Tool Paper: AGTix: Concise Use of Scope Graphs in Reference Attribute Grammars

Luke Bessant
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, Minnesota, USA
bessa028@umn.edu

Eric Van Wyk
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, Minnesota, USA
evw@umn.edu

Abstract

This tool paper introduces AGTix, an extension to reference attribute grammars that allows constraints, inspired by Visser et al.'s Statix system, that construct and query scope graphs to be written alongside traditional attribute equations. Previous work has shown that a restricted form of Statix specifications can be translated to an equivalent reference attribute grammar that produces the same results; however, the resulting attribute grammar is quite verbose. In AGTix specifications, the more concise constraints for specifying new scope nodes or edges are translated down to the verbose equations which propagate scope node and edge information up and down the syntax tree. The query constructs for resolving names are translated down to expressions; as in Statix, these include syntax for regular expressions over edge labels that describe the valid paths for a resolution. The scope graphs and query results described by these new constructs can be used in traditional attribute grammar equations for other language processing tasks such as collecting error and warning messages or using variable types to support a translation to another language. AGTix is implemented as an extension to Silver, a full-featured extensible attribute grammar system, and is evaluated by implementing several variations of the LM language introduced by Visser et al.

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

Keywords: reference attribute grammars, scope graphs, name resolution, language specifications

ACM Reference Format:
Luke Bessant and Eric Van Wyk. 2026. Tool Paper: AGTix: Concise Use of Scope Graphs in Reference Attribute Grammars. In *19th ACM SIGPLAN International Conference on Software Language Engineering (SLE '26), July 02–03, 2026, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3806383.3815522

SLE '26, Rennes, France

ACM ISBN 979-8-4007-2639-2/2026/07
https://doi.org/10.1145/3806383.3815522

1 Introduction

Language specifications for defining the syntax and semantics of software languages benefit from being declarative, concise, amenable to analysis, and applicable to differing styles of languages. A wide variety of specification styles and supporting tools for generating language implementations from those specifications have been developed and they vary in many different ways. Of interest here is the degree to which the meta-languages, in which object language specifications are written, are tailored to specific language processing tasks over the abstract syntax of a program.

Attribute grammars [Knuth 1968] (AGs) specify semantics for context free grammars by associating attributes with nonterminals, and equations associated with productions for computing their values. In this work we focus on *reference attribute grammars* [Boyland 2005; Hedin 2000] (RAGs) that allow attribute values to be references to nodes in the abstract syntax tree (AST) of a program, so that graph-like structures can be created. These can be used to specify language processing tasks and support general computations over structured data such as ASTs.

Other approaches are designed for specific kinds of language processing tasks, and thus allow for quite concise specification of those tasks. Term-rewriting system specifications [Baader and Nipkow 1998] take the form of rewrite rules that match patterns in an AST and replace them with others. These are especially useful in transformation or translation tasks but less so for tasks like type checking or other semantic analyses. The successful Stratego [Visser 2001] system pioneered the use of strategies to control the order and application of rewrites rules. Some systems, such as Spoofax [Kats and Visser 2010], are a combination of domain-specific languages (DSLs), each directed at particular tasks. Stratego is a DSL in Spoofax; another is Statix, a constraint solving system for defining the scoping structure of a program or model as a *scope graph* [Néron et al. 2015; Rouvoet et al. 2020; van Antwerpen et al. 2018], and solving generic queries over that structure to resolve name uses.

This paper introduces AGTix, an extension to RAGs that allows concise Statix-like constraints for building and querying scope graphs to be written directly as part of an RAG specification. One then gets the task-specific features of Statix

along with the more general-purpose tree-computation features in attribute grammars in one meta-language. These new AGTix constructs significantly reduce the boilerplate code the language engineer must write to use scope graphs in reference attribute grammars yielding specifications that more closely align to concise Statix specifications.

