Cyclic Name Resolution for Scope Graphs using Circular Attribute Grammars

Luke Bessant
bessa028@umn.edu

Abstract

Name binding, associating references with appropriate declarations, is an integral part of programming language analysis. However, no uniform model for this process has been widely adopted - many language designers instead implement ad-hoc approaches for their specific needs. Recognizing this, Eelco Visser and his colleagues proposed scope graphs to fill this role. The scope graphs model represents the binding structure of programs as a graph, and provides a generic means for resolving names. A challenge in this model is scheduling the construction and interrogation of the scope graph, which may grow during a program analysis, influencing the results of later name resolution in it.

Work with attribute grammars and Statix has shown that they can be used to interleave these actions, supporting features like type-dependent name resolution. Demand-driven evaluation of attribute grammars results in such a schedule being discovered on-demand, while Statix interprets scope graph-based program analysis as a constraint-solving problem, and uses a notion of missing critical edges to schedule name resolution. Both approaches are comparable in that the results of their separate analyses of a program, for instance determining if a program as type-correct, coincide. Importantly, both implementations fail under *self-influencing* name resolution semantics; when one result of a name lookup can influence the other results of the same lookup circularly. Such behavior is erroneous in conventional attribute grammars, so specifications using self-influencing name resolution are impossible. Similarly, in Statix, constraint solving becomes stuck. In this work we develop a solution using fixed-point computation of name resolution that can support self-influencing imports as used by Rust, and apply these ideas using circular attribute grammars.

1 Introduction

Binding references to their correct declarations is an integral part of the analysis of a program. For instance, type checking of references requires knowledge of the type of the corresponding declaration of that name, and implementing module imports requires a resolution of that import to the correct module. Despite the importance of this analysis, name binding has often been treated in an ad-hoc way. Language designers tend to define name binding structures and lookup operations to suit their own needs, so no widely used generic model exists. This greatly inhibits the intelligibility and portability of these systems. As such, the programming languages community would benefit from a model generic and expressive enough to support most name analyses.

```
1   pub mod foo {
2     pub mod bar {
3       pub static x:u8 = 1;
4     }
5   }
6   pub mod test {
7     use super::*;
8     use bar::*;
9     use foo::*;
10    pub static z:u8 = y+1;
11    pub static y:u8 = x;
12  }
```

(a)

```
1   pub mod foo {
2     pub mod foo {
3       pub static x:u8 = 1;
4     }
5   }
6   pub mod test {
7     use super::*;
8     use foo::*;
9     pub static y:u8 = x;
10
11  }
```

(b)

Figure 1. Two Rust programs. The left correctly resolves each `use` declaration. The right is ambiguous on `foo`.

The *scope graphs* model developed by Visser et al. [16, 24] is a promising candidate to fill this role. Under this approach, the name binding structure of a program is represented as a graph in which nodes represent the program's scopes and declarations. Scopes are regions of a program where name resolution acts uniformly. That is, identical name references in the same scope will resolve to the same declaration. An example of a program scope is the body of the Rust module `test` declared on line 6 of Figure 1a, as well as the bodies of `foo` on line 1 and `bar` on line 2. The Rust examples shown were are modified versions of programs defined by Rouvoet et al. [18], which we have extended with variable declarations and references. are used here for their interesting name resolution behavior. Names declared in a Rust module are visible in definitions of other names before and after it in the same module - *e.g.* `y` on line 11 of Figure 1a is visible in the definition of `z` on line 10. Declared names such as `x` on line 3 of Figure 1a are represented in a scope graph by declaration nodes. The name of a declaration is associated with its declaration node by a mapping of scope graph nodes to strings, used during name resolution. Certain program constructs can be represented by a node that acts as both a scope, i.e. where we look up names, as well as a declaration. Rust modules like `foo` declared on line 1, `bar` on line 2, and `test` on line 6 would each be represented by a node that is a declaration of the module name, as well as the lookup scope of references in the module. Directed edges in a scope graph represent syntactic or semantic relationships between scopes and declarations in the input program. Each edge has a label taken from a language-specific set, but some common examples of these are `LEX` for lexical parenthood and `VAR` for variable declarations. A scope graph for Figure 1a could use both of these; *e.g.* a `LEX` edge from the node representing module `bar` on line 2 to that of `foo` on line 1 would encode that the latter is the lexical parent of the former. `VAR` edges would associate variable declarations such as `y` on line 11 and `z` on line 10 of Figure 1a with their module scope.

Name analysis is performed by *queries* - walks in the scope graph from the scope containing a reference, such as `y` on line 10 of Figure 1a, to matching declarations. Queries walk the structure of the scope graph, following edges and computing paths from their start scope to declaration nodes whose name matches the reference being resolved. The result of a query is a set of paths in the scope graph from the start scope to declaration nodes whose name matches the reference to resolve. The results of queries can be used to *extend* the scope graph with new edges during the course of program analysis. For instance, resolving `foo` in the `use` declaration on line 9 of Figure 1a might result in an import (`IMP`) edge being added to the scope graph from the node representing module `test`, to that of `foo`. This edge would be used to resolve reference `bar` on line 8 to `bar` on line 2, after which we add another import edge used to resolve `x` on line 11. The resolution of variable references in an importing scope such as `test` in Figure 1a may not find the intended declarations unless the import resolutions occur first, and corresponding edges have been added to the graph that are followed when resolving a variable reference. This highlights a key challenge in the scope graphs model; *interleaving* name resolution with extensions to the graph.

The scope graphs model has been implemented in previous work on Statix [18, 21], as well as in attribute grammars [2]. Statix is a constraint-solving system for defining name/type analysis of programming languages. Statix ensures that name resolution and extensions to the scope graph are scheduled correctly by only executing a query once it is determined that the scope graph contains all edges that the query could ever follow, even those that may not lead to matching declarations. For instance, the resolution of `x` on line 11 of Figure 1a may only begin once all of the import edges that come from resolving the `use` declarations on lines 7–9. The attribute grammar approach instead uses *demand-driven evaluation* in such a way that we can begin the execution of any query. Along the way the query *demands* the unbuilt edges it requires, pausing to wait for their construction, which may entail executing other queries to compute the targets of those edges, before continuing. I.e. the attribute grammar will begin the query of reference `x` on line 11 of Figure 1a and discover along the way that there are missing import edges from the node representing module `test`. At this point, the query of `x` pauses to wait for those of `super`, `bar` and `foo` on lines 7–9, after which the missing import edges are added and the query of `x` continues. Our previous work [2] has shown that these two approaches produce the same results for a given language specification and input program, being equivalent in their expressiveness.

Importantly, both approaches are limited by their inability to support a particular class of import semantics - those that are *self-influencing* - namely, when one result in the set returned of a particular query is used by other results of the same query in a circular way. An example of a self-influencing name resolution semantics is that of Rust `use` declarations. For example, the `use foo::*` declaration line 8 of the program in Figure 1b, which contains two nested `foo` modules on lines 1–2. The Rust compiler determines this reference is ambiguous because its resolution to the outer module `foo` makes available another resolution to the inner one. In scope graph terms, one result of a query of `foo` on line 8 is to the outer `foo` on line 1, yielding an import edge that allows the same query to produce another result to the inner `foo` on line 2. In Statix or conventional attribute grammars approach it is not possible to define queries that behave in this way. In Statix, constraint solving becomes stuck because missing edges required to begin an import query are not build-able until the query, which determines the targets of those edges, is complete. In attribute grammars, self-influencing imports involve circular definitions which cannot be evaluated conventionally.

Circular attribute grammars [5, 10] extend conventional attribute grammars [12] with *circular attributes*, which are able to compute cyclic definitions using a fixed-point computation. These have been used in the past for other circular programming language analyses such as liveness analysis [5, 10, 13] and computing first/follow sets for parsing [13]. We show that they can be used to resolve self-influencing imports through a computation which iteratively grows the result set of an import query, using edges introduced by the resolution results of one iteration to find new resolutions in the next. This opens the door to more exotic name binding semantics, such as those of Rust imports that we have described, as well as others we explore.

We begin by introducing the theory of scope graphs in Section 2, and show that two different scope graphs for a given input program encode different name resolution semantics for a toy language LM defined by Neron et al. [16]. Section 3 details the implementation of scope graphs in conventional attribute grammars, following our previous work [2]. Here we detail how demand-driven evaluation is used to schedule the interleaving of scope graph construction and name analysis. We develop a fixed-point computation in Section 4 that implements self-influencing imports, using as examples two distinct import semantics inspired by our exploration of Rust. We also introduce a *coherence* property which says that all edges used in all name resolutions must appear in the final scope graph. Our attribute grammar definition of scope graphs in Section 3 is then extended in Section 5, where the fixed-point computation developed in Section 4 is implemented using circular attributes. A discussion of our work with self-influencing imports occurs in Section 6. We detail related work in Section 7, then conclude and identify interesting future work in Section 8.

