Specializing Scope Graph Resolution Queries

Zwaan, A.S.

DOI
10.1145/3567512.3567523

Publication date
2022

Document Version
Final published version

Published in
SLE 2022

Specializing Scope Graph Resolution Queries

Aron Zwaan
Software Technology
Delft University of Technology
Delft, Netherlands
a.s.zwaan@tudelft.nl

Abstract

To warrant programmer productivity, type checker results should be correct and available quickly. Correctness can be provided when a type checker implementation corresponds to a declarative type system specification. Statix is a type system specification language which achieves this by automatically deriving type checker implementations from declarative typing rules. A key feature of Statix is that it uses scope graphs for declarative specification of name resolution. However, compared to hand-written type checkers, type checkers derived from Statix specifications have suboptimal run time performance.

In this paper, we identify and resolve a performance bottleneck in the Statix solver, namely part of the name resolution algorithm, using partial evaluation. To this end, we introduce a tailored procedural intermediate query resolution language, and provide a specializer that translates declarative queries to this language.

Evaluating this specializer by comparing type checking run time performance on three benchmarks (Apache Commons CSV, IO, and Lang3), shows that our specializer improves query resolution time up to 7.7x, which reduces the total type checking run time by 38 – 48%.

CCS Concepts: • **Theory of computation** → **Program semantics**; **Graph algorithms analysis**; Regular languages; • **Software and its engineering** → *Domain specific languages*; Semantics.

Keywords: scope graphs, graph query resolution, specialization, partial evaluation, declarative languages

ACM Reference Format:
Aron Zwaan. 2022. Specializing Scope Graph Resolution Queries. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3567512.3567523

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12.
https://doi.org/10.1145/3567512.3567523

1 Introduction

Developers, whether they use a general-purpose or a domain-specific language (DSL), use static name and type analysis to understand and evolve their code. However, implementing a type checker takes significant time and effort. In particular, implementing name binding correctly is challenging, as it requires careful staging of program traversals [17]. Therefore, type checker frameworks that abstract over name resolution scheduling, such as Pacak et al. [14] (based on Datalog), Van Wyk et al. [29], Hedin and Magnusson [5] (using attribute grammars), and Van Antwerpen et al. [25] (using constraint programming and scope graphs) have been developed. These frameworks ensure executable type checkers can be developed with significantly reduced effort.

Interpreting such declarative specifications often requires intricate logic. Generally, the more a language abstracts from implementation details, the more complicated an interpreter or compiler will be. However, this comes with the risk of introducing significant run time overhead, resulting in suboptimal performance compared to low-level approaches.

In this paper, we improve the performance of type checkers based on *scope graphs* [11, 25]. Scope graphs are an established approach to modeling name-binding structure. In this model, the scoping structure and declarations of a program are represented in a graph. References can be resolved using a versatile graph query mechanism. Scope graphs are embedded in the Statix DSL for type system specification [17, 25]. This DSL allows high-level specification of type systems using declarative inference rules. It has a well-defined declarative semantics, which allows reasoning over type-systems while abstracting over operational details. The Statix solver interprets specifications as constraint programs, which yields executable type checkers that are sound with respect to the declarative semantics. Case studies using Statix have shown that scope graphs are expressive enough to support type systems with non-lexical bindings (e.g., imports and inheritance), structural types, and parametric polymorphism [11, 25]. In addition, they allow language-parametric definition of editor services, such as semantic code completion [15], renaming [9, 10] and inlining [28]. The expressiveness, declarativity, and additional services makes it especially suitable for DSLs and rapid language prototyping.

However, type checkers derived from Statix specifications are rather slow. For example, type checking the Apache Commons IO library takes 14.7 seconds with the concurrent solver

using 8 cores and even 73.4 seconds on a single core [26]. On the same machine, a full compilation using `javac` takes roughly 3 seconds on 8 cores, and 5 seconds on a single core.

In this paper, we resolve a newly identified performance bottleneck in Statix' scope graph query resolution algorithm using partial evaluation. Our evaluation shows that the approach ensures query resolution is up to 7.7x faster than traditional query resolution on average. This improves the performance of Statix-based type checkers by 38 – 48% on Java projects, which is a significant step forward to applying generated type checkers on larger codebases.

In summary, our contributions are as follows:

- We explain the scope graph query resolution algorithm, and identify one of its major performance bottlenecks (section 3).
- We introduce an intermediate language that makes scope graph traversal order and partial query result combination explicit (section 4).
- We present a specializer from traditional scope graph queries to our new intermediate language (section 5).
- We evaluate the correctness and performance of our approach (section 6). We show that specializing queries makes scope graph query resolution up to 7.7x faster.