2 Scope Graphs

Scope graphs [Néron et al. 2015] are a representation of name binding structures, containing nodes corresponding to scopes and declarations in a program, and edges defining relationships between them. Declaration nodes are distinguished by associating with them object language-specific data such as a name and type. Figure 1 shows an example scope graph representing sequential binding semantics for the program on the left, written in LM, defined by Neron et al. [Néron et al. 2015]. In this illustration we associate variable and module declaration nodes with the names of the corresponding program declarations, and variables with their type, *e.g.* node D_3 represents the variable declaration $\mathtt{b}_3$ on line 3. Nodes with no data represent scoping regions; either the global scope S_G, or S_i for a scope enclosing a statement on line i.

Directed edges in a scope graph define relationships between nodes, identified by a label taken from a language-specific set. The labels defined for LM and used in Figure 1 are `LEX`, `VAR`, `MOD` and `IMP`. These are for lexical parenthood, variable declaration, module declaration, and imports respectively. Certain edges *e.g.* `LEX`, `VAR` and `MOD` are drawn from syntactic analysis of a program. Others like `IMP` must be derived by name resolution.

Name resolution is performed by queries that compute valid paths from a start scope, that respect a path well-formedness predicate over edge labels. A query of import $\mathtt{a}_{11}$ on line 11 of Figure 1 may follow any number of `LEX` edges, followed by an optional `IMP` edge and ending in a `MOD` edge to compute a path reaching D_1. Then an `IMP` edge is created from S_{11}, the scope affected by the import, to D_1. We may then resolve $\mathtt{b}_{13}$ from S_{11}, which computes path $S_{11} \xrightarrow{-\mathtt{IMP}} D_1 \xrightarrow{-\mathtt{VAR}} D_3$. If $\mathtt{b}_{13}$ was resolved first, without the `IMP` edge, no valid path to a declaration would be found. A significant focus of previous work [Bach Poulsen et al. 2023; Bessant and Van Wyk 2025; Rouvoet et al. 2020] has been to identify means of correctly interleaving queries and graph extensions so that name analysis performs as intended.

3 Statix

Statix [van Antwerpen et al. 2018] is a constraint-solving DSL for specifying name-resolution and the static semantics of languages. Statix specifications contain user-defined predicates that identify the abstract syntax of the object language, assertions to construct scope graph nodes and edges, constraints specifying queries over scope graphs, and other

```
1  module a1
2  {
3    def b3 =
4      let x4 = 1.0
5          y5 = 3
6       in x6 + y6
7  }
8
9  module b9
10 {
11   import a11
12   def a12 =
13     b13 + 1
14 }
```

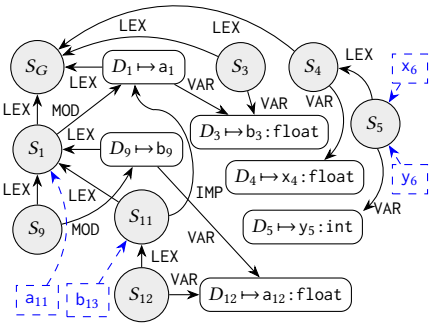

Figure 1. An LM program and scope graph under sequential binding with subscripting of name instances by line number

generic data-processing constraints. Figure 2 shows a partial specification of LM, written in MiniStatix, an implementation of core Statix defined by Rouvoet et al. [Rouvoet et al. 2020]. This includes predicates which operate over abstract syntax, *e.g.* `expr` on lines 1–19 matching on syntactic argument `e`. Each case contains constraints that operate on children of the expression, parameters of the predicate, and names introduced by existential constraints, written as $\{id_1, ..., id_n\}$ *body*, such as `tyL` and `tyR` on line 3. These predicates may have additional arguments for propagating information through the program's syntax tree. Importantly, scope graph nodes can be provided as arguments for use in name resolution or graph extension. Argument `s` on line 1 is the scope of an expression from which names are resolved on line 13. Variables received as arguments may also be instantiated within the predicate, *e.g.* argument `ty` to `expr` on line 2 and `s_last` on line 22.