2 Scope Graphs and Name Resolution

In this section we expand on the scope graphs model we have described; what a scope graph is comprised of, and how to do name analysis in them. The example program we use in Figure 2a is written in LM, a toy language defined by Neron et al. [16], which is flexible enough for us to "plug in" different import semantics such as those similar to Rust that we will see in Section 4. The syntax of LM includes variable and modules definitions, as well as imports, each of which should be self-explanatory. We see in Figure 2a an LM program with module declarations on $\mathtt{A}_1$ on line 1, $\mathtt{B}_1$ on line 3, $\mathtt{B}_2$ on line 8, $\mathtt{A}_2$ on line 10, and $\mathtt{C}_1$ on line 15. Variable definitions are present in these modules, such as $\mathtt{y}_1$ on line 18. Note that all names here have a numeric subscript. These are there for us to refer to specific instances of a name and are not part of LM syntax. The example provided in Figure 2a is interesting in that results of the imports on lines 16–17 are unique under *sequential* and *parallel* imports we discuss in this section, as well as under the two self-influencing import semantics we explore in Section 4. An import semantics is *sequential* if the result of an import declaration can influence only name resolutions which syntactically follow it in the same syntactic block. *e.g.* in Figure 2a, the result of importing $\mathtt{B}_3$ can influence the resolution of $\mathtt{A}_3$, but not the other way around. An import semantics is *parallel* if the result of an import can influence only import resolutions which occur in nested syntactic blocks. Neither of the imports on lines 16–17 of Figure 2a can influence the other under parallel imports, under which all imports in a module are resolved separately and simultaneously. We show the final scope graphs for our program under sequential and parallel imports in Figure 2b and Figure 2c, and discuss these throughout this section.

2.1 Scope graphs

As described in the introduction, scope graphs are graph representations of the binding structure of programs. Nodes in them represent declarations in a program, and its scope regions. Directed edges between nodes describe relationships defined by their language-specific edge labels. The edge labels are taken from a language-specific label set $\mathcal{L}$ defined by the language implementor. The scope graphs in Figure 2b and Figure 2c use `LEX` edges for lexical parenthood, `IMP` edges for imports, `VAR` edges for variable declarations, and `MOD` edges for module declarations. For example, the `LEX` edge from S_2 to the global scope S_1 in Figure 2a encodes that S_1 is its lexical parent. The `MOD` edge from S_1 to S_2 in the same graph says that module $\mathtt{A}_1$ is defined in the global scope. We associate with each declaration its name, *e.g.* node $\mathtt{S}_5$ in Figure 2b and $\mathtt{S}_5$ in Figure 2c both map to the program name $\mathtt{x}_1$ on line 2 of Figure 2a.

Definition 2.1. A *scope graph* with label set $\mathcal{L}$ is a triple $G = \langle S_G, E_G, \rho_G \rangle$ comprised of a set of nodes S_G, edges $(s, l, s') \in E_G$ with $s, s' \in S_G$, $l \in \mathcal{L}$, and $\rho_G : S_G \rightharpoonup String$, a partial mapping of nodes to names.

The structure of a scope graph helps to define the desired name resolution semantics for a language, influencing the set of possible paths usable when resolving names. We use as examples the two distinct scope graphs in Figure 2b and Figure 2c, both representing the LM program in Figure 2a. The name resolution and edge extension actions we describe here are explained in Section 2.2 and Section 2.3.

Figure 2b gives the scope graph for Figure 2a under sequential import semantics. As well as nodes corresponding to all module and variable declarations, this scope graph involves a distinct scope node S_i for every statement in the program. For instance there are three scope nodes S_{15}, S_{20} and S_{22} under S_{10} because module $\mathtt{C}_1$ in Figure 2a contains three statements - the two imports on lines 16–17, and variable $\mathtt{y}_1$ on line 18. The pattern is that the scope paired with each statement has as its lexical parent, encoded by `LEX` edges, the scope node of the previous. *e.g.* the `LEX` edge from S_{15} to S_{10} in Figure 2b. Names in each successive statement are then resolved in their statement scope, allowing extensions to scopes above to influence their result. The `IMP` edges in a module are applied to the statement scopes sequentially, so that an imports can only influence the resolution of names that occur after it. This process is discussed in Section 2.3. We focus particularly on sequential imports in the following subsections.

Figure 2c gives the scope graph under parallel import semantics for the program in Figure 2a. Unlike for sequential imports, we do not construct a new scope for every statement in the program. Instead, all variable name resolutions happen from the scope of their module. For example, resolving $\mathtt{x}_5$ on line 18 of Figure 2a starts in scope S_4 of Figure 2c. Imports are resolved in the *lexical parent* of their enclosing module scope, but the resulting `IMP` edges are applied to the module scope. *e.g.*, import references $\mathtt{B}_3$ on line 16 and $\mathtt{A}_3$ on line 17 of Figure 2a are both resolved from scope S_1, but their `IMP` edges are added to scope S_4. This is so that import resolutions in a module do not influence one another when considering the LM query regular expression we describe in Section 2.2, and only influence the resolution of variable references in the same module or any names in nested modules.

2.2 Name resolution

In the scope graphs model, a *name resolution*, denoted $r \mapsto_G p$, associates a reference r with a path p from the scope of the reference to declarations in the graph [18]. A *program resolution* for a scope graph G is a set $pr_G = \{r_1 \mapsto_G$

```
1  module A₁ {
2    def x₁ = 1
3    module B₁ {
4      def x₂ = 2
5    }
6  }
7
8  module B₂ {
9    def x₃ = 2
10   module A₂ {
11     def x₄ = 2
12   }
13 }
14
15 module C₁ {
16   import B₃
17   import A₃
18   def y₁ = x₅
19 }
```

(a)

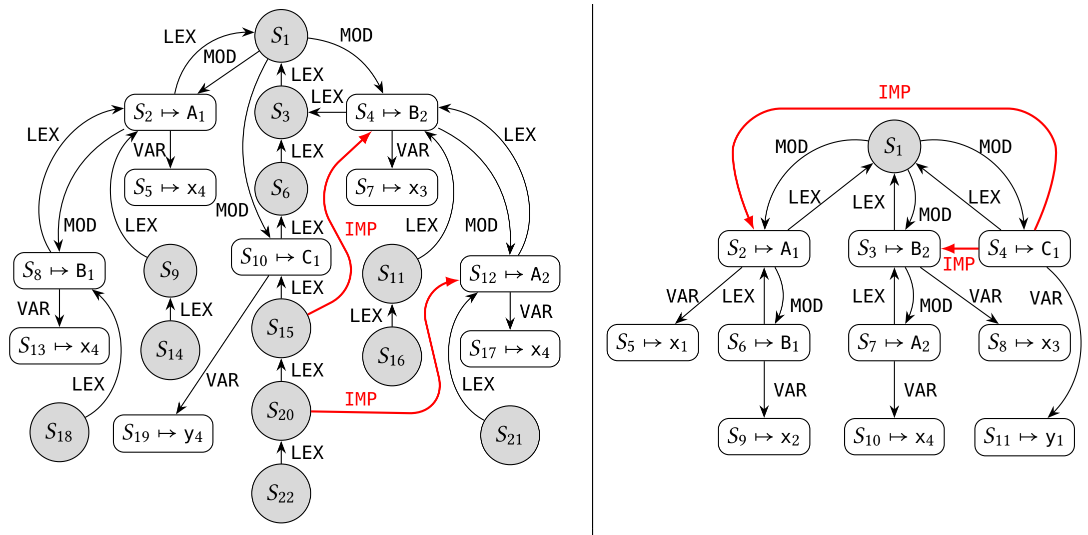

Figure 2. An LM program and its complete scope graph under sequential (left) and parallel (right) import semantics.

$p_1, ..., r_n \mapsto_G p_n\}$ of resolutions for references in an input program. We denote a path p in a graph G from scope s to s' as $G \vdash p : s \xrightarrow{w} s'$, with w being the word formed by concatenating the labels of the edges in the path. For instance, $G \vdash S_{15} \xrightarrow{\texttt{IMP}} S_4 \xrightarrow{\texttt{MOD}} S_{12} : S_{15} \xrightarrow{\texttt{IMP MOD}} S_{12}$ is a valid path when G is the scope graph in Figure 2b. Paths such as these are returned by *queries*, traversals of the scope graph from a start scope to declarations matching the name being resolved.

Definition 2.2. A *query* in scope graph G with label set $\mathcal{L}$ is a function of type $S_G \times R_{\mathcal{L}} \times String \to P$, taking a starting scope node, a path well-formedness regular expression, and a name to look for, yielding a set of resolution paths such that $p \in query(s, rx, \text{“}x\text{”})$ when $G \vdash p : s \xrightarrow{w} s'$, $w \in L(rx)$, and $\rho_G(s') == \text{“}x\text{”}$.

