow_C S_C}$$

$$\text{ENC-CI} \frac{\text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{THIS}_C} \top / \text{THIS}_C < \text{LEX} \mapsto \{ \langle p, s_c \rangle \}}{\vdash_{\mathcal{G}} s \uparrow_C s_c}$$

■ **Figure 9** Auxiliary relations for AML scope graphs.

PUB access policy are included in the scope graph declaration. Similarly, class B has a field j. The initialization expression of j references i, which is represented with the query shown in the dashed box.

Figure 8b shows the part of the typing derivation that checks the highlighted reference. Reference i is type checked in scope s_B , and has type `int`. The first premise repeats the query shown in the scope graph, with the parameters and result made explicit. In particular, the resolution path is $s_B \xrightarrow{\text{EXT}} s_A$. The validity of this path is checked by the second premise, which is satisfied by the **AP-PUB** rule.

Auxiliary Relations. Finally, Figure 9 presents some auxiliary relations that we will use later. First, the $\vdash_{\mathcal{G}} s \uparrow_M S_M$ relation asserts that S_M is the set of scopes of the enclosing modules of s . It is defined as a query that looks for a `THISM` edge in the lexically enclosing

scopes. There is no shadowing, so R can contain multiple results in the case of multiple nested modules. The result R is translated to the set of module scopes by discarding the access paths.

This relation is inhabited for any enclosing module scope. The second relation $\vdash_{\mathcal{G}} s \uparrow_{\mathbf{M}} s_m$ is only inhabited for the *innermost* enclosing module s_m . The query in its definition finds the closest $\text{THIS}_{\mathbf{M}}$ -edge, which is enforced by the shadowing policy $\text{THIS}_{\mathbf{M}} < \text{LEX}$. Thus, the query returns only one result, from which the module scope s_m is extracted. Analogously, $\vdash_{\mathcal{G}} s \uparrow_{\mathbf{C}} S_C$ relates s to all enclosing *class* scopes S_C , and $\vdash_{\mathcal{G}} s \uparrow_{\mathbf{C}} s_c$ is satisfied if s_c is the *innermost* enclosing class of s .

5 Defining Module Visibility

Some languages have access modifiers that regulate the visibility of a declaration in other modules. For example, in Rust, it is possible to write `pub(in ...)` to indicate in which module a declaration is visible. Similarly, some languages support giving particular classes access to an item. It is the primary accessibility mechanism for Eiffel, and C++'s `friend` modifier enables this as well. Less flexible approaches, such as Java's package visibility and C#'s `internal` keyword can be seen as special instances of this mechanism.

To demonstrate how these access policies can be encoded using scope graphs, we extend our base language as follows. Figure 10a introduces an additional modifier keyword `internal`, which can contain references to modules. The declaration is visible in these modules only. The corresponding accessibility policy `MOD` has a set of scopes, each corresponding to a name given in the keyword argument.

Next, we explain how this keyword is interpreted. An `internal` declaration is accessible if the reference occurs in a module that the arguments to the `internal` modifier give access to. This is formalized in the rules given in Figure 10b. Rule `A-INT` translates an `internal` access modifier to the `MOD` policy. Each module name argument to the modifier (x_i) is resolved relative to the current scope s . This yields a collection of module scopes s_i , which

$$\langle acc \rangle ::= \dots | \mathbf{internal} (\langle x \rangle^*) \quad \langle A \rangle ::= \mathbf{MOD} S$$

(a) Syntax of `internal` keyword.

$$\text{A-INT} \frac{S = \left\{ s' \mid x_i \in \bar{x}_{0\dots n}, s \vdash_{\mathcal{G}} x_i \overset{\mathbf{M}}{\rightsquigarrow} s' \right\}}{s \vdash_{\mathcal{G}} \mathbf{internal}(\bar{x}_{0\dots n}) \Rightarrow \mathbf{MOD} S} \quad \text{AP-INT} \frac{\vdash_{\mathcal{G}} s \uparrow_{\mathbf{M}} S_M \quad s_m \in S_M \quad s_m \in S}{s \vdash_{\mathcal{G}} p ! \mathbf{MOD} S}$$

(b) Semantics of `internal` keyword.

$$\text{A-INT}' \frac{\overbrace{S = \left\{ s' \mid x_i \in \bar{x}_{0\dots n}, s \vdash_{\mathcal{G}} x_i \overset{\mathbf{M}}{\rightsquigarrow} s', s' \in S_M \right\}}^{\vdash_{\mathcal{G}} s \uparrow_{\mathbf{M}} S_M}}{s \vdash_{\mathcal{G}} \mathbf{internal}(\bar{x}_{0\dots n}) \Rightarrow \mathbf{MOD} S} \quad \text{AP-INT}' \frac{\overbrace{\vdash_{\mathcal{G}} s \uparrow_{\mathbf{M}} s_m}^{\vdash_{\mathcal{G}} s \uparrow_{\mathbf{M}} S_M} \quad s_m \in S}{s \vdash_{\mathcal{G}} p ! \mathbf{MOD} S}$$

(c) Variant 1: Ancestor module only.

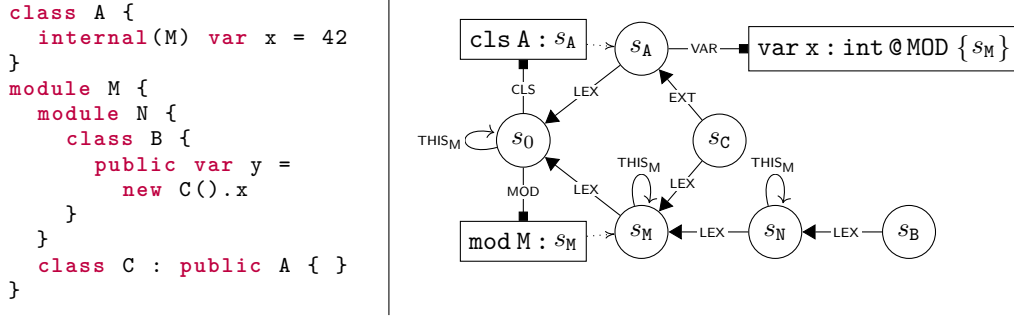
(d) Variant 2: Innermost module.

$$\text{AP-INT}'' \frac{\dots \left[\overbrace{\vdash_{\mathcal{G}} s \uparrow_{\mathbf{M}} S_{M_i} \quad s'_m \in S_{M_i} \quad s'_m \in S}_{s' \in (\text{scopes}(p) \setminus \{\text{tgt}(p)\})} \right]}{s \vdash_{\mathcal{G}} p ! \mathbf{MOD} S}$$

(e) Variant 3: Definition exposed to all classes in path.

■ **Figure 10** Extending AML (Figure 7) with module-level visibility.