Statix also has constraints that, when solved, create new nodes and edges in the scope graph. The `new` constraint constructs a scope graph node and optionally associates data with it. On lines 22 and 26 of Figure 2 we instantiate scope nodes with no data attached, whereas line 32 constructs a declaration node and maps it to a variable's name and type. Solving a `new` constraint updates the scope graph with the new node. Since arguments of a predicate may be instantiated in its body, names for scope graph nodes can be asserted in the body of one predicate and instantiated in another. Name `s_let` is introduced on line 10 but the scope itself is built on line 22 in `seqBinds`. Solving edge constraints *e.g.* on line 23 and line 27 adds the corresponding edges to the scope graph. Each `query` constraint computes the declaration of a name reference. The query on line 13 starts at the scope of the variable use and instantiates `vs` as a set reachable declaration nodes, with respect to a path well-formedness regular expression over edge labels. The constraints on lines 14–16 filter the declarations to include only those with the name `x` and then find the minimal set of visible declaration nodes, with respect to a path edge label ordering *e.g.* `'VAR < 'LEX` prioritizes resolutions found by following `'VAR` edges over follow `'LEX` edges.

```
1  expr(s, e, ty) :- e match
2    { ExprInt(_) -> ty == TInt()
3    | ExprAdd(e1, e2) -> {tyL, tyR}
4        expr(s, e1, tyL), expr(s, e2, tyR),
5        tyL match { INT() -> true | FLOAT() -> true
6                  | _ -> false},
7        tyR match { INT() -> true | FLOAT() -> true
8                  | _ -> false},
9        castAdd(tyL, tyR, ty)
10   | ExprLet(bs, e) -> {s_let}
11       seqBinds(s, bs, s_let), expr(s_let, e, ty)
12   | ExprVar(x) -> {vs, vs', vs'', p, s_tgt}
13       query s `LEX*`IMP?`VAR as vs,
14       filter vs (DatumVar(x', _) where x' == x) vs',
15       min vs' lexico(`VAR < `LEX, `IMP < `LEX,
16                      `VAR < `IMP) vs'',
17       only(vs'', p), tgt(p, s_tgt),
18       datumOf(s_tgt, DatumVar(x, ty))
19   | ... }.
20 seqBinds(s, bs, s_last) :- bs match
21   { SeqBindsLast(b) -> {s_dcl}
22       new s_last,
23       s_last -[ `LEX ]-> s, s_last -[ `VAR ]-> s_dcl,
24       bind(s, b, s_dcl)
25   | SeqBindsCons(b, bs) -> {s_next, s_dcl}
26       new s_next,
27       s_next -[ `LEX ]-> s, s_next -[ `VAR ]-> s_dcl,
28       bind(s, b, s_dcl),
29       seqBinds(s_next, bs, s_last) }.
30 bind(s, b, s_dcl) :- b match
31   { Bind(x, e) -> {ty}
32       new s_dcl -> DatumVar(x, ty),
33       expr(s, e, ty) }.
```

Figure 2. Expression and sequential bind tree rules for LM in MiniStatix

Statix includes other general purpose data processing constraints, *e.g.* the equality on line 2, uses of `match` constraints on lines 5–7 for pattern matching, and `only` on line 17 asserting that a set is a singleton.

4 AGTix: RAGS + Statix

AGTix is an extension to *reference attribute grammars* [Boyland 2005; Hedin 2000] that allows specifications to construct and query scope graphs using Statix-like constraints. This builds on work by Bessant and Van Wyk [2025], which showed that scope graphs have a natural implementation in RAGs by defining a translation of complete Statix specifications to corresponding RAG specifications, and proving that the resulting RAGs yield the same results as their Statix counterparts. AGTix is implemented as an extension to Silver [Van Wyk et al. 2010], so that the new Statix-like constructs are translated down to core RAG equations following Bessant and Van Wyk. Here we provide a brief introduction to RAGs, then discuss the new features AGTix introduces and their implementation.