A path in a scope graph belongs to the result set of a query for “x” when its combined labels form a word in the language of the query regular expression, and the name of its node is “x”. In LM we use regular expression `LEX* IMP? MOD` for import resolution, and `LEX* IMP? VAR` for variables. That is, a valid resolution path is comprised of zero or more `LEX` edges, followed by an optional `IMP` edge, and ending in a `MOD` edge for imports or a `VAR` edge for variables. $S_{22} \xrightarrow{\texttt{LEX}} S_{20} \xrightarrow{\texttt{IMP}} S_{12} \xrightarrow{\texttt{VAR}} S_{17}$ is one valid resolution path for resolving reference x_5 on line 18 of Figure 2a to x_4 under the scope graph in Figure 2b. Another follows the `IMP` edge from S_{15} to find x_3 in S_7.

We mentioned that queries return a set of paths in the scope graph from the start scope to all matching declarations. These paths are all of the paths to declarations that are *reachable* in the graph. Any declaration at the end of a valid path with respect to the query regular expression is reachable. It is up to the language designer to reduce that set to only the declarations that are *visible* - i.e. those that shadow others with respect to a language-specific partial ordering on labels. This involves defining an ordering over the edge label set, and then picking minimal paths from a query result set lexicographically. For LM, we use ordering `MOD` = `VAR` < `IMP` < `LEX`. I.e., we prefer `IMP` over `LEX` edges, and prefer `MOD` and `VAR` over others. Accordingly, the resolution of x_5 on line 18 of Figure 2a to x_4 in Figure 2b shadows the one to x_3, because the first edge of the former is an `IMP` edge whereas the latter starts with a `LEX`.

2.3 Extensions to scope graphs

The results of queries can be used to *extend* the scope graph with new edges. For instance in Figure 2b, after the resolution of import B_3 the scope graph was extended with the `IMP` edge shown from S_{15} to S_4. This edge was then used in resolving A_3, leading to the addition of the `IMP` edge from S_{20} to S_{12}. These edges are then used to resolve x_5 in our example. We say that a path p returned by a query *yields* an edge from s to s' with label l, written $G \vdash p \triangleright s \xrightarrow{l} s'$, when s' is identified as the target of path p. The scope graph can be extended during the course of a program analysis, but a given query can only take into account the current state of the scope graph when it is executed. This presents us with the challenge of *scheduling* our extension and name analysis operations so that a query result is *stable* - a property identified by Rouvoet et al. [18]. They say that the results of a query are stable when they are equal to those that would be returned if we had executed the same query on the final scope graph. This property ensures that name resolution is sound with respect the target language's desired name resolution semantics.

In Figure 2b, $\mathtt{x}_5$ is resolved from scope S_{22}. If we were to resolve this name before having resolved the imports on lines 16–17, and adding the `IMP` edges they yield to scopes S_9 and S_{10}, we would not find any matching declarations. On the other hand if we were to resolve only one of these imports, say $\mathtt{B}_3$ from $\mathtt{C}_1$, resolution of $\mathtt{x}_5$ would only find declaration $\mathtt{x}_3$ by following the `IMP` edge from S_{15}. Neither of these results is stable, since the later addition of `IMP` edges to the scope graph changes the result of the query. Thus we must only resolve $\mathtt{x}_5$ after both imports are resolved, and their `IMP` edges added. Resolving $\mathtt{x}_5$ then correctly results in two valid resolution paths, an ambiguity. Similarly, both `IMP` edges must be added to scope S_4 in Figure 2c by resolving the imports on lines 16–17 of Figure 2a, before we resolve $\mathtt{x}_5$.

Ensuring stability of name resolution using scope graph has been the focus of Rouvoet et al. [18] and our previous work [2] which builds on the former. We discuss in Section 3 how we ensure stability of name resolution in an attribute grammar implementation of scope graphs.

3 Attribute Grammars for Scope Graphs

In this section we define attribute grammars, using as examples our implementation of scope graphs and specification of sequential and parallel import semantics for LM. We then describe how name resolution proceeds in our attribute grammar implementation. In particular, how demand-driven evaluation solves the scheduling problem and results in stable name resolution.

3.1 Attribute Grammars

An attribute grammar [12] is a context-free grammar augmented with semantic attributes, and equations defining them, which compute information for a syntax tree. Attribute grammars are generally used for defining both the syntax of programming languages and their semantic properties.

Definition 3.1. An attribute grammar is a 4-tuple $\langle G = \langle NT,\ T,\ P,\ S \in NT \rangle,\ A,\ O,\ EQ \rangle$ consisting of a context-free grammar G, attributes A, attribute occurrence relation $O \subseteq NT \times A$ and equations EQ defining attributes in A, each equation being associated with a production in P.

The context-free grammar G defines the syntactic structure of a language, including nonterminal symbols NT, other symbol types T such as Booleans and strings, productions P and start symbol S. Our focus is on *abstract* syntax. That is, a representation that abstracts away the terminal symbols used in *concrete* syntax. We see in Figure 3 an attribute grammar defining the abstract syntax of LM. Nonterminals are declared with the `nt` keyword, such as `Prog` on line 5, which we take to be our start symbol. Productions have the form `prod` $p : n_0 : NT_0 ::= x_1 : X_1, ..., x_j : X_j \mid nta_1 : NT_1, ..., nta_k : NT_k$, defining a name p for the production, node name n_0

```
1  inh attr s:Scope, si:Scope;
2  syn attr ok:Bool, ty:Ty,
3           s_IMP:[Scope], s_VAR:[Scope], s_MOD:[Scope];
4
5  nt Prog with ok;
6  prod root: top:Prog ::= ds:Stmts | sg:Scope {
7    sg = mkScope(); sg.LEX = []; sg.IMP = ds.s_IMP;
8    sg.VAR = ds.s_VAR; sg.MOD = ds.s_MOD;
9    ds.s = sg; ds.si = sg;
10   top.ok = ds.ok; }
11
12 nt Stmts with ok, s, si, s_MOD, s_VAR, s_IMP;
13 prod cons: top:Stmts ::= d:Stmt ds:Stmts {
14   d.s = top.s; ds.s = top.s;
15   d.si = top.si; ds.si = top.si;
16   top.ok = d.ok && ds.ok;
17   top.s_IMP = d.s_IMP ++ ds.s_IMP;
18   top.s_MOD = d.s_MOD ++ ds.s_MOD;
19   top.s_VAR = d.s_VAR ++ ds.s_VAR;  }
20 prod nil: top:Stmts ::= {
21   top.ok = true;
22   top.s_MOD = []; top.s_VAR = []; top.s_IMP = []; }
23
24 nt Stmt with ok, s, si, s_MOD, s_VAR, s_IMP;
25 prod mod: top:Stmt ::= x:String ds:Stmts | sm:Scope {
26   sm = mkScopeDcl(x);
27   sm.LEX = [top.s]; sm.IMP = ds.s_IMP;
28   sm.VAR = ds.s_VAR; sm.MOD = ds.s_MOD;
29   ds.s = sm; ds.si = top.s;
30   top.ok = ds.ok; top.s_MOD = [sm];
31   top.s_VAR = []; top.s_IMP = []; }
32 prod imp: top:Stmt ::= i:String {
33   local ps:[Path] = query(top.si, LEX* IMP? MOD, i);
34   local ps':[Path] = min(ps, VAR = MOD < IMP < LEX);
35   local p:Path = hd(ps'); local s_tgt:Scope = tgt(p);
36   top.s_IMP = [s_tgt]; top.ok = length(ps') == 1;
37   top.s_MOD = []; top.s_VAR = []; }
38 prod var: top:Stmt ::= x:String e:Expr | sv:Scope {
39   sv = mkScopeDcl(x);
40   sv.LEX = []; sv.VAR = []; sv.MOD = []; sv.IMP = [];
41   e.s = top.s;
42   top.s_VAR = [sv]; top.s_MOD = []; top.s_IMP = [];
43   top.ok = e.ok; }
```

Figure 3. An attribute grammar definition of LM declarations under parallel imports.

used by equations associated with this production for referring to the node it builds (we use `top`), nonterminal type NT_0, as well as the names and types of children. The children before the vertical bar (`|`) are understood as regular production children corresponding to the LM syntax. Those after the `|` are discussed shortly, as well as the bodies of these productions, i.e. between the `{...}`. The single production for nonterminal `Prog` is defined on line 6: The productions and nonterminals in an attribute grammar define how to construct trees that represent the object language. For example, to construct a tree of nonterminal type `Prog` we require a child tree whose root is a node of nonterminal `Stmt`. Each node in such a tree has a unique identifier that we use in the definition of attributes in a tree.

