Scheduling the Construction and Interrogation of Scope Graphs Using Attribute Grammars

Luke Bessant
bessa028@umn.edu
University of Minnesota, Twin Cities
Minneapolis, MN, USA

Eric Van Wyk
evw@umn.edu
University of Minnesota, Twin Cities
Minneapolis, MN, USA

Abstract

Recognizing that name binding is often done in an ad-hoc manner, Visser and his colleagues introduced scope graphs as a uniform representation of a program's static binding structure along with a generic means for interrogating that representation to resolve name references to their declarations. A challenge arises in scheduling the construction and querying actions so that a name resolution is not performed before all requisite information for that resolution is added to the scope graph. Visser et al. introduced a notion of weakly critical edges to constrain the order in which name resolution queries are performed to a correct one, but this has been found to be somewhat restrictive.

Visser et al. also introduced Statix, a constraint solving language for scope graph-based name resolution. We show that specifications written in an annotated version of Statix can be translated into reference attribute grammars, and that the order in which equations are solved under demand driven evaluation provides a valid order for solving constraints in Statix. This formalizes what has been folklore in the attribute grammar community for some time, that scope graphs are naturally specified in reference attributes grammars.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**.

Keywords: name binding, scope graphs, attribute grammars

ACM Reference Format:
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), June 12-13, 2025, Koblenz, Germany*. ACM, New York, NY, USA, 14 pages.

SLE '25, Koblenz, Germany


```
1  module A1 {
2   def y1 =
3    let v1 = 2
4    in 1 + v2
5  }
6
7  module B1 {
8   import A2
9   def x2 = y2
10 }
```

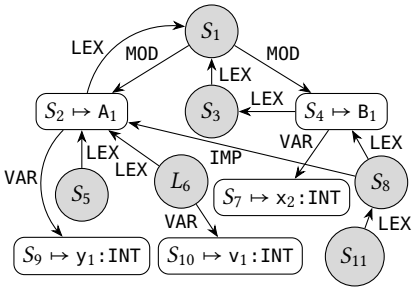

Figure 1. An LM program and its complete scope graph.

1 Introduction

The problem of name analysis, binding references to their declarations in a program, is often treated in language-specific ad-hoc manner. Visser and his colleagues [14] recognized this and, over a series of papers [1, 14, 15, 17, 20] developed *scope graphs* as a uniform representation of the static name-binding structure of a program and a constraint solving language Statix for specifying the solution to name analysis problems. In this approach scopes and declarations are represented graphically as nodes in a graph with labeled directed edges defining relationships such as lexical parenthood between scopes and name declarations within a scope. Name resolution is realized as a walk over the graph from the reference's scope node to possible declaration nodes. These resolution paths must satisfy a data well-formedness predicate associated with the lookup that identifies possible declarations. The set of allowed resolution paths is also constrained to those whose label form a word in a path well-formedness regular expression over edge labels.

Consider the example program in Figure 1 written in a version of LM, a family of languages defined by Neron et al. [14] for studying scope graphs and Statix. An issue arises with importing modules to make the names they define visible in a scope: resolving those imported names depends on first resolving the import. For example, while the `LEX` (lexical parent), `VAR` (local variable declaration) and `MOD` (local module declaration) edges are derived from the syntactic structure of the program, the `IMP` edge in Figure 1 arises from the `import` of module `A` by module `B`. For Figure 1, when the path well-formedness regular expression is `LEX* IMP? VAR`, the lookup of y_2 depends on first resolving A_2 and extending the scope graph with the resulting `IMP` edge. We then have

resolution path $S_8 \xrightarrow{\texttt{IMP}} S_2 \xrightarrow{\texttt{VAR}} S_9$ finding $\texttt{y}_1$. If we resolve $\texttt{y}_2$ without this edge, the result is *unstable*, something that must be avoided as later additions to the scope graph change the result of the query. Thus a key issue is determining when it is safe to execute a query so that it is *stable*.

Rouvoet et al. introduced the notion of *weakly critical* edges with respect to a query [15]. These are edges that are asserted but unbuilt, which *may* become part of a resolution path for a query. Their presence indicates that a query must not yet be executed as the result may be unstable. Van Antwerpen et al. [17] introduced Statix, a constraint solving name resolution language for scope graphs. Statix specifications define syntax-directed predicates and constraints that define what constitutes a correct program with respect to name binding and typing. During program analysis, constraints are non-deterministically chosen from a working set to be solved, but may be put back if that the constraint is not yet solvable. Statix recognizes *weakly critical edges* to decide whether the current state of the scope graph is ready for a given query to be solved. If there are weakly critical edges for it, a query constraint is put back into the working set. This can lead to a Statix program analysis becoming unresolvably stuck. A query cannot be started until a particular edge is built, but the edge cannot be built until that query is complete: a circular dependency issue. The weakly critical edges approach has shown to be too restrictive to support more complicated name binding systems, such as the unordered self-influencing `use` declarations in Rust.

Attribute grammars [12] (AGs) are also constraint-solving systems, but the constraints are directed equations that define an attribute on the left-hand side based on attribute references on the right. Attributes decorate the nodes of a syntax tree with values which are defined by these equations. Attribute evaluation propagates information up and down the syntax tree. Importantly the dependencies of attributes, based on their defining equations, determine a schedule under demand-driven evaluation. Reference [9] and remote [3] AGs support attributes that are references to other nodes in the syntax tree, thus allowing a graph structure to be superimposed onto the program's tree. Here, we will show that demand-driven [10] reference AGs [5, 9, 13] provide a natural way to schedule the construction and interrogation of scope graphs, where dependencies between name resolution operations and building edges in a scope graph allow us to dynamically derive a sound schedule. During a name resolution, we can identify the unbuilt edges which are weakly critical for that resolution and demand that they are added to the graph. This may involve building nodes or completing other name resolutions before continuing with the original. AGs thus provide a means to schedule the solution of Statix constraints and also leads to a better understanding other similarities and differences between these two approaches to name resolution.

Contributions and roadmap: Section 2 provides background on scope graphs. Section 3.1 provides Statix background and Section 3.2 describes our annotations to and restrictions to Visser et al.'s version that facilitates the translation to AGs. Section 4.1 provides some background on AGs and Section 4.2 gives our operational semantics of demand-driven evaluation. Section 5 describes how annotated Statix specifications are translated to AGs. Section 6 shows how the trace of AG evaluation operational semantics corresponds to a trace of Statix constraint solving semantics and produce the same results. Interestingly, Statix specifications that get stuck on missing weakly critical edges corresponds to a cycle in the attribute grammar. Section 7 discusses some aspects of this process; Section 8 described related work. Section 9 describes some future work and concludes.

2 Background: Scope Graphs

Scope graphs graphically represent the scoping structure of programs. They contain nodes that represent either program scope regions or name declarations. One reason for this uniformity is that constructs like `module` in Figure 1 both define a name and indicate a program region; thus all nodes are referred to as *scopes*. Information pertaining to an object language declaration, such as its name, is associated with a scope by a partial relation mapping scopes to object language terms $\mathcal{T}$. Scope graph edges define relationships between scopes with a label in $\mathcal{L}$, defined by the object language. For instance, the `LEX` label in Figure 1, indicating that the target lexically encloses the source, a common relationship.

Definition 2.1. A *scope graph* is a triple $G = \langle S_G, E_G, \rho_G \rangle$ with a set of nodes S_G, edges $(s_1, l, s_2) \in E_G$ with $s_1, s_2 \in S_G$ and $l \in \mathcal{L}$, and $\rho_G : S_G \rightharpoonup \mathcal{T}$ mapping nodes to data terms.

S_G is a set of scope identifiers which are unique and are represented as a name with an identifying numeric subscript. Resolution in a scope graph is performed by *queries*. These are traversals which, from a start scope, find all *reachable* scopes with respect to a regular expression over $\mathcal{L}$. Any scope at the end of a path from the query start scope whose labels form a word in the language of the regular expression is *reachable* [15]. Each query has a data predicate restricting the set of valid resolutions, here, to match reference names to declarations. A query for reference $\texttt{y}_2$ in Figure 1 with regular expression `LEX* IMP? VAR` yields the path $S_8 \xrightarrow{\texttt{IMP}} S_2 \xrightarrow{\texttt{VAR}} S_9$, finding the node for declaration $\texttt{y}_1$.

Definition 2.2. A *query* over graph G with label set $\mathcal{L}$ is a function $S_G \times R_{\mathcal{L}} \times (\mathcal{T} \to \texttt{bool}) \to R$, yielding a set of resolution paths R, such that $r \in \mathit{query}(s, rx, d)$ whenever $G \vdash r : s \xrightarrow{w} s'$, $w \in L(rx)$, and $d(\rho_G(s'))$.