47:14 Defining Name Accessibility Using Scope Graphs



(a) Example program and partial scope graph demonstrating the `internal` access modifier.

$$\text{A-INT} \frac{s_A \vdash_{\mathcal{G}} M \overset{M}{\rightsquigarrow} s_M}{s_A \vdash_{\mathcal{G}} \text{internal}(M) \Rightarrow \text{MOD} \{s_M\}}$$

(b) Part of typing derivation that shows how accessibility policy is derived.

$$\text{AP-INT} \frac{\dots \quad \vdash_{\mathcal{G}} s_B \hat{\uparrow}_M \{s_0, s_M, s_N\} \quad s_M \in \{s_0, s_M, s_N\} \quad s_M \in \{s_M\}}{s_B \vdash_{\mathcal{G}} (s_C \xrightarrow{\text{EXT}} s_A) ! \text{MOD} \{s_M\}}$$

(c) Part of typing derivation that shows how access is granted by the `MOD` accessibility policy.

■ **Figure 11** Example program demonstrating the meaning of the `internal` access modifier.

are included in the resulting policy. The **AP-INT** rule encodes that accessing an `internal` variable is valid if s_m , the scope of some enclosing module of s (the scope of the reference), is in the list of scopes to which access is granted.

Example. Figure 11 gives an example of an *internal* variable. Class `A` has a field `x` that can be accessed from module `M`. In the scope graph, this is indicated with the access policy `MOD {s_M}` on the corresponding declaration in s_A . The derivation of this policy is shown in Figure 11b. Module `M` contains a nested module `N`, which contains a class `B`. In class `B`, the field `x` is accessed on an instance of `A`. The (partial) typing derivation in Figure 11c shows this access is allowed by the **AP-INT** rule. The first premise asserts that s_0 , s_M and s_N are the enclosing modules of s_B . This can be seen in the scope graph, as those scopes are reachable via paths with regular expression $\text{LEX}^* \text{THIS}_M$ (Figure 9). As s_M occurs both in the enclosing modules and in the access policy, access is allowed.

Variant 1. Several variations on this scheme are conceivable. For example, languages can restrict the modules to which an `internal` modifier may expose a declaration. For example, Rust has the `pub(in <path>)` visibility modifier, similar to how we defined `internal`. However, at the $\langle \text{path} \rangle$ position, only “an ancestor module of the item whose visibility is being declared” is allowed [7, §12.6]. This is encoded in Figure 10c. Compared to **A-INT**, this rule adds premises (highlighted) that guarantee that the arguments of the `internal` modifier (x_i) resolve to an enclosing module ($s_i \in S_M$).

Note how these premises would make the example fail to type-check. Only s_0 is an enclosing module of s_A . In particular, the derivation in Figure 11b would have an additional premise $s_A \in \{s_0\}$, which is clearly unsatisfiable.

```

class A {
    private int x = 42;
    public int accessX(B b) {
        return b.x; // ERROR!
    }
}
class B extends A { }

class A {
    private int x = 42;
    public int AccessX(B b) {
        return b.x; // fine
    }
}
class B : A { }

```

■ **Figure 12** Difference in `private` member access of subclass instances between Java and C#.

Variante 2. Next, consider the example in Figure 11a again. In the system above, `x` is accessible in `B`, because `x` is exposed to one of its enclosing modules (M). However, s_M is not its *innermost* enclosing module. Such a more lenient accessibility scheme might be desirable (e.g., Rust has this behavior), but languages such as Java do not allow this. To model these languages, we instead use the premise that asserts s_m is the *innermost* enclosing module scope. The rule for this variante is given in Figure 10d.

With this addition, the example would fail to type-check as well. The access validation (Figure 11c) would now have to satisfy $\vdash_G s_B \uparrow_M s_M$, which is impossible, as s_M is the innermost enclosing module.

Variante 3. Finally, consider example Figure 1a from the introduction again. In this example, the reference to `x` in class `C` was not valid, as `B` (by virtue of residing in a different package), did not inherit `x`. The (partial) rule in Figure 10e covers this case. For each scope in the path (apart from the target), it adds premises that assert that the definition is exposed to that scope (similar to s in Figure 10b).⁵ The target is excluded because it is not inheriting the accessed field, but rather defining it. (Recall that paths move from reference to declaration, so the target is the scope of the defining class.) For that reason, there is no need to assert it inherits the field.

When adding this rule fragment to the derivation in Figure 11c, there will be additional premises that validate that class `C` inherits `x`. This is the case, as `C` resides in module M .

6 Defining Subclass Visibility

Next, we consider how to define access modifiers that regulate access from other *classes*: the `private` modifier (Section 6.1), and the `protected` keyword (Section 6.2).

6.1 Private Access

The `private` access modifier is slightly challenging to define, as languages implement it differently. For example, C# allows accessing private variables on instances of *subclasses*, whereas Java does not. Consider the example programs in Figure 12. In the Java case, the access `b.x` is invalid, because it only allows access on instances of `A`.

On the other hand, Java exposes `private` members to the *outermost* enclosing class⁶, while C# only exposes members to the *defining* (i.e., innermost enclosing) class (including nested classes), as shown in Figure 13.

⁵ Alternatively, the premises of Figure 10d can be used when direct exposure is required.

⁶ “[When] the member or constructor is declared `private`, (...) access is permitted if and only if it occurs within the body of the top level class [sic!] that encloses the declaration of the member or constructor.” [11, §6.6.1]

47:16 Defining Name Accessibility Using Scope Graphs

```

class A {
  class B {
    private int x = 42;
  }
  int accessX(B b) {
    return b.x; // fine
  }
}

class A {
  class B {
    private int x = 42;
  }
  int AccessX(B b) {
    return b.x; // ERROR!
  }
}

```

■ **Figure 13** Difference in `private` member access from enclosing class between Java and C#.

$$\langle acc \rangle ::= \dots \mid \text{private} \langle A \rangle ::= \dots \text{PRV} \qquad \text{A-PRIV} \frac{}{s \vdash_{\mathcal{G}} \text{private} \Rightarrow \text{PRV}}$$

(a) Syntax of `private` keyword.

(b) `private` to PRV access policy.

$$\text{AP-PRIV} \frac{\vdash_{\mathcal{G}} s \hat{\uparrow}_{\mathcal{C}} S_C \quad \text{tgt}(p) \in S_C}{s \vdash_{\mathcal{G}} p! A} \qquad \text{AP-PRIV}' \frac{\dots \quad p \sim \text{LEX}^*}{s \vdash_{\mathcal{G}} p! A}$$

(c) Semantics of `private` keyword.

(d) Prevent access on instances of subclasses.

$$\text{AP-PRIV}'' \frac{\vdash_{\mathcal{G}} s \hat{\uparrow}_{\mathcal{C}} S_{C_{ref}} \quad \vdash_{\mathcal{G}} \text{tgt}(p) \hat{\uparrow}_{\mathcal{C}} S_{C_{decl}} \quad s_c \in S_{C_{ref}} \quad s_c \in S_{C_{decl}}}{s \vdash_{\mathcal{G}} p! A}$$

(e) Allow access from enclosing classes.