Attributes in A occur on the nodes in a syntax tree as specified by the occurrence relation O. Examples of attribute definitions in Figure 3 are on lines 1–3, each with a name and a type. The occurrence relation O is defined in Figure 3 by uses of the `with` keyword. By line 12, all tree nodes of nonterminal `Stmt` must have instances of attributes `ok`, `s`, `si`, `s_MOD`, `s_VAR` and `s_IMP`. Equations define the values of these attribute instances. Equations are associated with productions in Figure 3 by being defined in the bodies of those productions, i.e. between the `{...}`. For example each tree node built by the `root` production on line 6 has its attributes defined by the equations on lines 7–10. Attributes associated with nonterminals in O are either *inherited* or *synthesized*. The values of synthesized (`syn`) attribute instances on a node are defined by equations associated that node. *e.g.*, the equation defining `top.ok` on line 10 defines the `ok` attribute on the current node, referred to locally as `top`. The values of inherited (`inh`) attribute instances on a node are instead defined by equations on its parent node. *e.g.* the equations on line 9 of Figure 3 define inherited attributes `s` and `si` of child `ds`. Individual productions may also declare *local* attributes, these are only definable by and can only be referred to in equations in the same production. Examples of these are on lines 33–35 of Figure 3.

In classical attribute grammars [12], all attributes have primitive types. For instance, `Bool`, `String`, or `[String]` for a list of strings. Later advancements introduced *higher-order attribute grammars* [22], which allow us to construct new trees at runtime of the same kind as the program syntax tree, but separated from it or as extensions to it. New trees are defined by *nonterminal attributes*. These are associated with particular productions, *e.g.* in Figure 3 by their declaration after the `|`. These new trees can have attributes such as inherited ones that are defined in the production that built them, just as those of regular children are. Definitions of nonterminal attributes, such as that of `sg` on line 7 of Figure 3, use productions in the attribute grammar to build these trees. The production used in this definition is `mkScope`, specified in Figure 4 which we discuss in Section 3.3. The nonterminal attribute `sg` is treated as a child in production `root`, just as `ds` is. Another advancement to attribute grammars are reference attributes [4]. These allow us to store references to tree nodes as attributes. These references can then be communicated around a tree by definition of inherited and synthesized attributes. In Figure 3, `s`, `si`, `s_IMP`, `s_VAR` and `s_MOD` are all reference attributes, and their values will be (or are lists that contain) trees. We pass a reference to the `Scope` tree `sg`, created as a nonterminal attribute in production `root` on line 7, to the child `ds` by defining `ds`'s inherited `s` and `si` attributes as `sg`.

3.2 Evaluation of attribute grammars

The task of an attribute grammar evaluator is to find an order in which to evaluate the equations that define attributes in the tree. Some of these equations cannot be evaluated until those which define other attributes have been evaluated first. That is, the equation, and thereby the attribute it defines, *depends on* the values of other attributes defined by other equations. For instance, the equation defining `top.ok` on line 10 for a `root` node clearly depends on the value of child `ds`'s `ok` attribute instance. Therefore `top.ok` has `ds.ok` as a dependency, meaning the equation defining the latter must be evaluated first. We can identify some dependencies statically by looking at the attribute grammar specification. For instance, the dependency of a `root` node's `ok` on child `ds`'s `ok` attribute value we described. However in general we cannot compute all dependencies ahead of time.

The most prevalent approach to finding such a schedule to evaluate equations is in a *demand-driven* way [9]. Under demand-driven attribute grammar evaluation, the dependencies of attributes determine a schedule for executing equations. That is, if when evaluating the equation defining a particular attribute we discover a dependency on another attribute, then we pause to compute the latter before coming back to the former, substituting in the resulting value of the attribute depended on. Computation of the attributes in a syntax tree begins by demanding a synthesized attribute from its root node. That node's attributes may depend on the values of others around the tree, which in turn have their own dependencies. For LM, our initial demand will be for the `ok` attribute of that node, as we discuss in Section 3.3. The computation of the initial attribute involves following dependencies transitively, which takes our evaluation around the program's syntax tree. Importantly, the equation defining an attribute instance is only ever evaluated if that attribute is demanded by another at runtime, and attributes that are never demanded are never computed. When an attribute's defining expression has become a value, the attribute is marked as completed and the value stored in the tree. Then if the same attribute is demanded later, the value is immediately available. I.e., each attribute instance is only computed at most one time. The demand-driven behavior we describe here is fundamental to how we schedule the construction and querying of scope graphs, described in Section 3.4.

3.3 Scope Graphs for LM in Attribute Grammars

Our approach to implementing scope graphs in attribute grammars is to represent the nodes in a scope graph as trees constructed as nonterminal attributes. In our implementation scopes in the graph are always single-node trees, whereas declarations have a child that is their name. Our representation of the collection of a nodes in a scope graph is thereby as a forest of such trees. Reference attributes occur on the scope trees which denote the edges in a scope graph. Each

```
1  inh attr LEX:[Scope], IMP:[Scope],
2            VAR:[Scope], MOD:[Scope];
3  syn attr name:String;
4
5  nt Scope with LEX, IMP, VAR, MOD, name;
6  prod mkScope: top:Scope ::= {
7    top.name = "";  }
8  prod mkScopeDcl: top:Scope ::= id::String {
9    top.name = id;  }
```

Figure 4. Scope graphs in attribute grammars.

edge label used in the graph is represented by an attribute whose type is a list of `Scope` references, similar to attributes `s_VAR`, `s_MOD` and `s_IMP` in Figure 3. Each `Scope` tree node then has instances of these attributes, defined as the list of references to scopes that are the targets of corresponding edges sourced from that scope.

We define in Figure 4 the nodes of a scope graph with a nonterminal `Scope`, and the edges from a scope as attributes occurring on that nonterminal. The edge attributes `LEX`, `IMP`, `VAR` and `MOD`, defined on lines 1–2 represent the LM edges seen in Figure 2. These all have type `[Scope]`, i.e. each instance of one of these attributes is a list of scopes that are the targets of corresponding edges in the scope graph. We define these as inherited attributes, which is important in scheduling scope graph construction and name analysis under eager evaluation. If we instead defined them as synthesized attributes and provided to a scope these edges as arguments to a production such as `mkScope` on line 6, an attribute grammar which eagerly evaluates expressions would fully evaluate the edge lists given as arguments before building the scope node itself. As we discuss in Section 3.4, we only evaluate the edges of a scope graph during the execution of queries. We also define a synthesized attribute `name` as a String which occurs on scopes, for use in name resolution. The two productions in Figure 4 define two ways to create new scopes. We use the `mkScope` production for scope graph nodes that are merely scoping regions in a program, and `mkScopeDcl` for declarations, taking a name as an argument. Both productions have one equation which defines the synthesized `name` attribute, the empty string used for regular scopes.

Scope and declaration nodes of an LM scope graph are constructed as nonterminal attributes in the productions of the LM grammar in Figure 3. Figure 3 shows the declaration of nonterminal attribute for the global scope on line 6 and its definition on line 7 using the production `mkScope` from Figure 4. I.e. `sg` is a new tree of nonterminal type `Scope` with no children, constructed by use of production `mkScope`. Similarly, module scopes are defined in `mod` on line 26, and a variable declaration scopes on line 39, instead using the `mkScopeDcl` production on line 8 of Figure 4. The module and variable declaration productions both pass their declaration name to the `mkScopeDcl` production. The synthesized attribute `name` on these declaration nodes is then defined on line 9 as the same string. The inherited edge attributes for a scope must be defined in the same production that the scope is built in. We see definitions of these attributes for each nonterminal attribute defined in Figure 3. For instance, the attribute definitions on lines 7–8 define the edges of the global scope.

Edge attributes are populated by references to scope nodes built in various places around a program's syntax tree. In LM, under the name resolution semantics we focus on, references to variable and module declaration scope graph nodes are passed up the tree as synthesized attributes before becoming part of the, `VAR` and `MOD` attribute of their enclosing module scope. This is because declarations in modules are always lower in the syntax tree than the module's own declaration. Often the definition of a synthesized attribute instance on a node is simply the combination of the same attribute on that node's children. In Figure 3 we use synthesized attributes `s_VAR`, `s_MOD` and `s_IMP`, defined on line 3 for this purpose. These are typed as `[Scope]`, just as the inherited edge attributes in Figure 4 are. References to lexical parent scopes, the targets of `LEX` edges, are instead passed down the tree as inherited attributes, since lexical parent scopes are always defined higher up in the tree than references to them. We use inherited attributes `s` and `si`, defined on line 1, to communicate lexical parent scopes, the targets of `LEX` edges, for parallel imports. These scopes are copied down to the attributes of module declarations by the inherited attribute equations on lines 14–15. `s` is the scope of the enclosing module with which member declarations are associated by `VAR` and `MOD` edges, and in which variable references are resolved. For example, this `s` attribute's value for the LM tree nodes corresponding to statements on lines 16–18 of Figure 2a is a reference to scope $\mathtt{C}_1$ in Figure 2c. `si` is the lexically enclosing scope of the module, in which we resolve all import references in that module. For the imports on lines 16–17 of Figure 2a, this is is the global scope S_1. We see an example of a `LEX` edges definition on line 27 of Figure 3, defining it as a singleton list of the inherited attribute `s`. We synthesize the targets of `VAR`, `MOD` and `IMP` edges as lists, but the targets of `LEX` edges as single `Scope` references `s` and `si`. This is because we do not know ahead of time how many variables, modules or imports will be defined in a module when writing the attribute grammar, but we do know that only two lexical scopes are required to implement LM with parallel imports.