Edges are added to a scope graph during analysis, *e.g.* an import may lead to a new `IMP` edge from the importing to imported scope. Rouvoet et al. [15] recognized that the

resolution of a query, which can only use the edges in the graph at the time of its execution, must be *stable*. That is, no new additions to the scope graph will affect the result of the query. If every query result is *stable*, then name analysis is sound. Ensuring stability is a key aspect of this and previous work [15].

3 Statix: Annotated, Restricted

Statix is a language for specifying constraints defining the name and type analysis of a language. In this work, we define a restricted variant of Statix-core presented by Rouvoet et al. [15] and refer to it simply as Statix. These modifications are minor and used to facilitate the translation of Statix to attribute grammars discussed in Section 5.

3.1 Statix background: syntax and semantics

Statix specifications consist of predicates with constraint bodies. Constraints operate over terms, sets, and variables in the specification. As constraints are solved, fresh variables can be introduced into constraints that are later replaced by ground terms by solving other constraints. During evaluation, the scope graph G is constructed and (unsolved) constraints are held in a working set $\overline{C}$ that grows as user-defined predicates are expanded, and shrinks as constraints are solved. The state of evaluation is denoted $\langle G|\overline{C}\rangle$. The set $\overline{C}$ is treated as the conjunction of its elements, so a program is only satisfiable (type correct) if every constraint that is introduced during the program's analysis is solved. Thus the final state for a correct program is its complete scope graph and the empty set of constraints, $\langle G|\varnothing\rangle$. Failure results in a single constraint in state $\langle G|\{\texttt{false}\}\rangle$. Rouvoet et al. [15] provide an operational semantics of Statix using stepping rules of the form $\langle G|\overline{C}\rangle \to \langle G|\overline{C}\rangle$. The initial state is the empty scope graph and a top-level constraint over the program term. Some constraints in $\overline{C}$ can be solved and may be chosen; some may not be, and depend on other constraints being solved first. Most constraints require ground arguments, *e.g.* an edge from s_1 to s_2 cannot be added until those scopes are identified. When there are active constraints and none can be solved, the system is said to be *stuck*.

Figure 2 provides the syntax of Statix predicates and constraints. Our annotations, discussed in Section 3.2, are italicized and colored and can be disregarded until then. Solving a `true` constraint simply removes that constraint from $\overline{C}$, while solving `false` steps Statix to a failure state where there is no more work left to do, as we have determined the program is unsatisfiable. Similarly, a true equality or inequality constraints are removed from the set when solved; otherwise, a `false` constraint is added. This differs from the definition of equality provided by Rouvoet et al. [15], that instead steps to a failure state. This difference allows a closer correspondence between Statix and our translation of it, and is discussed in Section 6. Solving a conjunction simply adds both operands to the working constraint set. The definition constraint substitutes the ground term `t` for variable `x` in the constraint set. The exists constraint replaces quantified names with fresh variables in the component constraint, and introduces that constraint to the working set. Predicates are solved by expansion - the constraint within the predicate definition is added to the set after argument substitution. The `new` $x \to t$ constraint asserts a new scope with associated data t. We present the Statix step rule for this constraint

Signatures

l	$\in$	$\mathcal{L}$	edge labels
f	$\in$	$\mathcal{F}$	compound term names
p	$\in$	$\mathcal{P}$	predicate id/nonterminal
rx	$\in$	$\mathcal{R}$	regular expressions

Variables

x	$\in$	$\mathcal{X}$	term variables
s	$\in$	$\mathcal{S}$	scope identifiers

Annotations (α, γ) and Types (τ)

α	::=	$@syntax \mid \epsilon$	predicate annotations
γ	::=	$@inh \mid @syn$	inherited/synthesized
	$\mid$	$@ret \mid \epsilon$	returns/arguments
τ	::=	p	nonterminal type
	$\mid$	$scope \mid datum \mid path \mid \ldots$	built-in type names
	$\mid$	$\{\tau\}$	set type

Predicates

P	::=	$\alpha\ p((\gamma\ x\ :\tau)^*)$ `:-` c `.`	predicates

Constraints

$c \in C$	`::=`	`true` $\mid$ `false`	true/false
	$\mid$	t `==` t $\mid$ t `!=` t	equality/inequality
	$\mid$	x `:=` t	definition
	$\mid$	c`,` c	conjunction
	$\mid$	`{`$(x:\tau)^*$`}` c	exists quantification
	$\mid$	$p(t^*)$	predicate use
	$\mid$	`new` $x \to t$	scope assertion
	$\mid$	t `-[`l`]->` t	edge assertion
	$\mid$	`getData`(t, x)	scope data assertion
	$\mid$	`query`(t, rx, d, x)	query
	$\mid$	`min`(t, ord, x)	minimum
	$\mid$	`single`(t, x)	singleton
	$\mid$	`inhabited`(t)	non-empty
	$\mid$	t `match {` $(t:\tau \to c)^*$ `}`	pattern matching

Terms, Variables, Identifiers, Sets

$t \in \mathcal{T}$	`::=`	$f(t^*)$	compound terms
	$\mid$	$s \mid l$	scope/label identifiers
	$\mid$	x	term variables
	$\mid$	$\varnothing \mid \{t^*\}$	sets

Scope Graph

G `::=` $\langle S \subseteq \mathcal{S}, E \subseteq (\mathcal{S} \times \mathcal{L} \times \mathcal{S}), \rho \subseteq (\mathcal{S} \times \mathcal{T})\rangle$

Figure 2. Statix syntax, along with our annotations.

here, but leave most to Appendix A.

$$\frac{s \notin S_G}{\langle\langle S, E, \rho\rangle | \texttt{new}\ x \to t; \overline{C}\rangle \to \langle\langle s; S, E, [s \mapsto t]\rho\rangle | [s/x]\overline{C}\rangle} \textsc{Op-Node}$$

It introduces a fresh scope identifier s that is inserted into S_G and substituted for x elsewhere in the constraint set, then update ρ_G with $s \mapsto t$.

Solving an edge assertion s_1 `-[l]->` s_2 updates the scope graph so that $(s_1, l, s_2) \in E_G$, building a new edge. We assert the data held in a scope with `getData(`t, x`)`, which, when t is a scope identifier s, substitutes $\rho_G(s)$ for x in the constraint set. The `query` constraint implements queries over scope graphs as described in Section 2, substituting the result of the query for x. The `min` constraint computes the minimum of the set given as the first argument with respect to the partial ordering given as its second. `single(`ts`, `x`)`, when ts is a singleton set $\{t\}$, steps to a state where t is substituted for x in the constraint set. Otherwise, when ts is not a singleton, a `false` constraint is introduced to the set. `inhabited(`ts`)` is satisfiable when ts is non-empty. The `match` constraint operates in the standard way, with constraint C associated with a matching case being introduced to the constraint set.

We provide an example of a Statix specification in Figure 3 for LM, a simple language with expressions and modules. The initial constraint set in Statix for LM is `{main(`p`)}` where p is the program term. This is expanded using the definition of `main`, and constraint solving continues on the resulting state. The `import` case of predicate `dcl`, and the `ref` case of `exp` both implement name resolution using queries. The former for resolving an import reference to a module scope, and the latter performing LM variable resolution.

To address the issue of query stability and ensure soundness of name resolution in Statix, previous work [15] introduced the notion of *weakly critical edges* for a query. These are edges asserted in the constraint set, but missing from the scope graph, whose solving *may* result in a graph which yields new paths for the query. The presence of any such edges for a query determines that the query must not be solved in the current state of the scope graph. Definition 3.1 gives an amended version of the Rouvoet et al.'s definition of weakly critical edges [15].

Definition 3.1. An edge (s_1, l, s_2) is *weakly critical* with respect to a graph G and *query*(s, r, d) when:

1. $G \vdash p : s \xrightarrow{w} s_1$ for some word w,
2. Word wl is a prefix of a word in $L(r)$,
3. $(s_1, l, s_2) \notin E_G$.