■ **Figure 14** Extending AML (Figure 7) with `private` visibility.

We start with modeling the C# semantics in Figures 14a–14c. Rule **AP-PRIV** states that the class in which the field is declared (which is the target of the path $\text{tgt}(p)$) should be an enclosing class of the scope in which the access occurs. This permits access from nested classes of $\text{tgt}(p)$, but does not expose it to enclosing classes. On the other hand, access on instances of subclasses is allowed, as there are no constraints on the structure of the path.

Note that we did not specify that this rule matches on the PRV policy specifically, but rather applies to *any* access policy A . This is a deliberate choice; it adds the possibility of using this rule as a fallback in case no other rule works. This ensures other accessibility policies will never be more strict than PRV, which corresponds to general intuition. By matching on an arbitrary A in **AP-PRIV**, we simplify the definition of the other policies, as they otherwise would need to define special rules for `private`-like access.

Current Instance. Now, we adapt these rules to match the Java semantics. First, Figure 14d shows how to prevent access to the private field on instances of subclasses (Figure 12). It uses a new type of constraint, $p \sim R$, which holds when the sequence of labels in path p is in the language described by the regular expression R . In this case, we assert that the access path p must adhere to the regular expression LEX^* . This prevents access from instances of subclasses of the defining class, as that requires traversing an `EXT` edge. For example, the access path in Figure 12 would be $s_B \xrightarrow{\text{EXT}} s_A \sim \text{LEX}^*$, which is not satisfiable.

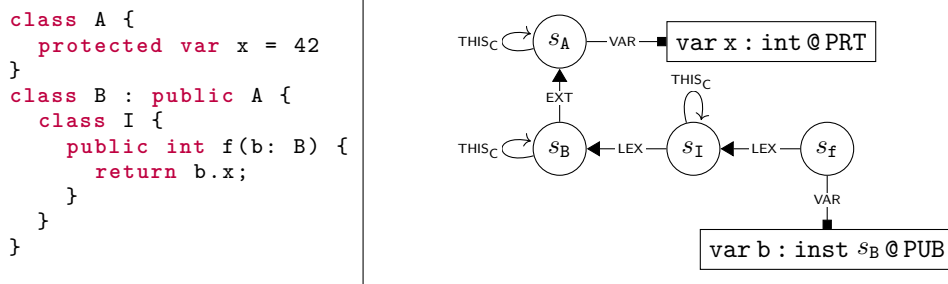
Outermost Class. Finally, Figure 14e shows how to expose `private` fields to the outermost enclosing class. In this rule, the set $S_{C_{ref}}$ contains the scope of the enclosing classes of the reference location, and $S_{C_{decl}}$ contains the scope of the enclosing classes of the class in which the declaration occurs. These sets should share a scope s_c , which represents the shared enclosing class of the reference and the declaration.

$$\langle acc \rangle ::= \dots \mid \text{protected} \qquad \langle A \rangle ::= \dots \mid \text{PRT}$$
(a) Syntax of `protected` keyword.

$$\frac{\text{A-PROT}}{s \vdash_G \text{protected} \Rightarrow \text{PRT}} \qquad \text{AP-PROT} \frac{\vdash_G s \hat{\uparrow}_C S_C \quad s_c \in S_C \quad s_c \in \text{scopes}(p)}{s \vdash_G p ! \text{PRT}}$$

(b) Semantics of `protected` keyword.

■ **Figure 15** Extending AML (Figure 7) with `protected` visibility.

(a) Example program and partial scope graph demonstrating the `protected` access modifier.

$$\frac{\dots \quad \vdash_G s_f \hat{\uparrow}_C \{s_I, s_B\} \quad s_B \in \{s_I, s_B\} \quad s_B \in \text{scopes}(s_B \xrightarrow{\text{EXT}} s_A)}{s_f \vdash_G (s_B \xrightarrow{\text{EXT}} s_A) ! \text{PRT}}$$

(b) Part of typing derivation that shows how access is granted by the PRT accessibility policy.

■ **Figure 16** Example program demonstrating the meaning of the `protected` access modifier.

Note how this rule enables type-checking the program in Figure 13. Using `AP-PRIV` does not work, as $\vdash_G s_A \hat{\uparrow}_C \{s_A\}$, which does not include $\text{tgt}(p) = s_B$. However, we can check it with `AP-PRIV`, as $\vdash_G \text{tgt}(p) \hat{\uparrow}_C \{s_B, s_A\}$, which includes the shared enclosing class s_A .

6.2 Protected Access

The `protected` access modifier (Figure 15a) grants access to subclasses of the defining class, including classes nested in subclasses. For field access expressions ($\langle e.x \rangle$), e must be an instance of a class that encloses *the reference* [11, §6.6.2.1]. This semantics (Figure 15b) can be modeled by asserting that there should be some class s_c that is both (a) an enclosing scope of the reference location ($\vdash_G s \hat{\uparrow}_C S_C$), and (b) occurs in the in the access path ($s_c \in \text{scopes}(p)$). The last condition implies that the enclosing class s_c is a subclass of the *defining class*, which is the intuitive understanding of the `protected` keyword.

Figure 16 demonstrates how this rule works. In this program, there is a class `A` which has a subclass `B`. Class `B` has a nested class `I`, which has a method `f` with a parameter `b` of type `B`. The body of `f` accesses field `x` on the instance of `B`. On the right-hand side of the picture, a part of the corresponding scope graph is shown. The scopes for classes `A` and `B` are connected by an `EXT`-edge again. The fact that class `I` is nested in class `B` is represented by the $s_I \xrightarrow{\text{LEX}} s_B$ edge, similar to other lexically nested constructs. Likewise, scope s_f , which represents the body of the method `f`, has a `LEX`-edge to s_I .

Figure 16b shows how the access to `b.x` is validated. The first premise states that s_I and s_B are the enclosing classes of s_f . The other premises assert that s_B is in the enclosing classes as well as in the access path. Together, this allows access to the protected member. Note how access to an instance of `A` in s_f would not be allowed. In that case, the access path would have been just s_A , which is not an enclosing class of s_f .

47:18 Defining Name Accessibility Using Scope Graphs

$\langle acc \rangle ::= \dots | \text{protected internal } (\langle x \rangle^*) | \text{private protected } (\langle x \rangle^*)$

$\langle A \rangle ::= \dots | \text{SMD } S | \text{SMC } S$

(a) Syntax of policy interaction keywords.

$$\text{A-PPROT} \frac{S = \left\{ s' \mid x_i \in \bar{x}_{0\dots n}, s \vdash_{\mathcal{G}} x_i \overset{M}{\rightsquigarrow} s' \right\}}{s \vdash_{\mathcal{G}} \text{private protected}(\bar{x}_{0\dots n}) \Rightarrow \text{SMC } S}$$

$$\text{A-PIINT} \frac{S = \left\{ s' \mid x_i \in \bar{x}_{0\dots n}, s \vdash_{\mathcal{G}} x_i \overset{M}{\rightsquigarrow} s' \right\}}{s \vdash_{\mathcal{G}} \text{protected internal}(\bar{x}_{0\dots n}) \Rightarrow \text{SMD } S}$$

(b) Translation of composite keywords to their policies.

$$\text{AP-SMC} \frac{s \vdash_{\mathcal{G}} p! \text{MOD } S \quad s \vdash_{\mathcal{G}} p! \text{PRT}}{s \vdash_{\mathcal{G}} p! \text{SMC } S}$$

$$\text{AP-SMD-PROT} \frac{s \vdash_{\mathcal{G}} p! \text{PRT}}{s \vdash_{\mathcal{G}} p! \text{SMD } S} \quad \text{AP-SMD-MOD} \frac{s \vdash_{\mathcal{G}} p! \text{MOD } S^{(*)}}{s \vdash_{\mathcal{G}} p! \text{SMD } S}$$

(c) Semantics of interaction policies.