3.4 Name Resolution in Attribute Grammars

We implement queries as functions taking as arguments the query source scope, its path well-formedness regular expression, and the name to resolve. Inspired by Zwaan et al. [23] edges in the scope graph are followed during a query based on a DFA representation of the query regular expression over edge labels. A transition table defining a DFA of the LM import query regular expression `LEX* IMP?`

MOD, with start state st_1, intermediate state st_2 and accepting state st_3 is as follows:

From	Transition	To
st_1	– LEX →	st_1
	– IMP →	st_2
	– MOD →	st_3
st_2	– MOD →	st_3

As we follow an edge during a query, we make the corresponding transition in the DFA. The transitions possible from the new state tell us what edges can now be followed in the graph to other scopes. When we reach the accepting state of the DFA, we know that the current scope is a declaration and can test for name equality. References to declaration scopes whose name matches are collected and combined together into one list returned by the query. In Figure 3 we show one example of a query function application on line 33 for resolving an import reference.

Demand-driven evaluation, as we described in Section 3.2, is a fundamental part of our approach to scheduling the construction and interrogation of scope graphs [2]. By using demand-driven evaluation in our implementation of scope graphs, we can ensure that the nodes and edges of a scope graph are not constructed until they are demanded during name resolution. In our implementation of LM, query functions are the only computation which demand that the edge attributes and scope definitions are evaluated. Initially a query such as the one on line 33 of Figure 3 depends on its source scope being built, which may have occurred already if another query was executed from the same scope. Our DFA-based name resolution then, starting in state st_1 described above, demands edge attributes on that scope, corresponding to valid transitions in the DFA. We use as an example the query of x_5 using the scope graph in Figure 2c that encodes parallel import semantics for the program in Figure 2a. This query starts in scope S_4 and has regular expression `LEX* IMP? MOD`. It first demands the LEX, IMP and then MOD edges from scope S_4 - these are all valid transitions from st_1 in our DFA. Demanding the LEX edges for S_4 causes the single node tree representing the global scope S_1 to be built, by the dependencies of that attribute. This involves evaluating the nonterminal attribute definition of S_1, specified on line 7. When S_4.LEX is computed, a continuation of the query begins at scope S_1 and state st_1 in the DFA described above. The edge attributes are demanded from this scope in the same way, building new scopes and edges by evaluating reference and nonterminal attributes to extend the scope graph.

Of particular importance is that the IMP attribute on scope S_4 depends on import queries in that module. That is, each member of this attribute comes from executing the import query specified on line 33 of Figure 3, the results of which are used in the definition of s_IMP on line 36. Demanding S_4.IMP during the query of x_5 thereby causes our current name resolution to pause and wait for the result of those import queries. The computation of S_4.IMP draws into the scope graph the two IMP edges from S_4 to S_2 and S_3 in the scope graph in Figure 2c. The query function for x_5 then continues at these module scopes after transitioning in the DFA to state st_2, from which the only transition is to follow a VAR edge. In this way the construction of a scope graph and queries in it are scheduled by the demand-driven evaluation of our attribute grammar. Namely, queries demand the construction of scopes and edges that may in turn depend on other queries that are completed before the original. This approach ensures that name resolution is stable, a property we discussed in Section 2, since queries can only be completed once all of the attributes which correspond to edges they can follow have been demanded.

4 Self-influencing Imports

We consider in this section import queries that are *self-influencing* and thereby circularly defined. The two case studies we use to present our ideas are *recursive* as well as *unordered* imports, both simplifications of the Rust import semantics we described previously. In our definition, the structure of the scope graph for these semantics, other than IMP edges, is the same as for parallel imports. What distinguishes the two is how imports are resolved, and thus what the set of IMP edges is. The idea is to develop a general fixed-point computation that finds a set of *candidate* imports in a process that is the same for both recursive and unordered imports, and then filter this set with a post-processing function that distinguishes one semantics from the other.

4.1 Self-influencing imports

A query is *self-influencing* when a path in its result set makes use of the edge yielded by another resolution in the same set. Self-influencing import queries are therefore those whose resulting IMP edges are used by that same query in producing new paths. This behavior is captured by the right-hand side Rust program in Figure 1, where the resolution of foo to the outer module named foo is used to find a new resolution to the inner one. Rust determines here that foo is ambiguous, since the self-influencing resolution behavior means that both module declarations are discovered, and the resolution to either module does not shadow the other.

Definition 4.1. A query $q_G(s, r, "x")$ and result ps in scope graph G is *self-influencing* if $\exists p, p' \in ps \,.\, G \vdash p \triangleright e \;\wedge\; e \in p'$.

That is, a query is *self-influencing* when there is a path p in its resolution set that yields an edge e that are used in another path p' in the same result set. For the two import semantics explored in this section, recursive and unordered, all edges yielded by self-influencing import queries are sourced at the start scope of the query. Thus e will be the first edge in p'.

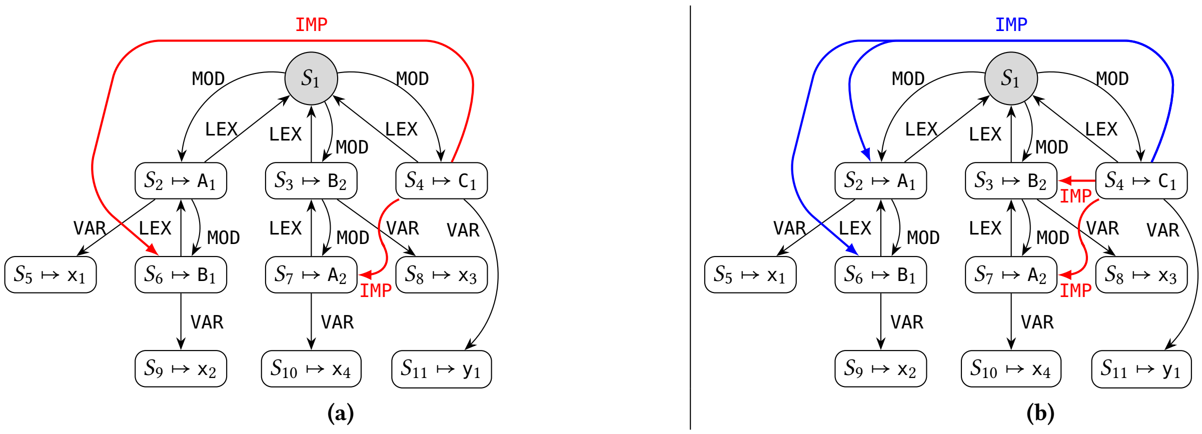

Figure 5. Scope graphs for program in Figure 2a under recursive (a) and unordered (b) import semantics

4.2 A fixed-point approach

Self-influencing imports exhibit circular behavior, given paths in the result set of a query may yield new edges that are used in other paths in the same result set. Problems that exhibit circular dependencies are often solvable using a *fixed-point computation* [20]. These are computations which interpret cyclic computation in an iterated, acyclic way. We are interested in *least* fixed-point computation in particular. That is, the iterative process starts with an initial minimal value, and each iteration successively contributes more to the result of the computation, until no new results are found by the final iteration. We have then reached a *fixed-point* where successive iterations can not grow the result any more. Our focus is on the fixed-point computation of *sets*, since we will compute a set of paths yielding candidate import edges for a particular query. Accordingly, we take the initial value for a fixed-point computation to be $\varnothing$. A fixed-point computation f with k iterations is characterized by an application $f_k(f_{k-1}(\ldots(f_1(\varnothing))\ldots))$ where each f_i for $i > 1$ is the application of computation f with the result of $f_{i-1}(\ldots)$ as its input. For a set c, it must be the case that $f_{i-1}(c) \subseteq f_i(f_{i-1}(c))$. Specifically, if i is the final iteration we have $f_{i-1}(c) = f_i(f_{i-1}(c))$ where a fixed-point has been found, otherwise $f_{i-1}(c) \subset f_i(f_{i-1}(c))$ with $f_i(f_{i-1}(c))$ contributing new elements to the growing set. This ensures that the result set monotonically grows with each iteration. That is, the result of every iteration is a superset of that of the previous.