As an example, consider a situation where during constraint solving in Statix we have a modified version of the scope graph in Figure 1, but with the `IMP` edge missing, and our constraint set is the following:

```
{ S8 -[IMP]-> S2,
  query(S8, LEX* IMP? VAR, var-is("y"), res), ... }
```

```
1  @syntax main(m: main) :- m match
2  { prog(ds: dcls) -> new sg -> noDatum(),
3                      dcls(sg, sg, ds)
4  }.
5  @syntax dcls(@inh s: scope, @inh sm: scope, ds: dcls) :-
6  ds match { nil() -> true
7  | cons(d: dcl, rest: dcls) -> {sn: scope}
8      new sn,  sn -[LEX]-> s,
9      dcl(s, sn, sm,d), dcls(sn, sm, ds)
10 }.
11 @syntax dcl(@inh s: scope, @inh sn: scope,
12             @inh sm: scope, d: dcl) :- d match
13 { mod(x: string, ds: dcls) ->  {sm': scope}
14     new sm' -> datumMod(x),
15     sm -[MOD]-> sm', sm' -[LEX]-> s,
16     dcls(sm', sm', ds)
17 | def(x: string, e: exp) -> {sv: scope, t: ty}
18      new sv -> datumVar(x, t), sm -[VAR]-> sv,
19      exp(s, e, t)
20 | import(x: string) ->
21   {rs: {path}, rs': {path}, r: path, sm: scope}
22     query (s, LEX* IMP? MOD, mod-is(x), rs),
23     min(rs, LEX > IMP > VAR = MOD, rs'),
24     single(rs', r),
25     tgt(r, sm), sn -[IMP]-> sm
26 }.
27 @syntax exp(@inh s: scope, e: exp, @syn ty: ty) :- e match
28 { add(l: exp, r: exp) ->  {tyL: ty, tyR: ty}
29     exp(s, l, tyL), tyL == INT(),
30     exp(s, r, tyR), tyR == INT(),
31     ty := INT()
32 | ref(r: string) ->
33   {rs: {path}, rs': {path}, r: path, s': scope, d: datum}
34     query (s, LEX* IMP? VAR, var-is(r), vars),
35     min(rs, LEX > IMP > VAR = MOD, rs'),
36     single(rs', r),
37     tgt(r, s'), getData(s', d),
38     d match {datumVar(_, ty'): datum -> ty := ty'}
39 }.
40 tgt(r: path, @ret s: scope) :- r match
41 { end(s'): path -> s := s'
42 | edge(_, _, r'): path -> tgt(r', s) }.
```

Figure 3. A Statix specification for toy language LM

The `IMP` edge asserted is weakly critical for the query, which can follow such an edge from S_8 according to the query regular expression. Thus the query must not be solved until the edge asserted has been added to the scope graph, using rule Op-Edge. Weakly critical edges can be identified by syntactic analysis of the constraint set [15], which must occur before the query is solved. This in itself entails a traversal of the scope graph, at each scope looking for weakly critical edges which are sourced from that node, and delaying the query if any are found.

This behavior of delaying a query if there are any weakly critical edges for it works, but due to the non-deterministic nature of Statix, may result in the query being selected any number of times before it is solvable. From a demand-driven perspective, we would like for the query to merely *pause*

when a weakly critical edge is found, immediately build that edge, and then continue the query. How we implement this approach is a focus of successive sections of this work.

3.2 Annotations and restrictions

We specify a class of Statix specifications that can be translated to attribute grammars by imposing a few modest restrictions on how they are written, and add annotations to Statix to simplify the translation discussed in Section 5. The translation is also facilitated by the addition of type annotations as shown in Figure 2 and Figure 3, since types, such as nonterminal symbols, are needed in the AG specification. A number of built-in types such as *scope*, *path* and *datum* are thus provided.

The *@syntax* annotations in Figure 2 identify the syntactic structure, the grammar, of the language defined. For example, `dcl` on lines 11–12 and `exp` on line 27 in Figure 3. We call such predicates *syntax predicates* and others such as `tgt` *function predicates*. A structural restriction is that syntax predicates must consist of a `match` constraint at the top level, which breaks down the syntax term given as the primary argument in order to derive the syntactic structure from the `match` cases. We also require that each `match` case contains a single existential constraint at the top level. This determines the local names (attributes) of the corresponding production as described in Section 5. The arguments to a syntax predicate must be annotated with *@syn* (*synthesized*) or *@inh* (*inherited*) to identify the flow of information in that predicate. These correspond to synthesized and inherited attributes introduced in Section 4. Similarly, function predicates must use *@ret* to identify returns, with other parameters treated as proper arguments.

A constraint-level restriction we make is that a `new` constraint must appear directly in the body of the existential asserting its name. This restriction does not change what can be expressed in Statix, only how the specification is written - but is a restriction on the *permission to extend* analysis defined by Rouvoet et al. [15] which allows the `new` constraint for a scope to appear anywhere under the exists constraint that introduces its name, with respect to predicate expansion.

Another restriction disallows unification in constraint arguments. Previous versions of Statix [15, 17] would allow a constraint such as `exp(s, l, INT())` to replace the two constraints `exp(s, l, tyL), tyL == INT()` we use on line 29 of Figure 3. More complex examples can occur, but this restriction simplifies the the translation to a directed attribute grammar. We also disallow unification of terms in the `==` operator. But this functionality can be expressed by a (sometimes much) more verbose use of `match` constraints. Instead, we treat `==` as an equality check on ground terms, as indicated by our modified semantics rule Op-Eq-True:

$$\frac{t_1\ ground \qquad t_2\ ground \qquad t_1 = t_2}{\langle G | t_1 == t_2; \overline{C}\rangle \to \langle G | \overline{C}\rangle}\ \textsc{Op-Eq-True}$$

These restrictions raise a question about the expressiveness of our formulation versus that of Statix as presented in previous work [15, 17]. We have not performed a rigorous analysis to this effect, but have found that all name analyses in example specifications in the MiniStatix artifact provided by Rouvoet et al. [15], including those that make use of unification, can be expressed in our formulation. These all exhibit the more directed use of constraints that we require. As acknowledged above, this can lead to more verbose specifications, something we consider in Section 9.

4 Attribute Grammars

This section provides background on attribute grammar specifications and our operational semantics of them. This is used in Section 6 to show the correspondence between the evaluation of Statix and demand-driven attribute grammars.

4.1 Background

Attribute grammars [12] (AGs), extended with higher order (non-terminal) attributes [19] and reference [9] (also known as remote [3]) attributes, provide semantics for languages defined by a context free grammar G and have the form

$$\langle G = \langle NT, T, P, S \in NT\rangle, A = A_s \cup A_i \cup A_l \cup A_{nta}, O, EQ\rangle$$

AGs associate with each nonterminal NT in the grammar a set of semantic attributes A that occur on it, specified by $O \subseteq NT \times A$. Productions in P have the form

$$\texttt{prod}\ pn : n_0 : NT_0 ::= x_1 : X_1 ... x_j : X_j \mid nta_1 : NT_1 ... nta_k : NT_k$$

in which the left hand side nonterminal NT_0 and right hand side terminals or nonterminals X_i where $X = NT \cup T$ are labeled with names used to reference those nodes in the equations defining the attributes. For example, Figure 4 contains an attribute grammar fragment (generated from the Statix specification in Figure 3) with an integer `add` production on line 8.

Attributes in A are either synthesized (A_s) and propagate information up the tree, inherited (A_i) and propagate information down the tree, are local (A_l), or are *nonterminal attributes* [19] (A_{nta}). The latter appear after the vertical bar in the production and are effectively additional children of n_0 that are not known in the original tree but are computed and "plugged into the tree" at attribute evaluation time. Lines 1–5, 9, and 15–16 declare attributes for type checking and scope graph construction.

Attribute values can be primitive values such as strings or Booleans but can also be program terms (higher order attributes) or references to attribute-decorated tree nodes in the current tree or another separate tree.