4.1 Attribute Grammars

Attribute grammars [Knuth 1968] augment context-free grammars with attributes that decorate tree nodes. The grammar contains nonterminal declarations such as those on lines 1–2 of Figure 3, and productions such as those defined for nonterminal `Expr` on lines 1, 18 and 42 of Figure 4 for constructing, addition, variable reference, and let expressions respectively.

```
1  nonterminal Expr; nonterminal Type;
2  nonterminal Bind; nonterminal Binds;
3
4  synthesized attribute errs::[String];
5  synthesized attribute ocaml::String;
6  inherited attribute inSeqLet::Boolean;
7  attribute errs, ocaml occurs on Expr, Binds, Bind;
8  attribute inSeqLet occurs on Bind;
9
10 synthesized attribute type::Type occurs on Expr, Bind;
11
12 scope labels lex, var, mod, imp as LMLabels;
13 scope attribute s occurs on Expr, Binds, Bind;
14 scope attribute s_last occurs on Binds;
15 scope attribute s_dcl occurs on Bind;
```

Figure 3. AGTix LM nonterminal and attribute definitions

Declarations on lines 4–6 of Figure 3 introduce *synthesized* and *inherited* attributes, and the occurrence declarations on lines 7–8 identify which nodes those attributes decorate. Synthesized attributes on a node are defined by equations associated with the production for that node, *e.g.* `errs` and `ocaml` are defined for `exprAdd` nodes by the equations on lines 7 and 8 of Figure 4. Inherited attributes are defined by parent nodes, *e.g.* the definition of attribute `inSeqLet` for child nodes `b` of type `Bind` on lines 65 and 55 indicate to the child node that it is in a sequential let binding and not a recursive let binding, whose productions are elided. Productions can also define *local* attributes that are only visible in the production, *e.g.* `msgLeft` and `msgRight` on lines 3–5.

Attributes values can also be undecorated syntax trees whose type is a nonterminal, these are called *higher-order* attributes [Vogt et al. 1989]. On line 10 of Figure 3 we declare a synthesized higher-order attribute `type` of nonterminal `Type`, whose productions are omitted. Line 73 of Figure 4 defines `type` for `bind` nodes. Pattern matching on types is used in the `exprAdd` production on lines 8–16 to generate the correct integer or floating point addition operator in the translation to OCaml in the `ocaml` attribute. *Reference* attributes [Hedin 2000] do not hold trees, but pointers to possibly distant nodes in the tree or tree nodes created during attribute evaluation, allowing the access of attributes on them. We use these to create graph structures over trees, particularly useful in name resolution as they allow variable uses to store references to declaration nodes in the AST. The implementation of AGTix as described below uses reference attributes in this way.