The problem of self-influencing imports lends itself to fixed-point computation, since there is a clear ordering of the paths that result from a self-influencing import query based on the import edges they yield. As identified in Definition 4.1, a path that results from a self-influencing import query can use the edges yielded by others. We denote this notion of use of one path by another in graph G as $p \prec p'$ iff $G \vdash p \triangleright e$ and $e \in p'$. The resolution of self-influencing queries in a module as can be framed as a fixed-point computation. If each iteration executes the import queries in a particular module, then the `IMP` edges yielded by paths contributed by iteration f_{i-1} can be used by f_i to find more resolutions. Since the imports of a particular module can influence each other, as well as themselves, this computation comprises all of the import queries in a module. All import queries in a module therefore contribute results to the fixed-point computation simultaneously. Unlike the computation f described previously, there are two inputs to this computation. These are the scope graph to resolve in, and the monotonically growing set of `IMP` edges from the query start scope. We split these into two arguments because the `IMP` edges computed during this process may not all appear in the final scope graph - some may be filtered out in the post-processing step we discuss in Section 4.3.

An application of our computation f is $f(G, c)$ for some scope graph G and set of `IMP` edges c, defined as $\varnothing$ in the first iteration. The scope graph itself does not grow during the computation, but is a necessary in discovering resolution paths. Our computation is such that each iteration finds new import resolution paths by following the `IMP` edges in the set computed by the previous. That is, $p \prec p'$ when p was contributed to the growing set of import edges by an earlier iteration than p'. Our use of $\varnothing$ as the initial set of `IMP` edges from the source scope ensures that the first iteration must execute the import queries by first following other edges from that scope. Import edges from other scopes can be followed, however, if the query regular expression allows as is the case in LM with query regular expression `LEX* IMP? MOD`. This process stops when the `IMP` edges from the final iteration do not lead to new resolutions.

We can think of the fixed-point computation of imports in a module as a strongly connected component whose vertices are the import queries in that module, and whose edges describe influence. Namely, each import can influence the resolution of the other as well as itself. Our definition of self-influencing imports in LM is such that there is a distinct strongly connected component for every module. This is because although a query may use the `IMP` edges of its lexical parent scopes, the reverse is not true. Thus the import queries in our strongly connected components may be influenced by resolutions that occur in lexically enclosing modules, but may not influence them. We will exploit this fact in Section 5 when implementing the ideas discussed here.

For the LM program in Figure 2 and either scope graph in Figure 5 without the `IMP` edges, the imports B_3 and A_3 are resolved from the scope of module S_4 find resolutions B_2 and A_1 in the first iteration of the fixed-point computation both by following a `LEX` edge followed by `MOD`, since there are currently no `IMP` edges sourced from the scope of S_4. We then contribute references to those discovered modules to growing the set of candidate `IMP` edges from scope S_4, resulting in candidate `IMP` edges to S_3 and S_2. In the second iteration, the `IMP` edges yielded by the first are followed to find the inner modules B_1 at S_6 and A_2 at S_7. The third iteration finds no new resolutions after following these candidate `IMP` edges. The fixed-point computation therefore finishes, and the resulting import edges point to both `A` and both `B` modules. The following representation of this result set highlights the iterated behavior of our fixed-point computation:

$$
\begin{aligned}
&\varnothing \\
&\cup\ \{p_1 : S_4 \xrightarrow{LEX} S_1 \xrightarrow{MOD} S_3,\ p_2 : S_4 \xrightarrow{LEX} S_1 \xrightarrow{MOD} S_2\} \\
&\cup\ \{p_3 : S_4 \xrightarrow{IMP} S_3 \xrightarrow{MOD} S_7,\ p_4 : S_4 \xrightarrow{IMP} S_2 \xrightarrow{MOD} S_6\}
\end{aligned}
$$

The initial set as input to the first iteration is empty, the input to the second is the empty set unioned with the new resolutions of the first, and so on. In terms of our use relation we have that $p_1 \prec p_3$ and $p_2 \prec p_4$.

4.3 The post-processing step

In general, this process for computing self-influencing imports will return *all* matching module declarations that are found during its iterations. This may be a large set of paths, and will likely not reflect the intended import semantics. To that end, we use a post-processing step which involves filtering the set of `IMP` edges to only those that satisfy the desired semantics. To support this behavior, we first re-define the `IMP` edges yielded by the fixed-point computation as *candidate* import edges, for which we use a label `IMPc`. The `IMPc` edges populate the second argument of our computation f, that may only be followed during their own fixed-point computation, i.e. only by the import resolution queries that comprise it. `IMP` edges of lexically enclosing scopes can be followed during this computation, but those are *persistent* edges in G, the first argument to f. I.e., those that will appear in the final scope graph. This ensures that each module in the program is associated with a single strongly connected component that is computed independently of others. The filtering step can then occur per-module. The persistent edges for each scope are the result of filtering the `IMPc` edges to a smaller set of edges which are then re-drawn as `IMP` edges. The definition of self-influencing import behavior then consists of a generic fixed-point computation of candidate import resolutions `IMPc`, followed by a language-specific filtering step which computes all of the persistent `IMP` edges - those which encode the desired import semantics. As examples of this filtering step we focus on two examples of self-influencing import semantics, *recursive* and *unordered*.

Recursive imports are characterized by a behavior where import references resolve to the modules found by the longest resolution paths, with respect to replacement of `IMP` edges by the paths that yielded them. For example we resolve the two imports on lines 16–17 of Figure 2a to the two inner modules, as shown in Figure 5, because the paths to these are longer when replacing their `IMP` edges with the paths to the outer modules that yielded them. In terms of our fixed-point computation, these are resolutions whose paths yield edges used by no other paths in the set of candidate resolutions - i.e. the resolutions contributed by the final iteration. More formally, given a set of candidate paths ps, under recursive imports the set of paths which should yield persistent `IMP` edges is $ps_{rec} = \{p \in ps \mid \nexists p' \in ps\ .\ G \vdash p \triangleright e \land e \in p'\}$. For the imports in module C_1 of Figure 2, must then be the inner modules - B_1 and A_2. This final resolution is represented in Figure 5b, where the scope graph encodes recursive imports with `IMP` edges to the last resolutions found, to the inner `A` and `B` modules.

Unordered imports are another example of a semantics, distinct from recursive, that we can compute with our fixed-point computation. Under unordered imports, the syntactic order of imports in a module does not determine the order in which they are resolved, and each import may influence the resolution of others in that module but not itself. The aim is to find an order with which all of the imports in a module can be resolved if they were interpreted sequentially. In Figure 5 there are two such orders whose resulting `IMP` edges are grouped by color. First, resolving B_3 before A_3 means that modules B_2 at scope S_3 and A_2 at S_7 are imported, since the resolution of A_3 uses the `IMP` edge yielded by the resolution of B_3 to B_2. Alternatively, resolving A_3 before B_3 instead leads to the `IMP` edges to A_1 at scope S_2 and B_1 at S_6. In this example both orders can be used to resolve the imports, and neither has paths that are more minimal than the other, with respect to the LM label ordering. This means that the imports in module C_1 are ambiguous. We leave it to the language designer to determine what effect this has on their program analysis. We simply make persistent in Figure 5 the edges yielded by both orderings. A post-processing function supporting unordered imports entails, for each ordering of imports in a module picking the minimal, with respect to label ordering, *coherent* resolution path for each import reference in sequence.

Definition 4.2. A name resolution $r \mapsto_G p$ is *coherent* with respect to a program resolution pr_G, iff $\forall e = (s_i \xrightarrow{IMP} s_j) \in p$, $\exists (i \mapsto_G p') \in pr_G$ such that $G \vdash p' \triangleright e$.

Coherence requires that every `IMP` edge used in a path to resolve a name is persistent. Note that recursive imports do not need to satisfy coherence, since a module's persistent `IMP` edges may have been resolved using candidate edges which were not made persistent. On the other hand non-self-influencing import resolution such as we saw in Section 4 is always coherent because there are never non-persistent `IMP` edges in the scope graph. The process for computing the persistent `IMP` edges for unordered imports is as follows: For every permutation of an ordering of import references in a scope, we find the minimal (with respect to the language-specific edge label ordering) candidate resolution for each import reference, in sequence, that is *coherent* with respect to the resolutions of the previous references in that permutation. This means that the first import reference in each permutation is resolved through a path that uses no candidate `IMP` edges, and therefore must have been contributed by the first iteration of our fixed-point computation. The next may use `IMP` edges yielded by a paths in that import's resolution, and so on. When ordering B_3 before A_3 in Figure 5, B_3 is resolved using path $\mathsf{C}_1 \xrightarrow{\mathsf{LEX}} \mathsf{G} \xrightarrow{\mathsf{MOD}} \mathsf{B}_2$, then the resolution of A_3 uses the `IMP` edge yielded by this path, in $\mathsf{C}_1 \xrightarrow{\mathsf{IMP}} \mathsf{B}_2 \xrightarrow{\mathsf{MOD}} \mathsf{A}_2$.