Each equation in EQ is associated with a production in P and computes values of attributes. These have the form $n_0.a = e$ for synthesized or local attributes, $x_i.a = e$ for inherited attributes, and $nta_i = e$ for nonterminal attributes. The language of expressions e is left unspecified here but

```
1  inh attr s:scope; inh attr sn:scope; inh attr sm:scope;
2  syn attr ok:bool; syn attr ty:type;
3  syn attr LEX_s::[scope]; syn attr VAR_s:[scope]; ... ;
4  syn attr LEX_sn:[scope]; syn attr VAR_sn:[scope]; ...;
5  syn attr LEX_sm:[scope]; syn attr VAR_sm:[scope]; ...;
6
7  nt exp with ok, s, ty, LEX_s, VAR_s, MOD_s, IMP_s;
8  prod add: top:exp ::= l:exp r:exp {
9    locals tyL:type, tyR:type;
10   l.s = top.s; r.s = top.s;
11   top.ok <- top.tyL == INT(); top.tyL = l.ty;
12   top.ok <- top.tyR == INT(); top.tyR = r.ty;
13   top.ty = INT();       }
14 prod ref: top:exp ::= x:string {
15   local ps:[path], ps':[path], p:path, s':scope, d:data;
16   local dwce:bool = dwce(top.s, LEX* IMP? VAR);
17   top.ps = query(dwce, top.s, LEX* IMP? VAR, var-is(x));
18   top.ps' = min(top.ps, LEX > IMP > VAR = MOD);
19   top.p = single(top.ps');
20   top.s' = tgt(p);  top.d = s'.getData;
21   top.ty = case top.d of | datumVar(_, ty') -> ty'
22                          | _ -> abort end;      }
23 nt dcl with ok, s, sn, sm, LEX_s, LEX_sn, LEX_sm, ...;
24 prod mod: top:dcl ::= ds:dcls | sm':scope {
25   sm' = mkScope(datumMod(x));
26   sm'.LEX = top.s::ds.LEX_s; sm'.VAR = ds.VAR_s;
27   sm'.MOD = ds.MOD_s; sm'.IMP = ds.IMP_s;
28   top.MOD_sm <- [sm'];
29   ds.s = sm'; ds.sm = sm';
30   top.ok <- ds.ok;       }
```

Figure 4. Attribute, nonterminal, and production definitions corresponding to the syntax predicates in Figure 3.

includes operations over primitive types, access attributes on visible tree nodes, constructing new terms, etc. The discussions that follow introduce various expression forms as they are needed. Several equations are shown in Figure 4. Of interest is the equation on line 25 that creates a tree (representing a scope as described in the following sections) and plugs it into the existing tree so that it can be decorated, as seen on the following 2 lines.

4.2 Operational semantics for attribute evaluation

In AGs, the scheduling problem is to find an order in which to solve the equations for attributes in the tree of an input program. To demonstrate the correspondence between constraint solving in Statix and computing attributes in AGs we provide an operational semantics which determines the schedule during attribute evaluation in a *demand-driven* manner [10]. Demand-driven evaluation is the approach used in recent AG systems such as JastAdd [6] and Silver [18]. The rules realizing this process begin by demanding the value of a synthesized attribute on the root node of the tree. To compute this, values of attributes referenced in the right hand side of the defining equation are demanded, which may then depend on other attributes. When the right hand side of the equation defining a demanded attribute is a value, then that value is stored in the tree and used in the expression that demanded it. A stack of demanded *equation instances* that correspond to and define attribute instances in the tree is maintained, as well as a set of active but undemanded equation instances. The evaluation state is the triple $\langle t|stk;set\rangle$, consisting of (partially decorated) trees t, equation stack stk, and equation set set. AG evaluation is determined by the transitive closure of the stepping relation $\langle t|stk;set\rangle \to \langle t|stk;set\rangle$.

Since we make use of higher-order attributes, t is a forest of trees containing the syntax tree, as well as others defined by nonterminal attributes. Members of t are nodes, each with a unique identifying number, a set of attributes that are *demanded*, *undemanded*, or *complete* with value v, and references to child and NTA nodes. Each node stores an indication that it has been visited or not. We say that a node is visited if at least one of its attributes has been demanded. The stack stk consists of instantiated equations defining *demanded* but not yet computed attributes. For each node in t, its instantiated equations are derived from the equations in that node's production, with unique node identifiers substituted for local node names. Node identifiers are written with a name from an equation and a unique number, such as $\texttt{l}_8$ for an instance of the first child of the `add` production on line 8 in Figure 4. An example of an instantiated equation is $\texttt{l}_8\texttt{.s = top}_{13}\texttt{.s}$ derived from the first equation of line 10. When the node $\texttt{l}_8$ is constructed by the `ref` production on line 14 it refers to itself with the name `top`. Its instantiated equations may use $\texttt{top}_8$ and thus *only* the subscript is used in determining node equality.

The *initial evaluation state* for program p consists of its tree, the instantiated equation defining some synthesized attribute s on the root node and the empty set of equations: $\langle mkTree(p,0)|[ag_0.s = root_1.s];\varnothing\rangle$. The attributed tree is constructed by *mkTree* from the term p and initial node id 0; the equation comes from the production for the syntax predicate `main` with pattern `prog` on line 1 of Figure 3. The rules defining the step relation $\langle t|stk;set\rangle \to \langle t|stk;set\rangle$ are given in Figure 5 and are discussed below. We denote the transitive closure of this relation by $\langle t|stk;set\rangle \to^* \langle t'|stk';set'\rangle$.

We discuss here a few of the attribute grammar evaluation rules to establish some intuition for them. Notation used in the rules but not formally defined includes v *val* asserting that expression v is a value, and n *unvisited in* $\langle t|stk;set\rangle$ to say that node n is unvisited in the attribute grammar state given.

We assign values to attributes instances in the tree with the CompAttr rule once the expression in the defining equation is a value. When such an equation is on the top of the stack it is popped and t is updated so that $n.a$ holds the value e, indicating that $n.a$ is *complete*. Rule CompNTA defines how we build the tree from a term for a nonterminal attribute. *mkTree* will recursively build the tree from term e, using fresh node ids for any child nodes. Attribute slots in the tree are given the *undefined* indicator of $\perp$. We abort evaluation

$\boxed{\langle t|stk;set\rangle \to \langle t|stk;set\rangle}$

$$\frac{e\ val \qquad n \in t}{\langle t|n.a = e :: stk';set\rangle \to \langle n.a := e/t|stk';set\rangle}\ \textsc{CompAttr}$$

$$\frac{e\ val \qquad t' = mkTree(e, nta)}{\langle t|nta = e :: stk';set\rangle \to \langle t \cup t'|stk';set\rangle}\ \textsc{CompNTA}$$

$$\frac{}{\langle t|x = abort :: stk;set\rangle \to \langle t|abort;\varnothing\rangle}\ \textsc{Abort}$$

$$\frac{\langle t|e;x = e :: stk;set\rangle \Rightarrow \langle t'|e';stk';set'\rangle}{\langle t|x = e :: stk;set\rangle \to \langle t'|stk';set'\rangle}\ \textsc{AttrExprStep}$$

Figure 5. AG equation-step rules

with rule Abort if an equations whose defining expression is `abort` is atop the stack.

The AttrExprStep rule makes use of an expression evaluation relation

$$\langle t|e;stk;set\rangle \Rightarrow \langle t|e;stk;set\rangle.$$

This relation works on states that extend the equation state to include the expression currently being evaluated in the equation on the top of the stack. This relation is responsible for two types of evaluation steps. The first is to carry out a small-step evaluation on the expression e until it is reduced to a value. In many instances, the tree t, the stack stk and the set set do not change. The second task is to demand that a not-yet-computed attribute be computed; this may manipulate the state by removing an equation from the set and pushing it onto the stack. We discuss a few of the rules shown in Figure 6 now, and continue when we describe the correspondence between attribute grammar and Statix evaluation states in Section 6. In the Equals-True rule we reduce an equality expression to `true` if e_1 and e_2 are equal. We implement left-to-right short-circuiting conjunction, so that an expression `true && e` is reduced to e and `false && e` to `false` with axioms that we omit for brevity. When the left operand of a conjunction is not a value, rule Conj-Step-Left applies. Apply tells us how to step a function application. We substitute arguments $e_1, ..., e_n$ for $x_1, ..., x_n$ in the function body. These rules are similar to those for small-step semantics in a functional language; similar rules are not shown.

References to complete attributes are reduced to their values by rule Dmd-Complete. An attribute $n.a$ is complete if $n.a \neq \bot$. Demanding an attribute from a visited node is handled by rule Dmd-Visited. The equation defining the attribute instance we demanded must be in the equation set, so we remove it and push it onto the stack. When we