■ **Figure 17** Extending AML (Figure 7) with keywords to combine module-level and subclass-level accessibility.

7 Combining Subclass and Module Visibility

Access modifiers regulating both the module and subclass dimensions occur in real-world languages as well. For example (as noticed earlier), Java’s **protected** keyword also exposes a definition in the same package, similar to C#’s **protected internal**. In addition, C# has a **private protected** modifier, which allows access to subclasses in the same assembly only. In fact, those two keywords denote the two main ways in which access modifiers can interact. First, **protected internal** denotes *disjunctive* interaction, where a declaration is accessible from the subclasses *or* the same module. Second, **private protected** denotes *conjunctive* interaction, where a declaration is accessible from the subclasses *in* the same module only. These interactions are straightforward to define, with one intricate case discussed below.

Figure 17a defines the syntax of the two new keywords (based on their name in C#) and policies. We add **SMD** (Subclass/Module, Disjunctive) and **SMC** (Subclass/Module, Conjunctive) policies, which each contain a list of module scopes to which they are exposed. The translation from keyword to policy is given in Figure 17b. Both rules resolve their module arguments, similar to **A-INT**. The **SMC** policy has one rule (**AP-SMC**), which simply asserts that access is granted by the module (**MOD**) and protected (**PRV**) policies. There are two rules for the **SMD** policy. The first one simply delegates to the **PRT** access policy, permitting access wherever a **protected** member would have been accessible. The other rule delegates to the **MOD** policy, but more careful attention must be paid here (hence the (*) mark). Recall that the semantics of this policy has a variant that asserts that the whole inheritance chain has access to the declaration (Figure 10e). However, this extension should *not* be applied here, because the **protected** part of this modifier already grants access, regardless of the module-level exposure.

$$\text{P-PUB} \frac{p \sim \text{LEX}^* \text{EXT}^*}{s \vdash_{\mathcal{G}} p \text{ i}} \quad \text{P-PRIV-PROT} \frac{\vdash_{\mathcal{G}} s \uparrow_C S_C \quad s_c \in S_C \quad \text{split-at}(s_c, p) = \langle p_1, p_2 \rangle}{p_1 \sim \text{LEX}^* \text{EXT}^* \quad p_2 \sim \text{EXT}_{\text{PRV}}^? (\text{EXT} | \text{EXT}_{\text{PRT}})^*} s \vdash_{\mathcal{G}} p \text{ i}$$

■ **Figure 18** Extending AML (Figure 7) with path-level visibility.

8 Defining Extends-Clause Accessibility Restriction

Until now, we have only considered inheritance as it exists in Java and C#. In this section, we shift our focus to C++, in particular the access modifiers on extends clauses. In C++, it is possible to add a `private` modifier to an extends clause, which reduces the accessibility of `public` and `protected` members to `private` in the derived class. Similarly, the `protected` keyword can be used to reduce the accessibility of `public` members to `protected`. For qualified accesses, C++ imposes the additional constraint that the inheritance chain leading to class in which the variable is declared should be accessible from the class in which the access occurs [8, §11.9.3 (4)].

Setup. In contrast to the previous sections, we cannot encode inheritance-imposed access control in our accessibility policy A . Instead, we encode it in the scope graph directly. For that purpose, we introduce two new labels: EXT_{PRV} and EXT_{PRT} , which model private and protected extension, respectively. Similar to the previous sections, EXT will model public extension; i.e. inheritance without access restriction.

Fortunately, we can validate path access independently from the declaration-level access policy.⁷ We require two adaptations to the rules **T-VAR** and **T-FLD** (Figure 7). First, the regular expressions of the queries must be changed to also traverse these new edges. Thus, in **T-VAR**, $\text{LEX}^* \text{EXT}^* \text{VAR}$ must be changed to $\text{LEX}^* (\text{EXT} | \text{EXT}_{\text{PRT}} | \text{EXT}_{\text{PRV}})^* \text{VAR}$. Similarly, **T-FLD** now has $(\text{EXT} | \text{EXT}_{\text{PRT}} | \text{EXT}_{\text{PRV}})^* \text{VAR}$ as regular expression instead of $\text{EXT}^* \text{VAR}$. Second, we add a premise $s \vdash_{\mathcal{G}} p \text{ i}$ to both rules. This premise asserts that the labels in the path p do not hide the accessed definition in scope s .

Path accessibility can be captured in two rules, shown in Figure 18. First, **P-PUB** asserts that a path is valid when there is only public inheritance. With this rule, the semantics of the programs that do not use private or protected inheritance has not changed. Second, rule **P-PRIV-PROT** covers the other two cases. This rule looks intricate, but the intuition behind it is not too complicated. Similar to the `private` and `protected` modifiers (Sections 6.1 and 6.2), access must occur within the class where the member is `private/protected`. This is now not necessarily the defining class, but rather the last class in the inheritance chain that has a non-public modifier on the extends clause. In the rule, this is encoded as follows. The first two premises introduce a scope s_c , which is an enclosing scope of the reference location s . The third premise asserts that the path p can be split into two segments at scope s_c . That is, p consists of two segments: a part p_1 from s_1 to s_c and a part p_2 from s_c to s_n . This implies that s_c is in the access path. To validate that all subclasses of s_c in the path have public inheritance, p_1 should match regular expression $\text{LEX}^* \text{EXT}$.⁸ The path leading from the current class to the declaration (p_2) may start with a private inheritance step ($\text{EXT}_{\text{PRV}}^?$), but may have only public and protected inheritance higher in the access path.

⁷ That also holds for the subtle interaction between `internal` and `protected` discussed in Section 7. `protected` or `private` inheritance in subclasses of the reference class can still compromise these access modes, and must therefore be validated.

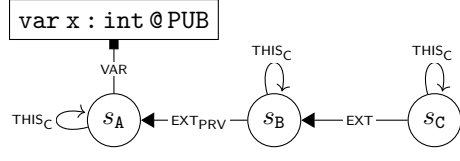
⁸ Alternatively, one can encode the requirement that the instance type must be s_c itself by using LEX^* , similar to Figure 14d.