The set of import resolution paths yielding persistent `IMP` edges for a particular scope under unordered imports, given candidate imports edges yielded by paths $ps = \{p_1, p_2, ..., p_n\}$, and ordered permutations of all import references in that scope os, is $ps_{uno} = \bigcup_{o \in os} \{ps_o \subseteq ps \mid o = r_1 < ... < r_k \wedge \forall r_i, r_j \in o, \exists p_i, p_j \in ps_o \,.\, (r_i \mapsto_G p_i \wedge r_j \mapsto_G p_j \wedge G \vdash p_j \triangleright e \wedge i < j \Rightarrow e \notin p_i)\}$. This is the union of sets of import resolution paths p_i is picked in a coherent way, for each ordering o of import references r_i. For every import reference in an ordering, the resolution path for it may only use edges yielded by resolution paths for references before in the ordering.

To summarize, our approach is one in which a set of *candidate* import resolutions are discovered by a generic fixed-point computation described in Section 4.2, and then a smaller set of language-specific *persistent* import edges are derived from this. Each candidate edge from a scope is only followable by self-influencing import resolutions in the same scope, each of which contributes to that set. The fixed-point computation is such that the set of candidate edges grows until no more import resolutions can be found to draw candidate import edges for. The persistent edges are the ones which encode the desired import semantics. On the other hand, for a given program, the set of candidate edges is the same for any self-influencing import semantics. One approach is distinguished from another by how the *filtering* of candidate resolutions is defined.

```
inh circ attr IMPc:[Path] init [];
nt Scope with LEX, VAR, IMP, IMPc, name;

inh attr s:Scope;
syn attr ok:Bool, ty:Ty, s_VAR:[Scope], s_MOD:[Scope];
syn circ attr s_IMPc:[Path] init [];

nt Stmts with ok, s, s_MOD, s_VAR, s_IMPc;
prod cons: top:Stmts ::= d:Stmt ds:Stmts | sm':Scope {
  d.s = top.s; ds.s = top.s;
  top.s_MOD = d.s_MOD ++ ds.s_MOD;
  top.s_VAR = d.s_VAR ++ ds.s_VAR;
  top.s_IMPc = d.s_IMPc ++ ds.s_IMPc;
  top.ok = d.ok && ds.ok; }

nt Stmt with ok, s, s_MOD, s_VAR, s_IMPc;
prod mod: top:Stmt ::= x:String ds:Stmt {
  sm' = mkScopeDcl(x); ds.s = sm'; sm'.LEX = [top.s];
  sm'.VAR = ds.s_VAR; sm'.MOD = ds.s_MOD;
  sm'.IMPc = ds.s_IMPc;
  sm'.IMP = lm-post-process(sm'.IMPc);
  top.ok = ds.ok; top.sm_MOD = [sm'];
  top.s_VAR = []; top.s_IMPc = []; }
prod imp: top:Stmt ::= i:String {
  top.s_IMPc = query_imp(top.s, LEX*IMP?MOD, i);
  top.s_MOD = []; top.s_VAR = [];
  top.ok = true; }
```

Figure 6. LM under recursive/unordered imports.

5 Imports as Circular Attributes

The ideas developed in Section 4 can be implemented in an attribute grammar by using *circular attributes*. Circular attribute grammars were introduced by Farrow [5], and combined with reference attribute grammars [8] by Hedin et al. [14]. These are attributes that may transitively depend on themselves, using fixed-point computation to compute cycles iteratively. Each attribute in one of these dependency cycles must be declared as circular, and must satisfy the conditions for fixed-point computation as discussed in Section 4. That is, they must be defined with an initial value which grows monotonically with each iteration of the fixed-point computation, and the output of one iteration is the input of the next. Each iteration computes every attribute in a cycle once by following the corresponding dependency graph. If an attribute in the dependency cycle is demanded more than once in an iteration, its result from the previous iteration is used. It is circular reference attribute grammars (CRAGs) that we use here since our fixed-point computation computes sets of scope references.

We identified in Section 4 that each module has associated with it a strongly connected component that describes influence between imports in that module. We define our circular attributes correspondingly, such that their dependencies encode the influence of imports on each other. Each scope has an `IMPc` attribute defining the candidate import edges for that module. This attribute and its occurrence on the `Scope` nonterminal are seen on lines 1–2 of Figure 6. `s_IMPc` is defined as a circular (`circ`) synthesized attribute on line 6. Note that the type of the circular attributes is `[Path]` as opposed to `[Scope]`. To do post-processing we need to know the path that leads to each module scope found during a query, particularly for ensuring coherence of unordered imports, and cannot rely on only that target scope itself. We implement sets of candidate imports found during the fixed-point computation as lists, with `[]` as the initial value and `++` to append two lists. A list representation is sufficient because this computation only involves adding paths that yield new candidate `IMP` edges to the set and following those yielded edges in successive iterations; the unordered property of sets is not needed for our purposes. The other edge attributes `LEX`, `VAR` and `MOD` have the same definition as in Figure 3 and Figure 4.

We described in Section 4 that each strongly connected component associated with a module consists of only the import queries in that module. This was without consideration of an attribute grammar implementation, which makes use of synthesized attributes to communicate data up a syntax tree. The attribute which lifts up the target scopes of `IMPc` edges becomes part of the dependency cycle under self-influencing imports and must be circular. The cycle of attribute dependencies for a module is represented by the equations that define these attributes in Figure 6, as well as the `query_imp` function. This function differs from the `query` in Figure 3 as it depends on the circular `IMPc` attribute of its source scope as opposed to the `IMP` attribute, thus completing the dependency cycle. The post processing function is denoted `lm-post-process` on line 21 of Figure 6, which takes as argument all of the paths yielding candidate `IMP` edges. The `IMP` attribute of a module scope, the list of persistent import edges, is defined as its result. In computing the tentative import edges on a scope, we do not need to do the ambiguity check for imports that we see for production `imp` in Figure 3. This is because we are gathering all reachable tentative imports. It is for the post-processing function to decide what to do with ambiguities when computing the persistent import edges.

An initial demand of the `IMP` attribute of a scope demands the same scope's `IMPc` attribute by line 21 of Figure 6. This then begins the fixed-point computation of our circular attributes for the associated module. Attributes on other scopes, such as those of lexical parents, may also be demanded during this computation. In these situations, the circular computation of imports in the original module is paused until a value comes from that demand. For a lexical parent scope, this may cause its `IMPc` circular attribute to be computed if our query demands its `IMP` attribute. After these demanded edge attributes are computed, computation of the original fixed-point computation continues. When `IMPc` is computed for a given scope the resulting value, a list of references to scopes at the end of candidate import edges, the attribute is marked as computed with that value.

6 Discussion

Recall that a coherent import resolution is one that only uses *persistent* import edges. That is, import edges that appear in the final scope graph. We highlighted in Section 4 that resolutions under recursive imports are always stable, but not always coherent. On the other hand unordered imports as we have defined them lead to always coherent but sometimes unstable resolutions. This is because a resolution that would be stable is often ignored in favor of one which is coherent with respect to the other imports in a module. With recursive imports we instead pick the resolution with the most minimal candidate import path with respect to the language-specific label ordering. As a consequence we forget the candidate import edges used in such a path's resolution. This relationship suggests that the properties of stability and coherence are at odds with one another, that in general only one of these holds for a particular resolution semantics. We do not claim that either property is more desirable than the other, but leave the choice of which to satisfy to the language designer.

Self-influencing imports are admittedly a peculiar language feature. As far as we have seen, Rust is the only example of a ubiquitous general purpose programming language that has self-influencing imports. One reason for this is that uncareful use of such imports for more complex programs than those given in this work could lead to incomprehensible resolution behavior. On top of this, they are often much more difficult to reason about than conventional kinds of imports, sequential for example. Our purpose here was to show that attribute grammars, when extended with circular attributes, go a step further than Statix and our previous use of attribute grammars [2], and not to recommend the use of self-influencing imports. To that end, we have provided a generic means for computing such imports by a generic fixed-point computation of *candidate* resolutions that can be defined using circular attributes, followed by a language-specific filter step that distinguishes one semantics from another.

A kind of semantics that we have not considered here are *transitive* imports. These allow resolution paths to use multiple import edges, meaning not only names declared in imported module are visible, but also those imported by that module. An example query regular expression for transitive imports in LM is `LEX* IMP* MOD`, allowing any number of successive `IMP` edges in a resolution path. Under transitive imports it is possible for a module `A` to import itself, if it imports another module `B` which has its own import of `A`. If implementing transitive sequential or parallel imports, one may place a simple restriction on resolution paths determining that they cannot be circular in this way. The DFAs used to compute queries would also be defined to satisfy the new query regular expression. The implementation of queries under transitive sequential or parallel imports can remain unchanged otherwise. On the other hand, combining transitive with self-influencing imports requires more modification. In terms of the fixed-point computation of candidate `IMP` edges, there is no longer one strongly connected component per LM module. There would instead be larger ones that encompass multiple modules. For our example this means that the `IMPc` circular attributes for modules A and B, as discussed in Section 5, would depend on one another. An implementation of a coherent self-influencing import semantics would then have to take into account the candidate import resolutions for each module in such a cycle. One approach to this could be to compute the candidate import resolutions for every import resolution in a program collectively, and then perform one filter over them all that enforces coherence.