$\boxed{\langle t|e;stk;set\rangle \Rightarrow \langle t|e;stk;set\rangle}$

$$\frac{e_1\ val \qquad e_2\ val \qquad e_1 = e_2}{\langle t|e_1{==}e_2;stk;set\rangle \Rightarrow \langle t|true;stk;set\rangle}\ \textsc{Equals-True}$$

$$\frac{\langle t|e_1;stk;set\rangle \Rightarrow \langle t'|e_1';stk';set'\rangle}{\langle t|e_1\ \&\&\ e_2;stk;set\rangle \Rightarrow \langle t'|e_1'\ \&\&\ e_2;stk';set'\rangle}\ \textsc{Conj-Step-Left}$$

$$\frac{f = \lambda x_1 .... \lambda x_n\,.\,e \qquad e' = [e_1/x_1]...[e_n/x_n]e}{\langle t|f(e_1, ..., e_n);stk;set\rangle \Rightarrow \langle t|e';stk;set\rangle}\ \textsc{Apply}$$

$$\frac{(n.a = v) \in t \qquad v \neq \bot}{\langle t|n.a;stk;set\rangle \Rightarrow \langle t|v;stk;set\rangle}\ \textsc{Dmd-Complete}$$

$$\frac{\begin{array}{c} m \in t \qquad m\ unvisited\ in\ \langle t|stk;set\rangle \\ \{m.b = e\} \cup set' = instantiateEqs(m) \\ t' = t\ with\ m\ visited \end{array}}{\langle t|m.b;stk;set\rangle \Rightarrow \langle t'|\_;m.b = e :: stk;set' \cup set\rangle}\ \textsc{Dmd-Unvisited}$$

$$\frac{\begin{array}{c} n \in t \qquad n\ visited\ in\ \langle t|stk;set\rangle \\ set = \{n.a = e\} \cup set' \end{array}}{\langle t|n.a;stk;set\rangle \Rightarrow \langle t|n.a;n.a = e :: stk;set'\rangle}\ \textsc{Dmd-Visited}$$

$$\frac{m \notin t \qquad set = \{m = e\} \cup set'}{\langle t|m.b;stk;set\rangle \Rightarrow \langle t|m.b;m = e :: stk;set'\rangle}\ \textsc{Dmd-NTA}$$

Figure 6. AG expression-step rules

demand an attribute from an unvisited tree node, rule Dmd-Unvisited applies. We instantiate the equations for that node with node identifiers already stored in t for all node references, pick out the one that defines the attribute instance we demanded, and put the remainder in the equation set. We mark the node as visited in the tree. A nonterminal attribute is demanded using rule Dmd-NTA. The equation defining it must be in the stack, so we remove it in a similar way to rule Dmd-Visited.

5 Translating Statix to Attribute Grammars

In translating Statix specifications to attribute grammars, we note that attribute equations can also be seen as constraints, but ones that are directional, or functional, in that they define the attribute on the left-hand side of the equation using values and attributes references on the right hand side. Since Statix is in a sense a relational language, constraints may conceivably not fit this pattern. However, name and type analyses written in Statix typically exhibit the more

directed style of constraints and thus have natural functional counterparts. The `query` constraint, for example, follows this pattern - the variable given as the final argument is substituted with the result of the query when it is solved, whereas the first three arguments are inputs.

While Statix picks constraints to solve until none remain, demand-driven AG evaluation only solves equations as needed. The translation thus uses a Boolean-valued `ok` attribute. The equations defining this attribute are what drive evaluation and demand that the interesting work, that which corresponds to solving of constraints, is done. When `ok` is true on the root node, the program is deemed correct; when false, the program has an error. This corresponds in Statix, respectively, to successfully solving all constraints or failing.

Syntax and functional predicates: Specifications in Statix that define programming language semantics follow a particular pattern as illustrated in Figure 3. Some predicates, the ones annotated with *@syntax*, specify the grammar of the object language and the constraints that need to be solved for each construct (production) in the language. The names of this predicates, *e.g.* `main`, `dcl`, `exp`, become the nonterminals in the attribute grammar and the patterns in the nested `match` construct specify the productions, *e.g.* `prog`, `add`, `cons`. The arguments annotated as *@syn* or *@inh* become synthesized and inherited attributes for that nonterminal respectively.

Most names introduced in the exists constraint within a syntactic case become local attributes in our translation. But *scope*-typed names are realized as nonterminal attributes that become the single-node tree representing a scope node in the scope graph when there is a `new` constraint for that scope in the body of the existential. The `mod` production on line 24 in Figure 4 shows a scope defined in this way, corresponding to the module scope being asserted on line 14 in Figure 3. These correspondences can be seen in the declarations of nonterminals `exp` and `dcl`, their attributes on line 7 and line 23, and productions `add`, `ref`, and `mod` in Figure 4. Function predicates are those without the *@syntax* annotation and these are more naturally translated into simple functions that can be called from attribute equations.

Scope assertions: A Statix `new` constraint translates to a nonterminal attribute definition whose right-hand side is `mkScope(`d`)`, where d is the data associated with the node, *e.g.* `sm'` on line 25 in Figure 4. The restriction on the location of `new` constraints in Statix discussed in Section 3.2 makes the translation much simpler, as it becomes easy to decide whether a constraint `{x:` *scope*`}` `C` corresponds to a local or nonterminal attribute. If there is a `new` $x \to t$ constraint in `C`, we use a nonterminal attribute. Otherwise the name simply refers to an already defined scope we want to hold a reference to, in which case we can use a local attribute.

Edge assertions: Scopes are implemented as tree nodes created as nonterminal attributes. We represent edges as reference attributes that occur on the *Scope* nonterminal. For LM, this nonterminal has four inherited edge attributes: `LEX`, `VAR`, `IMP`, and `MOD`, corresponding to the edge labels in Figure 1 and Figure 3. Their equations follow the definition of a scope nonterminal attribute, such as those for `sm'` on lines 26–27 in Figure 4.

In Statix, all locations where a new scope is passed are valid edge assertion sites. To capture this behavior in the attribute grammar translation, we say that any nonterminal with an inherited attribute `s` of type *scope* must have a synthesized attribute L`_s` of type [*scope*] for every edge label L in the specification. Edge assertions translate to contributions to these attributes, whose purpose is to carry references to edge targets as lists to the nonterminal attribute denoting the source. Contributions have the form `lst <- [item]` for list attributes. At compilation time, these contributions are collected in the attribute grammar implementation into a single equation, empty by default. Implicit contributions of a child node's instance of the same attribute are inserted if the attribute occurs on that child. Thus a Statix constraint `sa -[LAB]-> sb` is translated into the contribution `sa.LAB <- [sb]` if `sa` is locally defined, and into `top.LAB_sa <- [sb]` otherwise, where `top` is the left-hand side nonterminal collecting these edge assertions up the tree. We also use contributions for the `ok` attribute, as seen in Figure 4, which are combined by conjunction similarly, with default value `true`.

Another useful consequence of the "permission to extend" analysis in Statix [15] comes into play when considering how we translate edge assertions. The requirement says that edge assertions sourced at a particular scope may only occur under the exists constraint which defines that scope's name, with respect to predicate expansion. Considering our restriction that the `new` constraint for a scope must be directly under the exists constraint that asserts its name, we determine that only *synthesized* attributes are required to copy edge target references to a scope definition. In other words, the only valid locations of edge assertions in Statix correspond in our attribute grammar translation to nodes in the sub-tree rooted at the scope's nonterminal attribute definition site. Consequently references to edge targets need only travel up the syntax tree as members of these synthesized attributes.

Queries: In Statix, a query can only be solved when there are no weakly critical edges for it in the scope graph. The execution of the query itself does not result in the extension of the scope graph. It simply runs to completion when the scope graph has all requisite information present, and it is picked from the current constraint set.

We capture this behavior by specifying that every query depends on an auxiliary function `dwce` ("demand weakly critical edges"). This function takes as arguments the same start scope and path well-formedness regular expression as a query, and traverses the scope graph with respect to a derivative [4] of the query regular expression, so that all reference attributes corresponding to the targets of weakly critical edges for the query are demanded and evaluated. Consequently, those target scopes and edges are built into

the scope graph. Since our `query` equations depends on these `dwce` operations, the queries themselves do not demand that any scopes or edges are be built into the scope graph. Thus evaluating a query in the attribute grammar translation has the same effect as solving its Statix counterpart.

Predicate uses: Applications of syntax predicates correspond to the definition of inherited attributes on a syntax tree node, as well as demanding its synthesized attributes. The use of `expr(s, l, tyL)` on line 29 of Figure 3 corresponds to the left equations on lines 10–11 of Figure 4, recalling that all syntax nodes have the `ok` attribute. The inherited attribute equations correspond to what are given in `inh` argument positions in the predicate. On line 29 the first argument `s` of the `expr` predicate use says that the lexical scope of the left addition operand is the locally known `s` that is the addition's lexical scope. We correspond in Figure 4 on line 10 by defining `l`'s inherited `s` attribute as the addition node (`top`)'s attribute `s`. The arguments given in the `syn` argument positions correspond to demands of synthesized attributes from a tree node. In the addition example, `tyL` is used as the third argument on line 29 of Figure 3. Since `tyL` corresponds to a local attribute in the translation, we define that local as the value of synthesized attribute `l.ty` on line 11 of Figure 4.

A function predicate use in Statix is translated as function application in our attribute grammar. The term defined with the nonterminal attribute matches that of the Statix function predicate, although only containing its proper arguments. We see an example of a function predicate use on line 25 in Figure 3, where `tgt` is used in case `var` to determine the target scope of path `p`. This corresponds to the `tgt` function call on line 20 of Figure 4.