47:20 Defining Name Accessibility Using Scope Graphs

```

class A {
  public var x = 42
}
class B : private A {
  public var y = new C().x
}
class C : public B { }

```



(a) Example program and partial scope graph demonstrating path access restrictions.

$$\frac{\dots}{\vdash_{\mathcal{G}} s_B \uparrow_C \{s_B\}} \quad s_B \in \{s_B\}$$

$$\text{split-at}(s_B, s_C \xrightarrow{\text{EXT}} s_B \xrightarrow{\text{EXT_PRV}} s_A) = \langle s_C \xrightarrow{\text{EXT}} s_B, s_B \xrightarrow{\text{EXT_PRV}} s_A \rangle$$

$$\frac{(s_C \xrightarrow{\text{EXT}} s_B) \sim \text{LEX}^* \text{EXT}^* \quad (s_B \xrightarrow{\text{EXT_PRV}} s_A) \sim \text{EXT_PRV}^? (\text{EXT} | \text{EXT_PRT})^*}{s_B \vdash_{\mathcal{G}} (s_C \xrightarrow{\text{EXT}} s_B \xrightarrow{\text{EXT_PRV}} s_A) \text{ i}}$$

(b) Part of typing derivation that shows how access is granted by the **P-PRIV-PROT** rule.

■ **Figure 19** Example program demonstrating path accessibility.

Figure 19 gives an example that uses this rule. There is a class `A` with a field `x`. Class `A` is inherited privately by class `B`, which makes `x` private in `B`. Next, class `C` extends `B` publicly. In class `B`, `x` is accessed on an instance of `C`. This access should be allowed, as class `B` is the class in which `x` is private as well as the class in which the reference occurs. The partial derivation in Figure 19b asserts this. s_B is the scope that encloses the reference. Splitting the access path from s_C to s_A at that s_B yields two segments of a single step. The segment leading up to s_B ($s_C \xrightarrow{\text{EXT}} s_B$) does indeed match the regular expression $\text{LEX}^* \text{EXT}^*$. Likewise, the other segment also matches its regular expressions, showing that this access is valid. Note that, when class `C` would have extended class `B` with `protected` or `private` visibility instead, the premise on the first section would not hold anymore. This corresponds with the behavior in Section 6 (the field must be accessible as if it was defined on the instance type) as well as the specification of `C++` cited above.

9 Analysis

A comprehensive model of accessibility can be made by composing the system fragments we discussed so far (Figures 7, 10, 14, 15, 17, and 18). In this section, we discuss a few properties that our system adheres to.

9.1 Soundness of Access Policies

First, we claim some soundness theorems for `private`, `protected` and `internal` access. There is no soundness theorem for `public`, as access is allowed unconditionally. Soundness theorems for `private` `protected` and `protected` `internal` are easily derived from Theorems 2 and 3, and hence omitted. In the theorems, P_G ranges over valid typing derivation for an AML program with scope graph \mathcal{G} , x_r over references, and x_d over declarations. Appendix D [35] defines the predicates used in these theorems, and proves them.

First, soundness for `private` access is stated as follows:

► **Theorem 1** (Soundness of `private` member access).

$$\begin{aligned} \text{resolve}_{P_G}(x_r) = x_d \wedge \text{private}_{P_G}(x_d) &\Rightarrow \\ \exists s_d. \text{definingClass}_{P_G}(x_d) = s_d \wedge \text{enclosingClass}_{P_G}(x_r, s_d) \end{aligned}$$

This should be read as “when x_r resolves to x_d , and x_d is `private`, then x_r must occur in the class s_c that defines x_d ”.

Likewise, soundness for `protected` access is stated as:

► **Theorem 2** (Soundness of `protected` member access).

$$\begin{aligned} \text{resolve}_{P_G}(x_r) = x_d \wedge \text{protected}_{P_G}(x_d) &\Rightarrow \\ \exists s_c, s_d. \text{definingClass}_{P_G}(x_d) = s_d \wedge \text{enclosingClass}_{P_G}(x_r, s_c) \wedge \text{subClass}_{P_G}(s_c, s_d) \end{aligned}$$

Compared to Theorem 1, this theorem states that x_r can occur in some arbitrary subclass s_c of s_d if x_d is `protected`.

Finally, `internal` access is specified correctly when:

► **Theorem 3** (Soundness of `internal` member access).

$$\begin{aligned} \text{resolve}_{P_G}(x_r) = x_d \wedge \text{internal}_{P_G}(x_d, \bar{x}) &\Rightarrow \\ (\exists x, s_m. \text{enclosingMod}_{P_G}(x_r) = s_m \wedge x \in \bar{x} \wedge \text{resolveMod}(x) = s_m) \vee \\ (\exists s_d. \text{definingClass}_{P_G}(x_d) = s_d \wedge \text{enclosingClass}_{P_G}(x_r, s_d)) \end{aligned}$$

This theorem states that references to declarations with modifier `internal` are valid if the enclosing module of the reference s_m is referred to in the arguments of the access modifier \bar{x} , or if it is accessed as a private variable.

9.2 Equivalence of Access Policies

The access policy language $\langle A \rangle$ we defined is not minimal. It is possible to define equivalent policies in multiple ways. To analyze that, we define equivalence of access policies as follows:

► **Definition 4** (Equivalence of Access Policies).

$$\frac{\forall \mathcal{G}, s, p. (s \vdash_{\mathcal{G}} p ! A) \Leftrightarrow (s \vdash_{\mathcal{G}} p ! A')}{A \equiv A'}$$

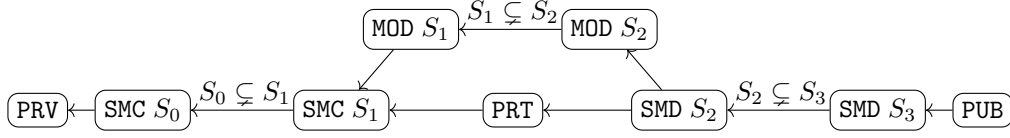
That is, two accessibility policies are equivalent when, for any scope s , path p , scope graph \mathcal{G} , either both policies admit access, or neither does.

The equivalences that hold in our model are: $\text{PRT} \equiv \text{SMD } \emptyset$ and $\text{PRV} \equiv \text{SMC } \emptyset \equiv \text{MOD } \emptyset$. This follows from the fact that module access grants nothing if no module parameters are given. Thus, the $\text{SMD } \emptyset$ policy reduces to PRT , while $\text{SMC } \emptyset$ and $\text{MOD } \emptyset$ do not elevate accessibility beyond PRV . Appendix B [35] gives proofs for each of these equivalences. Because of these equivalences, we did not include PRT and PRV in our implementation (Section 10).