4.2 AGTix

AGTix introduces new syntax for defining attributes related to scope graphs, and equation forms which resemble the Statix constraints for building scope graph nodes and edges, and executing queries. The set of edge labels that will appear in a scope graph is declared using the `scope labels` construct, as on line 12 of Figure 3, also introducing a name `LMLabels` for this set. AGTix allows the definition of a `scope attribute` which acts as an inherited attribute for passing scopes down syntax trees, corresponding to Statix predicate

```
1  production exprAdd top::Expr ::= e1::Expr e2::Expr
2  { e1.s = top.s; e2.s = top.s;
3    local msgLeft::[String] = assert(addable(e1.type),
4      "(+) LHS type not int/float, is " ++ e1.type.pp);
5    local msgRight::[String] = assert(addable(e2.type),
6      "(+) RHS type not int/float, is " ++ e2.type.pp);
7    top.errs = msgLeft ++ msgRight ++ e1.errs ++ e2.errs;
8    top.ocaml = "(" ++
9      case e1.type, e2.type of
10     | tFloat(), tFloat() -> e1.ocaml ++ " +. " ++ e2.ocaml
11     | tFloat(), tInt() ->
12       e1.ocaml ++ " +. (float_of_int " ++ e2.ocaml ++ ")"
13     | tInt(), tFloat() ->
14       "(float_of_int " ++ e1.ocaml ++ ") +. " ++ e2.ocaml
15     | _, _ -> e1.ocaml ++ " + " ++ e2.ocaml
16     end ++ ")";
17   top.type = castAdd(e1.type, e2.type); }
18 production exprVar top::Expr ::= x::String
19 { local exact::[Scope] =
20     query(`lex*`imp?`var,
21           `lex > `imp, `lex > `var, `imp > `var,
22           isVarCalled(x), top.s);
23   local close::[Scope] =
24     query(`lex*imp?(`var|`mod),
25           `lex > `imp, `lex > `var, `lex > mod,
26           `imp > `var, `imp > `mod,
27           editDistanceAtMost(1, x), top.s);
28   local bindNode::Decorated Bind with {s, inSeqLet} =
29     if singleton(exact) then getBind(head(exact))
30                         else defaultErrBind;
31   local closeNames::[String] = getDclNames(close);
32   top.errs = case exact, closeNames of
33              | [_], [] -> []
34              | _::_, _ -> [x ++ " ambiguous"]
35              | [], [] -> [x ++ " unresolvable"]
36              | _, cs -> [x ++ " unresolvable, close to: "
37                          ++ implode(", ", cs)] end;
38   top.ocaml = if !bindNode.inSeqLet
39               then "(Lazy.force " ++ x ++ ")" else x;
40   top.type = bindNode.type; }
41
42 production exprLet top::Expr ::= bs::Binds e::Expr
43 { exists scope s_let;
44   bs.s = top.s; bs.s_last = s_let;
45   e.s = s_let;
46   top.errs = bs.errs ++ e.errs;
47   top.ocaml = bs.ocaml ++ e.ocaml;
48   top.type = e.type; }
49
50 production seqBindsCons top::Binds ::= b::Bind bs::Binds
51 { exists scope s_dcl;
52   scope s_next;
53   edge s_next -[ lex ]-> top.s;
54   edge s_next -[ var ]-> s_dcl;
55   b.inSeqLet = true;
56   b.s = top.s;  b.s_dcl = s_dcl;
57   bs.s = s_next; bs.s_last = top.s_last;
58   top.errs = b.errs ++ bs.errs;
59   top.ocaml = "let " ++ b.ocaml ++ " in " ++ bs.ocaml; }
60 production seqBindsLast top::Binds ::= b::Bind
61 { exists scope s_dcl;
62   scope top.s_last;
63   edge top.s_last -[ lex ]-> top.s;
64   edge top.s_last -[ var ]-> s_dcl;
65   b.inSeqLet = true;
66   b.s = top.s; b.s_dcl = s_dcl;
67   top.errs = b.errs;
68   top.ocaml = "let " ++ b.ocaml ++ " in "; }
69
70 production bind top::Bind ::= x::String  e::Expr
71 { scope top.s_dcl -> datumVar(x, top);
72   e.s = top.s;
73   top.type = e.type;
74   top.errs = assert(anno == e.type, "bad type of " ++ x)
75              ++ e.errs;
76   top.ocaml = x ++ " = " ++
77     if top.inSeqLet
78     then e.ocaml else "lazy (" ++ e.ocaml ++ ")"; }
```

Figure 4. AGTix productions and equations for LM

arguments for scopes. In Figure 3, lines 13–15 define three such scope attributes. Uses in Figure 4 include line 2 defining `s` (the lexically enclosing scope) for the two sub-trees of `exprAdd`, and line 45 passing the last scope created by sequential let binders (`s_let`) to expression `e`.