6 Scheduling AG equations and Statix constraints

This section formalizes the notion that a valid Statix schedule of constraint solving can be determined by an attribute grammar and that the results of each are the same.

6.1 Corresponding AG and Statix evaluation traces

Here we show that the demand-driven evaluation of an attribute grammar translated from a Statix specification corresponds to valid schedule of constraint solving, based on the operational semantics of each as given in Section 4.2 and Appendix A. We define a correspondence between states in the attribute grammar evaluation, $\langle t|stk;set\rangle$, and the Statix evaluation, $\langle G|\overline{C}\rangle$, and extend this to evaluation traces. We also provide a property stating that these correspondences hold for all attribute grammar evaluation traces. The correspondence relation

$$\langle t|stk;set\rangle \approx \langle G|\overline{C}\rangle$$

has two primary components: that the scope graph constructed in t corresponds to G and the constraints in $\overline{C}$ have corresponding equations or expressions in stk or set. The first is specified $sg(t) = \langle S_G, E_G, \rho_G\rangle$ where $sg(t)$ is the scope graph $\langle S_t, E_t, \rho_t\rangle$, and; S_t are all node identifiers for trees in t constructed by a `mkScope` production; E_t are all edges s_x `-[L]->` s_y such that $s_y \in s_x.L$ in the tree t, that is s_y is in the appropriate edge attribute on s_x; ρ_t consists of the mappings $s \mapsto d$ when s`.getData` $= d$ in the tree t.

The second requirement on $\overline{C}$ checks that each *new* $s_i \to d$ constraint in $\overline{C}$ has an equation s_i `= mkScope(`d`)` equation in stk or set; each s_x `-[`L`]->` s_y constraint corresponds to the expression s_y in an equation $s_x.L = [...s_y...]$; constraints such as *single* have equations with the corresponding attribute grammar function or predicate as its body; equality constraint t_1`==`t_2 corresponds to a subexpression t_1`==`t_2 in a conjunction defining an `ok` equation.

Informally, the correspondence $\approx$ says that the scope graph embedded in the tree in the attribute grammar state is the same as that in the Statix state, and each constraint in $\overline{C}_j$ corresponds to an equation or expression in stk_i or set_i. This holds for the initial attribute grammar evaluation state $\langle mkTree(p,0)|[ag_0.\texttt{ok} = root_1.\texttt{ok}];\varnothing\rangle$ and the corresponding initial Statix evaluation state $\langle\varnothing|\{\texttt{main}(p)\}\rangle$.

The correspondence between a partial or terminating trace of AG evaluation steps and Statix steps is denoted as $\approx^*$ and defined as:

$$\frac{\langle t_0|stk_0;set_0\rangle \to^* \langle t_{n'}|stk_{n'};set_{n'}\rangle \approx^* \langle G_0|\overline{C}_0\rangle \to^* \langle G'_m|\overline{C}'_m\rangle \quad n' = n-1 \quad m' \le m \quad \langle t_n|stk_n;set_n\rangle \approx \langle G_m|\overline{C}_m\rangle}{\langle t_0|stk_0;set_0\rangle \to^* \langle t_n|stk_n;set_n\rangle \approx^* \langle G_0|\overline{C}_0\rangle \to^* \langle G_m|\overline{C}_m\rangle}$$

The correspondence between traces accommodates steps in the attribute grammar evaluation that do not change the Statix state or correspond to a Statix rule; in this case $m' = m$. It also accommodates the case in which one attribute grammar step corresponds to more than one Statix steps; in this case $m' < m - 1$. Some attribute grammar steps do not correspond to a step in Statix; copying inherited scopes down the tree, *e.g.* line 10 of Figure 4 and the equation on line 28 collecting edges up the tree are examples of this. Another consideration is that there may be multiple Statix edge assertions that correspond to one attribute equation that represents the targets of those edges as a list. So stepping from $\langle t|s_3.VAR = [s_8, s_{13}] :: stk';set\rangle$ to $\langle t'|stk';set\rangle$ where $s_3.VAR = [s_8, s_{13}] \in t'$, corresponds in Statix to the transition $\langle G|s_3$ `-[VAR]->` $s_8;\overline{C}\rangle \xrightarrow{\text{Op-Edge}} \langle G|s_3$ `-[VAR]->` $s_{13};\overline{C}\rangle \xrightarrow{\text{Op-Edge}} \langle G|\overline{C}\rangle$. In this case, if the first attribute grammar state above corresponds to the first Statix state, then the second attribute grammar state corresponds to the last Statix state above. But during this evaluation there is no attribute grammar state that corresponds to the second Statix state, where only one of the edges has been built into the scope graph.

Theorem 1 uses this correspondence to state that for any partial attribute grammar evaluation trace beginning at the

initial state, there is a corresponding Statix evaluation trace. We later use this result to establish that the attribute grammar and Statix evaluations produce the same results.

Theorem 1. For a program p (with start symbol of *main*) with the partial or terminating trace

$$\langle t_0 | stk_0; set_0 \rangle \rightarrow^* \langle t_n | stk_n; set_n \rangle$$

where $t_0 = mkTree(p, m_0)$, $stk_0 = m_0.ok = e$ and $set_0 = \emptyset$, there exists

$$\langle G_0 | \overline{C}_0 \rangle \rightarrow^* \langle G_m | \overline{C}_m \rangle$$

where $G_0 = \langle \emptyset, \emptyset, \emptyset \rangle$ and $\overline{C}_0 = \{main(p)\}$ such that

$$\langle t_0 | stk_0; set_0 \rangle \rightarrow^* \langle t_n | stk_n; set_n \rangle \approx^* \langle G_0 | \overline{C}_0 \rangle \rightarrow^* \langle G_m | \overline{C}_m \rangle.$$

Theorem 1 is can be proved by induction on the length of the attribute grammar evaluation trace $\langle t_0 | stk_0; set_0 \rangle \rightarrow^* \langle t_n | stk_n; set_n \rangle$; we provide a partial sketch of that proof below. Each equation step taken to extend that trace will add zero or more states in the corresponding Statix trace $\langle G_0 | \overline{C}_0 \rangle \rightarrow^* \langle G_m | \overline{C}_m \rangle$ ensuring that $\langle t_n | stk_n; set_n \rangle \approx \langle G_m | \overline{C}_m \rangle$. We consider the most relevant attribute grammar evaluation rules and describe how the correspondence is maintained.

Building scopes: In state $\langle t | n = \texttt{mkScope}(d) :: stk; set \rangle$, rule CompNTA steps to a state that corresponds to the Statix state after solving a `new` s_n constraint, where n corresponds to s_n, by applying Op-Node. If s_n is the name for the new tree created by CompNTA, d is a data term that becomes the sole child of node s_n. On the Statix side, we use s_n as the scope identifier in the Op-Node rule to add a new scope to G in the corresponding Statix state and update ρ with $s_n \mapsto d$. The `new` s_n constraint is removed from the Statix constraint set, and equation $s_n = \texttt{mkScope}(d)$ is popped from the attribute grammar stack. Thus correspondence $\approx^*$ is maintained.

Building edges: In state $\langle t | s_n.LAB = [s_1, ..., s_m] :: stk; set \rangle$ where all s_i's are scope node references, we apply the CompAttr rule to complete the attribute instance $s_n.LAB$ with value $[s_1, ..., s_m]$. In our representation of scope graphs, this builds m new edges of label LAB and targets $s_1, ..., s_m$. We follow suit in Statix by identifying the m edge assertions whose source is s_n, label is LAB and target is one of $s_1, ..., s_m$. This entails several applications of the Op-Edge Statix rule. Thus the state resulting from one step in the attribute grammar corresponds to the state we get in Statix after solving the m edge assertions. Correspondence is maintained, as in the attribute grammar scope graph we built edges from s_n to $s_1, ..., s_m$, just as we did in Statix by applications of Op-Edge.

Executing queries: In the attribute grammar state $\langle t | n.a = \texttt{query}(dwce,\ s,\ r,\ D) :: stk; set \rangle$ we step the function application using rule Apply. A number of rules in Figure 5 and Figure 6 will then be applied until we have $n.a = v$ atop the stack, where v is the result of the query. We then use rule CompAttr to put $n.a = v$ in the tree, marking $n.a$ as a complete attribute instance. In Statix we apply rule Op-Query to execute the query. In so doing, the result of the query, that is equal to v, is substituted for name x in the remaining constraint set. The work done in the attribute grammar between Apply and CompAttr for $n.a$ is not interesting work, as use of `dwce` ensures that before the query is executed, all of the scope graph that it can traverse has been built. Thus correspondence is maintained between the use of Apply and CompAttr. Each instance of variable x we had in the Statix constraint set corresponded to $n.a$. Since, $n.a$ is now complete with the same query result, the terms set which replaced x in Statix corresponds to v.