9.3 Order of Access Policies

Intuitively, there exists an ordering between accessibility policies, where PRV is the bottom most restrictive, and PUB is the least restrictive. This order is partial, as the module-exposure dimension and subclass-exposure dimension are orthogonal. Assuming a subset relation on scope sets ($S \subset S'$), we can define a strict partial order $A <_A A'$ as follows:

47:22 Defining Name Accessibility Using Scope Graphs



where the edges indicate instances of the $<_A$ -relation. The edges with a condition indicate that SMC, MOD, and SMD become more permissive when more scopes are added to the policy.

The intuition behind this order is not arbitrary. In fact, we claim the following:

► **Theorem 5** (The order on access policies $<_A$ is well-behaved).

$$(A <_A A') \Rightarrow \forall \mathcal{G}, s, p. (s \vdash_{\mathcal{G}} p ! A) \Rightarrow (s \vdash_{\mathcal{G}} p ! A')$$

That is, when A is more restrictive than A' , and A permits access in scope s via a path p , then A' will permit that access too. A proof of this theorem can be found in Appendix C [35].

10 Evaluation

So far, we have motivated our specification with examples from real-world languages such as Java and C#, and stated some generic properties of our model. However, for our specification to be usable as a basis for practical tools, it must correspond with the behavior of the actual languages. To validate that, we evaluated our specification in two ways. First, we systematically compared our specification with reference compilers of Java, C#, and Rust. Second, we validated the compatibility of our framework with recent work on language-parametric code completion [19].

10.1 Comparison with Reference Compilers: Implementation

The comparison to compilers of real-world languages is implemented as follows:

1. Apply our type system on an AML program (the test case).
2. Translate the AML program to the target language.
3. Compile the translated program using a compiler of the target language.
4. Compare results: either both analyses should succeed, or both should give errors.

We discuss these steps in more detail below.

AML Type Checker. To compare our model with real-world compilers, we need a way to type check concrete AML programs. To that end, we implemented AML in the Spofax language workbench [13, 31]. The actual type system is implemented using the Statix specification language [27, 22]. Statix is a suitable choice, as its declarativity allows an overall straightforward translation from our inference. For example, the Statix encoding of rule **T-VAR** in Figure 20a strongly corresponds to the original (Figure 7). Using this implementation, we can systematically check accessibility in concrete AML programs.

Compiling with Reference Compiler. Next, we implemented source-to-source translations from AML to each of Java, C# and Rust. This translation was straightforward by design, as otherwise the results of the type checkers can be different due to semantic differences introduced by the translation. For that reason, we do not support AML features that have no direct counterpart in the target language. For example, the translation to Java will error when the AML program uses the `private protected` access modifier, as Java does not support that accessibility policy. This way, we know that correspondence between the programs is guaranteed when the translation succeeds.

<pre> typeOfExpr: scope * Expr -> TYPE typeOfExpr(s, Id(x)) = T :- {p A} query var filter LEX* EXT* and { x' :- x' == x } min \$ < EXT, EXT < LEX in s -> [(p, (x, T, A))], accessOk(s, p, A), pathOk(s, p). </pre>	<pre> test private - nested [[class A { private var x = 42 class B { public var y = x } }]] analysis succeeds run java-compat </pre>
<p>(a) Encoding of rule T-VAR in Statix.</p>	<p>(b) Example test case.</p>

■ **Figure 20** Overview of Approach to Comparison with Reference Compilers.

After translating, we invoke the reference compiler, observe its output (success or failure), and compare the given output with the result from our own type checker (step 1). If those are different (i.e., our type checker accepts the program, while the reference compiler emits errors, or vice versa), the test fails.

10.2 Comparison with Reference Compilers: Test Cases

To the best of our knowledge, there exists no test suite specifically aimed at verifying the semantics of access modifiers. For that reason, we manually created an extensive test suite. Each test contains a class **Def**, that defines some variable **x** with some access modifier **A**. Furthermore, each test contains a class **Ref**, in which a reference to **x** occurs. **Def** and **Ref** can be related in two different ways at the same time:

- By inheritance: either (1) **Def** and **Ref** are actually the same class, (2) have no mutual inheritance, (3) **Ref** inherits **Def**, or (4) **Def** inherits **Ref**.
- By module position: either **Def** and **Ref** (1) occur in the same module, or (2) **Ref** occurs in a parent/sibling/child module of **Def**.
- By class nesting: either (1) **Def** and **Ref** are top-level classes, (2) **Ref** is nested in **Def**, (3) **Def** is nested in **Ref**, or (4) **Def** and **Ref** have a shared enclosing class.

In addition, tests for member accesses (i.e., `recv.x`) have a receiver type **Recv**. This type must either be equal to **Def**, or inherit from it. However, it can be related in all possible other ways to **Def** and **Ref**. By systematically exploring all options, we derived our test suite.

We excluded cases that are (1) impossible (e.g., **Ref** cannot be nested in **Def** and live in a different module at the same time), (2) use features not supported by the target language, (3) invalid for another reason (i.e., inheriting from a nested class is not allowed by Java), or (4) do not bind properly (i.e., lexical access where **Ref** and **Def** do not inherit from each other, and are not nested in each other). To reduce the number of test cases, we restricted the cases that involved nested classes to have one module only. Additionally, we only used `private`, `protected` and `protected internal` as access modifiers in these cases. Table 2 summarizes the results of the test suite generation.

Figure 20b shows an example test case written in the Spoofox Testing Language (SPT) [12]. This test validates that a private field is accessible from a nested class. The test consists of a program (between double square brackets), and some *expectations*. In this case, we expect the (Statix-based) analysis to succeed. Moreover, we expect the `java-compat` transformation to succeed. This transformation is executing the steps in Section 10.1.

■ **Table 2** Summary of Test Suite.

	Java	C#	Rust	Manual
<i>Acc. Mods.</i>	<code>public</code> <code>protected internal</code>	Same as Java, and <code>protected</code> <code>private protected</code>	<code>public</code> <code>internal</code>	All
<i>Features</i>	class inheritance class nesting, and packages	class inheritance class nesting, and assemblies	structs, modules	advanced modules inheritance visibility
<i>#Cases</i>	433	522	60	168
<i>Compiler</i>	<code>javac</code> 11.0.20.1	<code>dotnet</code> 7.0.401	<code>rustc</code> 1.73.0	—

Results. There are several features present in AML that were not covered by any of the reference compilers, most notably `private/protected inheritance`, and module visibility beyond what Rust supports. To validate we cover these features to some extent, we have written 168 additional test cases. While initially exposing a lot of edge cases, in the end all test cases succeeded. This shows that our specification covers the languages it set out to model rather accurately.

10.3 Code Completion

One of our future goals is to use our framework to implement refactoring tools that are sound with respect to accessibility. The most recent work in this area is done by Pelsmaecker et al. [19]. They show how Statix specifications can be used to generate editor auto-completion proposals language-parametrically. We applied auto-completion to the access modifiers in the C#/Java and Rust tests (152, after deduplication), and validated soundness and completeness. That is, when the analysis succeeded, code completion should propose the current modifier at that position. Otherwise, if the access was invalid, the modifier should not be proposed, as only less restrictive ones are valid at that position.