We build scope graph nodes in AGTix using a `scope` equation corresponding to `new` in Statix, *e.g.* for `s_next` (the next enclosed scope defined by the `let` binding tree) on line 52 of Figure 4. As in Statix, an optional datum can be associated with a scope graph node as on line 71 of Figure 4, corresponding to line 32 in Figure 2. Statix has a *permission to extend* property, defined by Rouvoet et al. [2020], which says that an edge `s -[` *l* `]->` *tgt* can only be asserted if `s` is built using a `new` constraint within the body of the existential that introduces name `s`, with respect to predicate application. In Statix this means that the `new` for a scope may appear anywhere under the construct with the existential introducing its name. A restriction on Statix required by Bessant and Van Wyk [2025] to facilitate translation to RAGs is that the `new` constraint for a scope must be in the same predicate as the existential introducing its name. AGTix lifts this restriction by introducing an `exists scope` construct which acts as a Statix existential for scopes. Line 43 of Figure 4 asserts the existence of scope `s_let`, corresponding to line 10 of Figure 2. This scope is built on line 62 of Figure 4, as on line 22 of Figure 2. AGTix syntax for building scope graph edges, prefaced with `edge`, mirrors the corresponding Statix constraints. Edges can be asserted on scopes that are received as a scope attribute or defined locally with `scope`. On line 53 we define a `lex` edge on the locally defined `s_next` scope, corresponding to line 27 of Figure 2. Lines 63 and 64 construct new edges on scopes passed down the tree as scope attributes declared in the `exists scope` construct above. This corresponds to line 23 in Figure 2.

AGTix includes `query` syntax similar to Statix, *e.g.* on lines 20–22, with syntax for defining path regular expressions and label ordering. The local reference attribute `bindNode` on line 28 in Figure 4 holds a `Decorated` reference, if it exists, to its variable declaration node in the AST. This is extracted from the results of this query by checking that there is exactly one match in the list of results and then getting the binding node (`getBind`) out of it. This is the node `top` given as the second argument to `datumVar` in the variable declaration on line 71. The type of the declaration (`bindNode.type`) determines the type of this use (line 40). If the declaration is in a recursive let-binding (line 38) then lazy evaluation is required in the OCaml translation, but is not if it is a sequential let binding. AGTix queries including a path regular expression, a label ordering, a data well-formedness predicate, and a start scope correspond to uses of Statix `query`, `filter` and `min` on lines 13–16 of Figure 2.

Of interest is the use of the results of these queries in the definition of `errs` on lines 32–37, generating error messages such as ones which identify potential close matches if the

variable itself is not resolved on lines 36–37. We introduce syntax for defining optional partial orders over labels as shown on line 21 and lines 25–26 of Figure 4 to specify shadowing of paths.

4.3 Implementation of AGTix

AGTix is implemented in Silver and thus benefits from certain features for specifying languages in a modular and extensible way. In particular, the implementation of AGTix uses *forwarding* [Van Wyk et al. 2002] to translate AGTix constructs to core RAG attribute definitions and equations of Silver [Bessant and Van Wyk 2025]. As a result AGTix is composable with other extensions to Silver.

Each label introduced by `scope labels` declarations becomes an inherited attribute on an AGTix built-in `ScopeNT` nonterminal to hold references to target scopes along edges with that label. These inherited attributes are provided to a particular scope graph node at either the point at which the node is introduced if its name is local *e.g.* `scope s -> d` or where the existence of the scope graph node is asserted using `exists scope` and the scope to be built is a scope attribute *e.g.* `scope top.s -> d`. The `scope` and `exists scope` constructs translate into production calls to build `ScopeNT` nodes and decorate them with their edge label inherited attributes.

A `scope attribute` declaration generates an inherited attribute for the scope itself, and synthesized attributes which lift the targets of edges to be sourced at that scope up the tree to where the scope is decorated with the inherited attributes for its labels. As identified by the permission-to-extend property of Statix, the `new` constraint building a scope may appear anywhere under the existential that introduces its name, with respect to predicate application. Consequently scopes need only be passed down an AST as inherited attributes since they are constructed at the location of the exists constraint for a scope name in Statix.