Singleton: If we have $\langle t | n.a = \texttt{single}(ts) :: stk; set \rangle$ as the attribute grammar state, we have $\langle G | \texttt{single}(ts', x); \overline{C} \rangle$ in Statix, where value ts corresponds to ts' and, and x is a term variable that corresponds to $n.a$. In the attribute grammar we apply singleton using Apply. This results in a new state $\langle t | n.a = \texttt{case}\ ts\ \texttt{of}\ [x]\ \texttt{->}\ x\ \texttt{end} :: stk; set \rangle$. There are now two cases to consider: **When ts is a singleton**: In the attribute grammar, when $ts = \{tm\}$, we step to: $\langle t | n.a = tm :: stk; set \rangle$. In Statix there is a requirement that tm is ground, ensured by the attribute grammar stepping tm as an expression until it is a value. The equation is then completed and the value is stored in the tree; this corresponds to the solving of the $\texttt{single}(ts', x)$ constraint in Statix using Op-Single-True. Both the equation and the constraint are the removed from their respective states and the correspondence is maintained. **When ts is not a singleton**: Due to case matching failure, the state $\langle t | n.a = \texttt{case}\ ts\ \texttt{of}\ [x]\ \texttt{->}\ x\ \texttt{end} :: stk; set \rangle$ steps to $\langle t | n.a = \texttt{abort} :: stk; set \rangle$. The Abort rule then steps to the final attribute grammar state $\langle t | \texttt{abort}; \emptyset \rangle$. This corresponds to using the Op-Single-False rule to step the Statix state to $\langle G | \{\texttt{false}\} \rangle$. At this point the correspondence between states holds. Attribute grammar rules that correspond to solving the the `min`, `inhabited` constraints follow the same pattern as described above except that they will not abort.

Type equality: Consider checking the constraint `tyL == INT()` on line 29 of Figure 3 when `tyL` has been instantiated to `INT()`. This corresponds to the AG state $\langle t | m.ok = \texttt{INT()} == \texttt{INT()}\ \&\&\ e :: stk; set \rangle$ and the Statix state $\langle G | \texttt{INT()} == \texttt{INT()}; \overline{C} \rangle$ as our Statix state. This equality check in Statix was mapped to an attribute contribution to `top.ok` on line 11 in Figure 4 and then all such contributions are folded into a single nested conjunction in an equation expression. Then in the attribute grammar we use rules ConjLeftStep followed by Eq-True to transition from $\langle t | m.b = \texttt{INT()} == \texttt{INT()} \&\& e :: stk; set \rangle$ in two steps to $\langle t | m.b = \texttt{true}\ \&\&\ e :: stk; set \rangle$. Over in Statix we have state $\langle G | \texttt{INT()} == \texttt{INT()}; \overline{C} \rangle$ and can step immediately to $\langle G | \overline{C} \rangle$ using rule Op-Eq-True when we fire the Eq-True for expressions.

Predicate expansion: In AG state $\langle t | n.a = ...m.b... :: stk; set \rangle$ where we are ready to demand $m.b$, and m is a tree of nonterminal type nt that has been constructed but not yet visited. This corresponds to the Statix state $\langle G | nt(..., m', ...); \overline{C} \rangle$,

where m' is the syntactic term corresponding to the tree m; *e.g.*, the constraint `exp(s, r, tyR)` on line 30. The DMD-UNVISITED rule causes the equations in that production to be instantiated and added to the equation set, with the exception of the equation defining $m.b$ which is pushed on to the stack. If a `ref` production built `r` then `r`'s equations are instantiated for that node in this process. In STATIX this corresponds to expanding the corresponding syntax predicate using rule OP-PREDICATE, solving the immediate `match`, solving the immediate exists constraint with rule OPEXISTS, and finally expanding any conjunctions. The result is that the constraint set now includes all constraints from the production-case corresponding to m', instantiated with fresh names that all correspond to local attributes of node m.

6.2 Equivalent results

The correspondence $\approx^*$ of AG and STATIX evaluation traces ensures that they produce the same scope graph and results for name and type-analysis. The stack and `ok` attribute in the AG state imply ($\supset$) certain corresponding conditions in the STATIX state. This is stated in the following theorem.

Theorem 2. For an attribute grammar generated from a STATIX specification without spurious constraints (see below) and for a program p we have the terminating attribute grammar trace, meaning no more steps can be made,

$$\langle mkTree(p,0)|r_0.ok = e; \varnothing\rangle \rightarrow^* \langle t|stk; set\rangle$$

and this trace corresponds, via $\approx^*$, to

$$\langle\langle\varnothing,\varnothing,\varnothing\rangle|\{main(p)\}\rangle \rightarrow^* \langle G|\overline{C}\rangle$$

then

1. $stk = \square \supset r_0.ok = true \supset sg(t) = G \wedge \overline{C} = \varnothing$, or
2. $stk = \square \supset r_0.ok = false \supset sg(t) = G \wedge \overline{C} = \{false\}$, or
3. $stk =$ `abort` $\supset sg(t) = G \wedge \overline{C} = \{false\}$, or
4. $stk \neq$ `abort` $\wedge\ stk \neq \square \supset sg(t) = G$.

The first case states that if the final stack is empty and the `ok` attribute is true, then the scope graphs are the same and there are no STATIX constraints left to solve, indicating success on the STATIX side. The second and third cases are similar except for the failure cases. In these cases the contents of the equation *set* do not matter. The fourth case is when there was a cycle in the attribute grammar evaluation, and the STATIX specification was "stuck" due to a missing weakly critical edge or other mutually dependent constraints. In the given semantics, a cycle amounts to getting "stuck" on DMD-VISITED since it will not find the being demanded equation (the second time) in the set because it is already in the stack. A rule could be added to raise a `cycle` error, similar to the `abort` if an explicit error was desired. These four cases follow from the correspondence in Theorem 1.

Not taken into account in Theorem 2 is the case when a STATIX specification has constraints that do not need to be solved to generate the scope graph or establish program correctness. These correspond to equations in the AG that were never demanded in the evaluation process. These are not interesting with respect to using attribute evaluation to discover a valid schedule for constraint solving, and they can be eliminated with a alternative translation in which every constraint generates equations that contribute to the `ok` attribute, thus ensuring that they are all evaluated.

7 Discussion

The attribute grammar generated by the translation in Section 5 is meant to mimic the evaluation of STATIX constraints, but it is not written in a style one would likely see in an AG, even one using scope graphs. Notably, a hand-written AG would not use `dwce` to demand all the relevant scope graph nodes and edges be built before applying a query. Instead, the query function can be the computation that *demands* the construction of those scopes and edges, effectively interleaving the STATIX constraints that do this with the query constraint. The query could also be written to return only the *visible* declarations found, instead of all reachable ones, and then filter out those shadowed by other declarations. This shows the difference between the *eager* evaluation in STATIX and the *lazy* scheduling of equations in AGs.

All the "bookkeeping" steps in attribute grammar evaluation that correspond to equations being solved but do not have a corresponding step in the STATIX evaluation can be seen as evidence of the conciseness of STATIX specifications (like those in Figure 3) versus the more verbose specifications for attribute grammars (*e.g.* those in Figure 4). The attribute grammar specifications require equations to pass a scope down the tree in inherited attributes, and synthesized attributes to pull edge contributions up. The substitution of an identifier created by solving an *exists* constraint into all the constraints in its body corresponds to all those inherited attribute equations that pass a information down the tree.

We experimentally validated some aspects of Theorem 2 in an implementation artifact that accompanies the paper. It consists of a SILVER implementation of our annotated STATIX language that translates these specifications to (i) plain STATIX that can be run with Rouvoet et al.'s MiniStatix system [15] (ii) to the operational semantics of Section 4.2 implemented in OCaml, and (iii) to a SILVER attribute grammar specification. We have written specifications for 3 versions of LM implementing sequential, parallel, and self-referential/recursive import semantics; all three have the same syntax, just different import semantics. We also have several sample LM programs that can then be run through all 3 implementations of all the 3 different import semantics. The artifact checks that the three implementations always terminate with the corresponding MiniStatix satisfiability/stuck and AG `ok`/`abort`/cycle results. That is, Theorem 2 is verified with the exception of automatically checking that the scope graphs are the same, but manual

inspections do show that these are the same. This includes specifications for import semantics that lead to Statix getting "stuck" and to the two AG systems exhibiting cycles in attribute evaluation. It is worth noting that the lazy evaluation of expressions in attribute equations in Silver gives the same results as the eager evaluation of expressions in the OCaml implementations. It is only the laziness in scheduling the evaluation of attribute equations that matters.