We consider the fact that all completion tests pass a good indication that our specification can be applied with refactoring tools in the future. Apparently, the code completion framework is sound and complete with respect to our encoding of access modifiers. Accessibility errors introduced by a refactoring can be fixed by generating proposals for that position, and using the ordering from Section 9.3 to pick the most restrictive one.

10.4 Threats to Validity

In Section 4, we briefly mentioned that the specification as presented in the paper did not model the interaction between shadowing and accessibility correctly. Doing so would require a full path order, instead of ordering paths by a lexicographical order on labels. Appendix A.1 [35] explains how we think that could be done. However, Statix does not support full path orders. To work around that, we emulated this behavior using a few helper predicates. Our test suite gives confidence we modeled it correctly, but we did not prove that the specification in Statix and the full path order are semantically equivalent.

Finally, we might have modeled incorrect/unspecified behavior if the reference compilers were incorrect. Examples such as Figure 12 were derived from actual compiler behavior. However, we could not find our interpretation of the implementation behavior explicitly specified in the JLS [11, §6.6.1].

11 Related Work

In this section, we discuss previous work related to access modifiers and scope graphs.

11.1 Access Modifier Semantics and Implementations

The origin of access modifiers dates back to at least Simula 67, which around 1972 introduced `protected` and `hidden` access modifiers [4, §8] (the latter being equivalent to our `private`). Later, languages such as Java and C++ incorporated these keywords, making them well-known and often used. Design principles and patterns [9] using these keywords were developed, making contemporary software development heavily reliant on accessibility features the programming language provides.

Giurca and Savulea (2004) [10] apply object-oriented notions of `public`, `protected` and `private` to logic programs, with the purpose of better knowledge distribution and run time optimization. Moreover, Apel et al. [2, 1] introduce access modifiers in feature-oriented programming. Where we define accessibility for module nesting and class inheritance, they add the “feature refinement” dimension to this. In particular, the `feature` keyword restricts access to the “current feature” only (comparable to `private` in the class inheritance dimension), the `subsequent` keyword grants access to the current feature and later refinements (comparable to `protected`), and the `program` modifier allows access from any position (similar to `public`). In our terminology, their model supports “conjunctive” combination of the class and feature dimensions. As Section 7 shows that combining the module and class dimensions conjunctively is straightforward, we expect that integrating their work in our model will not pose major challenges (apart from a combinatorial explosion of policies).

Semantics. As access modifiers mainly originated from practical needs, it is not very surprising that little attention to them was paid from a more theoretical perspective. A few attempts to create a more formal account have been performed, however. In 1998, Yang [33] presented a formalization of Java access modifiers using attribute grammars. At that time, attribute grammars still lacked several convenience features, such as default attributes [30] and collection attributes [14]. For that reason, all members must be propagated explicitly to the scopes where they are accessible, which makes the specification rather verbose. Additionally, since fields and methods are not treated equally (shadowing vs. overriding), they are treated separately, doubling the specification size. In contrast, we specify the propagation of members queries in scope graphs, which is more concise. The additional requirements on methods (not explicitly discussed), can be handled at the definition site. Furthermore, we cover more features than just the Java ones. Fharkani et al. [6] present a generalized model of accessibility, where accessibility is modeled as a set of rules granting access of a member to another member (similar to Eiffel/`friends` in C++). In addition, rules can *deny* access to the named member, or apply to all members except the named ones.

Tools. Steimann et al. [25] observe that disregarding accessibility can result in a lot of subtle mistakes. For example, a method may silently fail to override another method when it is moved to a different package, which results in different dynamic dispatch. To capture these errors, they present nine constraint generation rules that model the accessibility semantics of Java. Refactoring tools can use these constraints to detect where the accessibility level of a member must be elevated. This work was incorporated in the JRRT refactoring tool [23], which was evaluated on a large number of real-world Java projects, showing the accuracy of their implementation. While their work also covers overriding-specific constraints, which our

specification treats rather superficially, we think our model is more comprehensible, and also gives insight in the differences between languages. Moreover, their work is applied in real refactoring implementations, while the quest for Statix-based refactorings is still ongoing. Meanwhile, a similar approach was applied to Eiffel accessibility [24, §6.3].

While these tools *elevate* accessibility if needed, a different line of research aims to *restrict* accessibility if possible [5, 17, 34]. The purpose of these tools is to detect access modifiers that are more lenient than needed, and restrict those. This is claimed to improve readability, enable optimizations, and increase modularity [17]. The exact underlying model is not the topic of these publications, and hence remains unclear. Despite that, the tools appear to be useful in practice. Zoller and Schmolitzky mention some challenges in porting their tool to other object-oriented languages [34, V.B]. A language-parametric model such as ours helps in that regard by (1) making differences between languages explicit, and (2) make implementations of these (kind of) tools language-parametric.

11.2 Scope Graphs

Scope Graphs (Section 3) have been introduced by Neron et al. [18], and later refined by Van Antwerpen et al. [27] and Rouvoet et al. [22]. In order to bridge the gap between language specification and implementation, scope graphs have been embedded into the NaBL2 constraint language [26]. Later, the Statix logic language was introduced [27, 22], which is more expressive than NaBL2. Both languages allow specifying type checking as constraint programs, giving the language a declarative appeal, but also yielding an executable type checker. Scope graphs are also available in a framework for concurrent and incremental type checkers [28, 39] and an embedded DSL in Haskell [20]. Finally, Statix specifications have been used for language-parametric code completion [19] and refactorings [16, 29]. Zwaan and Van Antwerpen provide a detailed overview of the development history of scope graphs, their embeddings in type system specification DSLs, and their applications [38].

12 Conclusion

Access modifiers occur in many real-world languages. To implement high-quality tooling for these languages, a good understanding of access modifiers is required. In this paper, we presented a model for access validation based on scope graphs. Our model covers the most important accessibility features in contemporary languages, including module accessibility, and inheritance accessibility, both on declarations and extends-clauses. Variations between different languages, both in supported features and their semantics, are made explicit in our model. Our specification is quite declarative, partly because scope graphs abstract over low-level name resolution and scoping details. Our model was validated using an extensive test suite, using Java, C#, and Rust compilers as oracles. This test suite was also used to show that we can synthesize access modifiers accurately using previous work on code completion [19].

Our main motivation for this work is twofold. First, we aim to provide a “language-transcendent” model for accessibility that enables comparison of different languages regarding accessibility. To this end we identify and formalize differences in the semantics of several access modifiers. In addition, we formulate soundness theorems of several access modifiers, and prove them. As such, we consider our specification accurate enough to serve as a reference for future tool implementations. Second, we aim to use our model in language-parametric refactorings, ensuring they respect accessibility properly. As these refactoring tools are still in development, actual validation of this application is still future work.