7 Related Work

Name binding has been a topic of interest for a long time, and many systems exist to relate name references with declarations in programs. De Bruijn indices [3] are numeric representations of names where references are replaced with indeces that are shared with declarations of the same name. Name resolution then acts uniformly for identical indices in a scope. The higher-order abstract syntax of and Miller and Nadathur [15], and later Pfenning et al.[17], represents specification of binding constructs as a polymorphic simply-typed λ-calculus, in which object language variables become meta-variables in the abstract syntax. Gabbay et al. [6] use nominal sets to develop a meta-theory of formal systems that have name binding operations. Each of these formalisms supports the definition of program transformations involving names, such as substitution and α-conversion, by lifting program names up to the meta level. Using scope graphs and attribute grammars we can implement transformations such as these by, for instance, resolving program references to declarations using scope graph queries and then rewriting those program references to the value their declaration is defined with by using forwarding of attribute grammars - a notion that is out of scope for this paper.

Statix, introduced by van Antwerpen et al. [21], is a constraint-solving system for specifying the name and type analysis of languages in terms of predicates over abstract syntax trees. This first formulation of Statix was found to be too restrictive in its scheduling of name resolution, over-approximating extensions to a scope graph and blocking the execution of queries whose result would in fact be stable. Further development of Statix by Rouvoet et al. [18] allowed for sound interleaving of name resolution and the extension of a scope graph by modifying the conditions on when a query can be executed. Their approach uses the presence of assertions of *weakly critical edges* for a given query, edges whose addition to the graph *may* affect the result of that query, as a condition for blocking the query from executing. Thus name resolution only occurs when the scope graph is deemed ready enough for the query, as opposed to starting a query and building necessary edges along the way as we do. Statix also does not support name resolution that is self-influencing, with constraint-solving becoming stuck. In our previous work [2] we determined that for corresponding name/type analysis specifications, Statix and non-circular attribute grammars produce the same result. For instance, when constraint solving in Statix becomes stuck during self-influencing name resolution, we have an attribute cycle in the attribute grammar.

Later work by Poulsen et al. [1] supports unordered imports for LM, expanding on previous work on Statix [18, 21]. Their approach schedules name resolution for scope graphs by using the applicative functors of Gibbons et al. [7] to map syntax trees onto phased monadic computations which allow fine tuning of the order in which names are resolved. They implement unordered imports by trying sequential resolutions in every permutation of import references in a module, speculatively adding candidate edges to the scope graph during the computation of each permutation, and backtracking the state of the scope graph if no good resolution is found for a particular permutation. The approach they describe is more restrictive than ours, in that stability is always enforced even when name resolution is coherent, which limits the set of acceptable programs. For example the program in Figure 1b is deemed erroneous based on a stability error, rather than by ambiguity of results as the Rust compiler determines.

We are not the first to consider scope graphs in attribute grammars. Kastens and Waite [11] define an alternate, although less general, scope graphs model implemented in attribute grammars with a built-in edge set and a notion of multiple scope graphs for a program with multiple namespaces. Our goal here is not to advocate for any particular formalism for specifying name binding systems, but to solve a particular problem we identified with the expressiveness of conventional attribute grammars and Statix. That is, neither implementation of scope graphs has the expressive power to define the semantics of self-influencing name resolution.

Circular attribute grammars [5] have been used in various applications such as liveness analysis of programs [5, 10, 14, 19], constant propagation [10], and computation of nullable, first, and follow sets in parsers [14]. Our use of circular attributes was a natural solution to a problem we with self-influencing imports that we observed in our previous work [2], namely that the corresponding name analysis is circularly defined. Of particular interest to our work are the circular reference attribute grammars of Magnusson et al. [14], which combine the reference attribute grammars of Hedin et al. [8] with circular attribute grammars, allowing reference attributes to be computed using circular attributes, which we exploit in the computation of candidate import edges in a scope graph.

8 Conclusion

We have shown that self-influencing imports can be resolved in scope graphs using circular reference attribute grammars. By supporting this kind of import behavior, we have gone a step further than Statix and our previous work [2], which both abruptly fail with errors when name resolution is circularly defined. These additions result in a formalism that has the expressive power to support Rust-style self-influencing imports, as well as other similar import semantics.

Future work on this topic could include extensions to Statix, described in Section 2, supporting fixed-point computation for self-influencing imports using the ideas presented here. This would give Statix the expressive power to support language specifications that we employ circular attributes for. Another direction involves further investigating the idea of transitive imports in our formulation, which we have not focused on here. As mentioned in the discussion, this may mean implementing program-wide candidate import filtering when implementing a coherent self-influencing import semantics. A formalism of these kinds of imports and their properties would be an interesting application of scope graphs, and would further highlight their flexibility.

References

[1] Casper Bach Poulsen, Aron Zwaan, and Paul Hübner. 2023. A Monadic Framework for Name Resolution in Multi-phased Type Checkers. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Cascais, Portugal) *(GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 14–28. doi:10.1145/3624007.3624051

[2] Luke Bessant and Eric Van Wyk. 2025. Scheduling the Construction and Interrogation of Scope Graphs using Attribute Grammars. In *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering (SLE '25)* (Koblenz, Germany). ACM. doi:TODO

[3] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.

[4] Torbjörn Ekman and Görel Hedin. 2004. Rewritable Reference Attributed Grammars.. In *18th European Conference on Object-Oriented Programming, ECOOP 2004 (Lecture Notes in Computer Science, Vol. 3086)*. Springer, 144–169.

[5] R. Farrow. 1986. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN Notices* 21, 7 (1986), 85–98. doi:10.1145/13310.13320

[6] Murdoch J. Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Form. Asp. Comput.* 13, 3–5 (July 2002), 341–363. doi:10.1007/s001650200016

[7] Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2022. Breadth-First Traversal via Staging. In *Mathematics of Program Construction*, Ekaterina Komendantskaya (Ed.). Springer International Publishing, Cham, 1–33.

[8] Görel Hedin. 1999. Reference Attribute Grammars. In *2nd International Workshop on Attribute Grammars and their Applications (WAGA)*, Didier Parigot and Marjan Mernik (Eds.). INRIA Rocquencourt, 185–204. http://www-sop.inria.fr/members/Didier.Parigot/WAGA99/waga99.html

[9] T. Johnsson. 1987. Attribute grammars as a functional programming paradigm. In *Proc. of Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 274)*. Springer-Verlag, Berlin, Heidelberg, 154–173. doi:10.1007/3-540-18317-5_10

[10] L.G. Jones. 1990. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 429–46.

[11] Uwe Kastens and William Waite. 2017. Name analysis for modern languages: a general solution. *Software: Practice and Experience* 41 (2017), 1597—1631.

[12] Donald E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. doi:10.1007/BF01692511 Corrections in **5**(1971) pp. 95–96.

[13] E Magnusson and G. Hedin. 2003. Circular reference attribute grammars - their evaluation and applications. In *Proc: LDTA 2003 (ENTCS, Vol. 82)*. Elsevier Science, 532–554.

[14] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. doi:10.1016/j.scico.2005.06.005 Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA '03).

[15] Dale Miller and Gopalan Nadathur. 1987. *A logic programming approach to manipulating formulas and programs*. University of Pennsylvania. Moore School of Electrical Engineering ....

[16] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015 (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 205–231. doi:10.1007/978-3-662-46669-8_9

[17] Frank Pfenning and Conal Elliott. 1988. Higher Order Abstract Syntax. In *Proc. ACM SIGPLAN '88 Symposium on Language Design and Implementation*. 199–208.

[18] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 180 (Nov. 2020), 28 pages. doi:10.1145/3428248

[19] A. Sasaki and S. Sassa. 2000. Circular Attribute Grammars with Remote Attribute References. In *Proceedings of 3rd Workshop on Attribute Grammars and their Applications*. 125–140.

[20] Michael J Todd. 2013. *The computation of fixed points and applications*. Vol. 124. Springer Science & Business Media.

[21] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (Oct. 2018), 30 pages. doi:10.1145/3276484

[22] H. Vogt. 1989. *Higher order attribute grammars*. Ph. D. Dissertation. Department of Computer Science, Utrecht University, The Netherlands.

[23] Aron Zwaan. 2022. Specializing Scope Graph Resolution Queries. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 121–133. doi:10.1145/3567512.3567523

[24] Aron Zwaan and Hendrik van Antwerpen. 2023. Scope Graphs: The Story so Far. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:13. doi:10.4230/OASIcs.EVCS.2023.32