8 Related Work

Visser and his colleagues have written a serious of papers refining and extending their initial notion of scope graphs [14] to include edge labels [15, 17] to describe relationships like lexical enclosement and name declarations in a scope. Name resolution maps references to the most visible declarations, with respect to a language-wide path ordering. Later work [1, 17] introduced constraint-solving as a means of performing name and type analysis of programs based on scope graphs.

Poulsen et al. [2] introduce a monadic interface for phased name resolution in Mophasco, which provides generic effectful operations for carrying out name resolution using scope graphs. Inspired by Gibbons et al. [8], Mophasco makes use of applicative functors to compositionally map abstract syntax trees onto type checking constraints defined as phased monadic computations. This yields an approach which can support language features which Statix cannot. In particular, it lends support to the unordered import semantics of the original LM definition by Neron et al. [14].

Kastens and Waite [11] provide an alternate scope graphs model for name analysis, implemented in attribute grammars. The structure of their scope graphs differs in some respects from Visser et al.'s. It categorizes edges into two kinds; *parent* edges and *path* edges. The former corresponding to our use of the `LEX` edge label, and the latter being labeled by positive integers associated with particular relations. While attribute grammars are used in the implementation they do not rely on demand-driven evaluation and thus must maintain a worklist for name analysis to schedule various resolution tasks.

Reference attribute grammars support a number of different styles for solving name resolution problems. A common approach uses and environment as an inherited attribute that contains binding of all names in scope to their type or other relevant information. References are especially useful in this still when the names are bound to references back to the declaration node for the name. Once the name is looked up in the environment, any attributes, such as type or location, can be immediately accessed. Another style is closer to that of scope graphs. Here the inherited attribute is a reference to the syntax tree node defining the enclosing scope. This node can maintain a list of declarations in it that can be search when looking for a declaration. The node can also maintain a reference to its enclosing parent or imported scopes so that those can also be queried if need be. In this approach the scope graph is very much present in the edges superimposed over the tree. Visser et al. might claim these approaches are one that are more ad-hoc and more closely tied to the language specification than the scope graph and constraint solving approach they have proposed.

9 Future Work and Conclusion

The use of demand-driven evaluation of reference attribute grammars to schedule the construction and analysis of scope graphs opens some interesting avenues of future work. One is to extend the Statix translator support Statix specifications which are not as restricted as discussed Section 3.2. For instance, to what extent can the *@syntax*, *@syn*, *@inh*, and type annotations could be inferred? It would also be possible to restore Statix unification with a more sophisticated translation. This could lead to a more user-friendly version of Statix that is closer to the original, even though our formulation is sufficient for our goals.

Since a cycle in the attribute grammar can correspond to Statix getting stuck on missing weakly critical edges, a natural next step would be to translate to circular attribute grammars [7, 13, 16]. These cycles arise when `import` declaration are not interpreted in a sequential manner as they are in the version of LM discussed here. More interesting are the import models in which the import order is unordered, or when imports can be *self-influencing*. This later is seen the Rust language where an import declaration may bring into scope new names that cause that import resolution to not be stable and to then "re-import" based on the new names. These models are noted as being ones that Statix cannot solve due to constraints becoming stuck [2, 15]. We have begun experimenting with this idea and have a prototype that translates Statix specifications into the JastAdd system (that includes both circular and reference attributes [13]) that has shown promising initial results.

We are also investigating the use of scope graphs in the Silver attribute grammar specifications for (domain-specific) languages through a library of commonly used constructs, such as the `mkScope` production, and an extension to Silver for writing Statix-like constraints directly instead of the equations that constraints translate to as seen in Section 5.

To conclude, this paper formalizes what has been the folklore in the attribute grammar community for some time, that scope graphs are naturally specified in reference attributes grammars. The translation of Statix specifications into reference AGs also illustrates the differences between the two approaches and suggests future work that may tie them both closer together.

Acknowledgments

This work was supported in part by the National Science Foundation, award #2123987. We thank the SLE reviews for the helpful comments that improved this paper.

References

[1] Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/2847538.2847543

[2] 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

[3] John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687. doi:10.1145/1082036.1082042

[4] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494.

[5] Torbjörn Ekman and Görel Hedin. 2004. Rewritable Reference Attributed Grammars.. In *Proceedings of the 18th European Confernece on Object Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 3086)*. Springer, Berlin, Heidelberg, 144–169. doi:10.1007/978-3-540-24851-4_7

[6] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69 (December 2007), 14–26. Issue 1-3. doi:10.1016/j.scico.2007.02.003

[7] 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

[8] 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.

[9] Görel Hedin. 2000. Reference Attribute Grammars. *Informatica* 24, 3 (2000), 301–317.

[10] 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

[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] 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).

[14] 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

[15] 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

[16] 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.

[17] 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

[18] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54. doi:10.1016/j.scico.2009.07.004

[19] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 131–145. doi:10.1145/73141.74830

[20] 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

A Statix operational semantics

This appendix contains many of the rules defining the operational semantics of Statix, with our small modifications. These closely follow the original rules [15].

$\boxed{\langle G|\overline{C}\rangle \to \langle G|\overline{C}\rangle}$

$$\frac{}{\langle G|\texttt{true};\overline{C}\rangle \to \langle G|\overline{C}\rangle}\ \textsc{Op-True}$$

$$\frac{}{\langle G|\texttt{false};\overline{C}\rangle \to \langle G|\{\texttt{false}\}\rangle}\ \textsc{Op-False}$$

$$\frac{t\ ground}{\langle G|x\texttt{:=}t;\overline{C}\rangle \to \langle G|[t/x]\overline{C}\rangle}\ \textsc{Op-Define}$$

$$\frac{t_1\ ground \qquad t_2\ ground \qquad t_1 = t_2}{\langle G|t_1\texttt{==}t_2;\overline{C}\rangle \to \langle G|\overline{C}\rangle}\ \textsc{Op-Eq-True}$$

$$\frac{t_1\ ground \qquad t_2\ ground \qquad t_1 \neq t_2}{\langle G|t_1\texttt{==}t_2;\overline{C}\rangle \to \langle G|\texttt{false};\overline{C}\rangle}\ \textsc{Op-Eq-False}$$

$$\langle G|(C_1, C_2);\overline{C}\rangle \to \langle G|C_1; C_2;\overline{C}\rangle\ \textsc{Op-Conj}$$

$$\frac{t\ ground}{\langle G|\texttt{single}(\{t\}, x);\overline{C}\rangle \to \langle G|[t/x]\overline{C}\rangle}\ \textsc{Op-Single-True}$$

$$\frac{\nexists t.\ ts = \{t\}}{\langle G|\texttt{single}(ts, x);\overline{C}\rangle \to \langle G|\texttt{false};\overline{C}\rangle}\ \textsc{Op-Single-False}$$

$$\frac{C'' = [id_1/x_1]...[id_n/x_n]C' \qquad id_i\ fresh\ ids}{\langle G|\{x_1, ..., x_n\}C';\overline{C}\rangle \to \langle G|C'';\overline{C}\rangle}\ \textsc{OpExists}$$

$$\frac{p(x_1, ..., x_n)\texttt{:-}\ C. \qquad C' = [t_1/x_1]...[t_n/x_n]C}{\langle G|p(t_1, ..., t_n);\overline{C}\rangle \to \langle G|C';\overline{C}\rangle}\ \textsc{Op-Predicate}$$

$$\frac{s \notin S_G}{\langle\langle S, E, \rho\rangle|\texttt{new}\ s \to t;\overline{C}\rangle \to \langle\langle s; S, E, [s \mapsto t]\rho\rangle|[s/x]\overline{C}\rangle}\ \textsc{Op-Node}$$

$$\frac{s_1, s_2 \in S_G}{\langle\langle S, E, \rho\rangle|s_1 - l \to s_2;\overline{C}\rangle \to \langle\langle S, (s_1, l, s_2); E, \rho\rangle|\overline{C}\rangle}\ \textsc{Op-Edge}$$

$$\frac{\rho(s) = t}{\langle G|\texttt{getData}(s, x);\overline{C}\rangle \to \langle G|[t/x]\overline{C}\rangle}\ \textsc{Op-Data}$$

$$\frac{no\ weakly\ critical\ edges}{\langle G|\texttt{query}(s, r, x);\overline{C}\rangle \to \langle G|[Ans(G, s, r)/x]\overline{C}\rangle}\ \textsc{Op-Query}$$