AGTix assertions of edges on scopes that are passed down as scope attributes populate the definitions synthesized attributes for labels that the `scope attribute` declaration generates. On the other hand assertions of edges on locally built scopes are directly included in the definition of the corresponding label attribute on that scope. AGTix queries become applications of a built-in query function. The regular expression syntax compiles to a tree representation, and the partial label ordering syntax becomes a comparison function over labels used by the query function to decide which paths to follow are most preferred.

5 Discussion, Future Work, and Conclusion

We previously formalized [Bessant and Van Wyk 2025], by showing how a complete Statix specification can be turned into a RAG specification, "what has been the folklore in the [AG] community for some time, that scope graphs are naturally specified in [RAGs]." This work follows a similar approach, but instead of a monolithic translation of a complete specification it individually translates Statix-like constructs embedded in a RAG specification down to RAG declarations of attributes and equations. Implementing scope graphs in RAGs without Statix-like constructs tends to result in more verbose specifications; *e.g.*, the constraints on possible name resolutions specified by the regular expressions and label orderings written in a `query` construct must instead be encoded in explicit checks against edge labels to determine validity. In Silver, one would also need to write the synthesized attribute equations to collect the edge targets for scopes created higher up in the AST. It is worth noting that this would not be required in JastAdd since its list *collection attributes* [Magnusson et al. 2007] can have new elements added to the list from a remote reference to the node on which such an attribute occurs. Thus a scope node can be passed down the AST and a collection attribute holding edge targets for that scope can be extended through that reference. This is a feature that Silver does not support.

AGTix creates new nodes for scopes and declarations which often correspond to nodes in the AST. The datum on a declaration, as seen in the `bind` production, contains a reference to that node named `top`. In an ad-hoc encoding of scope graphs in RAGs one can instead draw scope graph edges between nodes *in* the AST instead of newly-created scope nodes. Then the data on declarations is just the attributes already declared the AST nodes. As future work, we are investigating doing this in AGTix. For example, in the `seqBindCons` production in Figure 4, both the `top` and `b` nodes could be designated as scopes and a `var` edge drawn directly between them. This requires an ability to indicate that different AST nonterminals may appear in the results of queries. Silver supports type-classes and this may provide a solution to what would otherwise be a static typing problem, but this needs to be studied further.

The AGTix approach combines the generality of attribute grammars for computing a variety of tasks over trees with the specificity of Statix constraints for concisely specifying the scoping and name binding structure of programs or models and running name-resolution queries over that structure. It is implemented as an extension to Silver, an attribute grammar system designed to support modular, independently-developed extensions that can add new syntax and semantic analyses. Silver is freely available under an open-source license at https://melt.cs.umn.edu/silver/. The AGTix artifact that accompanies this paper can be found at https://doi.org/10.5281/zenodo.20276179.

Acknowledgments

This work was supported in part by the National Science Foundation, award #2123987.

References

F. Baader and T. Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press, Cambridge, U.K. doi:10.1017/CBO9781139172752

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

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:10.1145/3732771.3742711

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

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

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA) (OOPSLA)*. Association of Computing Machinery. doi:10.1145/1869459.1869497

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.

Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. 2007. Extending Attribute Grammars with Collection Attributes–Evaluation and Applications. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 69–80. doi:10.1109/SCAM.2007.13

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

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

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

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

Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the Conference on Compiler Construction (CC) (Lecture Notes in Computer Science, Vol. 2304)*. Springer-Verlag, 128–142. doi:10.1007/3-540-45937-5_11

Eelco Visser. 2001. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01) (Lecture Notes in Computer Science, Vol. 2051)*, A. Middeldorp (Ed.). Springer-Verlag, 357–361. doi:10.1007/3-540-45127-7_27

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

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