References

- 1 Sven Apel, Sergiy S. Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming*, 77(3):174–187, 2012. doi:10.1016/j.scico.2010.07.005.
- 2 Sven Apel, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. An orthogonal access modifier model for feature-oriented programming. In Sven Apel, William R. Cook, Krzysztof Czarnecki, Christian Kästner, Neil Loughran, and Oscar Nierstrasz, editors, *Proceedings of the First International Workshop on Feature-Oriented Software Development, FOSD 2009, Denver, Colorado, USA, October 6, 2009*, ACM International Conference Proceeding Series, pages 27–33. ACM, 2009. doi:10.1145/1629716.1629723.
- 3 Casper Bach Poulsen, Xulei Liu, and Luka Miljak. Towards a Language-parametric DSL for Refactoring (Short Paper), 2024. URL: https://popl24.sigplan.org/details?action-call-with-get-request-type=1&c9432bfaa61a48fb852237f9e99a821daction_1742650661080820307cb713fc2d28c30ae360b0bed=1&__ajax_runtime_request__=1&context=POPL-2024&track=pepm-2024&urlKey=8&decoTitle=Towards-a-Language-parametric-DSL-for-Refactoring-Short-Paper-
- 4 Andrew P. Black. Object-oriented programming: Some history, and challenges for the next fifty years. *Inf. Comput.*, 231:3–20, 2013. doi:10.1016/j.ic.2013.08.002.
- 5 Philipp Bouillon, Eric Großkinsky, and Friedrich Steimann. Controlling Accessibility in Agile Projects with the Access Modifier Modifier. In Richard F. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 41–59. Springer, 2008. doi:10.1007/978-3-540-69824-1_4.
- 6 Toktam Ramezani Farkhani, Mohammadreza Razzazi, and Peyman Teymoori. Eam: Expansive access modifiers in oop. In *2008 International Conference on Computer and Communication Engineering*, pages 589–594, 2008. doi:10.1109/ICCCE.2008.4580672.
- 7 The Rust Foundation. The Rust Reference. Accessed 25-09-2023. URL: <https://doc.rust-lang.org/reference/>.
- 8 The Standard C++ Foundation. Working Draft, Standard for Programming Language C++. Online version from <https://github.com/cplusplus/draft/releases/tag/n4868> was consulted. Per release notes, ‘only editorial changes compared to C++20’ were made.
- 9 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Oscar M. Nierstrasz, editor, *ECOOP’93 — Object-Oriented Programming*, pages 406–431, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/3-540-47910-4_21.
- 10 Adrian Giurca and Dorel Savulea. Logic programs with access modifiers. In *4th International Conference on Artificial Intelligence and Digital Communication, AIDC*, pages 22–31, 2004.
- 11 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification - Java SE 8 Edition, February 2015. URL: <https://docs.oracle.com/javase/specs/jls/se8/html/>.
- 12 Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: enabling test-driven language development. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 139–154. ACM, 2011. doi:10.1145/2048066.2048080.
- 13 Lennart C. L. Kats and Eelco Visser. The Spoofox language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869497.

- 14 Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. Extending Attribute Grammars with Collection Attributes—Evaluation and Applications. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0, 2007. doi:10.1109/SCAM.2007.13.
- 15 Luka Miljak, Casper Bach Poulsen, and Flip van Spaendonck. Verifying Well-Typedness Preservation of Refactorings using Scope Graphs. In Aaron Tomb, editor, *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023*, pages 44–50. ACM, 2023. doi:10.1145/3605156.3606455.
- 16 Phil Misteli. Renaming for Everyone: Language-parametric Renaming in Spoofox. Master’s thesis, Delft University of Technology, May 2021. URL: <http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5>.
- 17 Andreas Müller. Bytecode analysis for checking java access modifiers. In *Work in Progress and Poster Session, 8th Int. Conf. on Principles and Practice of Programming in Java (PPPJ 2010), Vienna, Austria*, pages 1–4, 2010.
- 18 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.
- 19 Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA):1–30, 2022. doi:10.1145/3527329.
- 20 Casper Bach Poulsen, Aron Zwaan, and Paul Hübner. A Monadic Framework for Name Resolution in Multi-phased Type Checkers. In Coen De Roover, Bernhard Rumpe, and Amir Shaikhha, editors, *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2023, Cascais, Portugal, October 22-23, 2023*, pages 14–28. ACM, 2023. doi:10.1145/3624007.3624051.
- 21 Mark Reinhold. Java platform module system, August 2017. URL: <https://jcp.org/en/jsr/detail?id=376>.
- 22 Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428248.
- 23 Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Trans. Software Eng.*, 38(6):1233–1257, 2012. doi:10.1109/TSE.2012.13.
- 24 Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A Refactoring Constraint Language and Its Application to Eiffel. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 255–280. Springer, 2011. doi:10.1007/978-3-642-22655-7_13.
- 25 Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 2009. doi:10.1007/978-3-642-03013-0_19.
- 26 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

- 27 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.
- 28 Hendrik van Antwerpen and Eelco Visser. Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPICs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ECOOP.2021.1.
- 29 Loek Van der Gugten. Function Inlining as a Language Parametric Refactoring. Master’s thesis, Delft University of Technology, June 2022. URL: <http://resolver.tudelft.nl/uuid:15057a42-f049-4321-b9ee-f62e7f1fda9f>.
- 30 Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2002. doi:10.1007/3-540-45937-5_11.
- 31 Guido Wachsmuth, Gabriël Konat, and Eelco Visser. Language Design with the Spoofox Language Workbench. *IEEE Software*, 31(5):35–43, 2014. doi:10.1109/MS.2014.100.
- 32 Bill Wagner, Manuel Zelenka, and Youssef Victor. C# Reference — Keywords — file, November 2022. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/file>.
- 33 Wu Yang. Discovering anomalies in access modifiers in java with a formal specification, 1998. URL: <http://dspace.fcu.edu.tw/bitstream/2377/2120/1/ce07ics001998000164.pdf>.
- 34 Christian Zoller and Axel Schmolitzky. Measuring Inappropriate Generosity with Access Modifiers in Java Systems. In *2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement, Assisi, Italy, October 17-19, 2012*, pages 43–52. IEEE Computer Society, 2012. doi:10.1109/IWSM-MENSURA.2012.15.
- 35 Aron Zwaan and Casper Bach Poulsen. Defining Name Accessibility using Scope Graphs (Artifact), May 2024. doi:10.5281/zenodo.11179594.
- 36 Aron Zwaan and Casper Bach Poulsen. Defining Name Accessibility using Scope Graphs (Extended Edition). *CoRR*, May 2024. doi:10.48550/arXiv.2407.09320.
- 37 Aron Zwaan and Casper Bach Poulsen. Defining Name Accessibility using Scope Graphs (Artifact). Software (visited on 2024-08-05). URL: <https://zenodo.org/records/11179594>.
- 38 Aron Zwaan and Hendrik van Antwerpen. Scope Graphs: The Story so Far. In Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann, editors, *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*, volume 109 of *OASICs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/OASICs.EVCS.2023.32.
- 39 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: using scope graphs to derive incremental type-checkers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):424–448, 2022. doi:10.1145/3563303.