2 Partial Evaluation for DSL Interpreters

In this section, we provide a brief introduction to partial evaluation (popularized by Futamura [4]), and explain why we think it is especially beneficial for declarative languages such as Statix. From the perspective of partial evaluation, a program can be seen as a function from inputs to an output O. Some of these inputs may be known statically (S), while some of them may vary per invocation (D). Then, the signature of a program looks as follows:

$$\text{prog} : S \times D \to O$$

A *specializer* takes such a program and its static input, and returns a *residual program* $\text{prog}_S : D \to O$. When generating prog_S, it performs the part of the computation that does not depend on D, making prog_S generally faster than prog.

2.1 Partial Evaluation for Interpreters

This pattern can easily be applied to programming languages. In that case, prog is an interpreter that evaluates the semantics of a program (its static input S) with respect to some arguments D. The residual program is essentially a compiled version of S. This is called the *first Futamura projection*.

Specialization is generally only beneficial when a program is executed multiple times. However, Futamura argues that specializing an interpreter to a program may already be beneficial when executing the program once, as programs may repeatedly evaluate a particular piece of code. Specializing repeatedly executed program fragments removes the interpretation overhead, which might outweigh the run time costs of compilation. This effect becomes stronger when the computational complexity of interpreting particular language constructs is high. That is, the more overhead an interpreter introduces, the more beneficial specialization will be.

Declarative languages are languages in which a user specifies intent rather than procedure. The logic to compute a result that corresponds to the intent is then implemented in the interpreter (or compiler) of the language. Thus, a declarative language moves part of the complexity of a problem from the program or programmer to its interpreter.

Having an interpreter with intricate logic means that declarative languages are susceptible to introducing relatively more run time overhead compared to non-declarative languages. Interpreters of declarative languages might have to execute non-trivial algorithms in order to evaluate a program. For that reason, partial evaluation might be particularly beneficial for declarative languages.

2.2 Application to Statix

Applying partial evaluation to Statix introduces a few problems. In fact, it is as complex as finding a compiler from a constraint language with an embedded scope graph logic to an imperative language, such as Java. Such a compiler should ensure that all internal scheduling of rule selection and query resolution is handled correctly. We regard this as an open research challenge that is too complicated to solve in one step. Instead, in this paper, we specialize only a computationally complex part of the interpreter, namely the query resolution algorithm, to a specification. This yields a specification in which the query resolution constraints are partially evaluated, but other constraints are not. This specification can then be interpreted by a constraint solver without using the query resolution algorithm, ensuring faster execution.

To characterize this approach, regard a Statix specification $C_Q \uplus C_O$ as a collection of query constraints C_Q and other constraints C_O. The $\uplus$ symbol indicates that these groups are mutually embedded in actual specifications. Our approach is then summarized in the following functions:

$$\text{specialize} : C_Q \uplus C_O \to C_Q^* \uplus C_O$$
$$\text{solve} : C_Q^* \uplus C_O \times P \to T$$

Here specialize specializes the query resolution algorithm with respect to particular query constraints, yielding specialized queries (C_Q^*). These constraints are embedded in the original abstract syntax tree (AST), yielding a partially specialized specification $C_Q^* \uplus C_O$. When type-checking a concrete object program P, this specialized specification is interpreted by an adapted solver solve, returning a typing T.

Specialized queries C_Q^* cannot be represented in Statix, and only as a verbose shallow embedding in Java (in which the solver is written). Instead, we express those in an intermediate language. For this language, we provide an interpreter, which Statix uses instead of the name resolution algorithm.

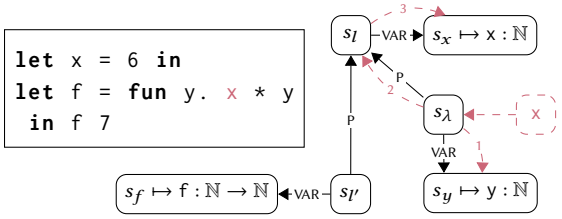


Figure 1. PCF program with scope graph and query

3 Resolving Queries in Scope Graphs

In this section, we introduce scope graphs and query resolution. Section 3.1 introduces scope graphs and three parameters of scope graph queries using two examples. After that, section 3.2 explains how Statix interprets these query parameters. Then, we explain that repeated querying of some of these parameters makes query resolution slow, which motivates the effort to optimize it (section 3.3). Finally, section 3.4 provides the full resolution algorithm, which is required to understand the remainder of the paper.

3.1 Query Resolution by Example

We consider two examples of queries in scope graphs. These examples motivate the declarative query language of Statix, as well as explaining the resolution algorithm that interprets such queries. Each example discusses the scope graph of a program, and the resolution of a particular query in that scope graph. The examples are included in the artifact [31], and the extended edition of this paper contains their full derivations [32].

Example 1. In fig. 1, a small program written in PCF (Programming Computable Functions) is shown. The program contains two let-bindings, a function definition and its application. Next to the program, the scope graph is depicted. In a scope graph, each 'scope' is modeled by a node. In this case, s_I and $s_{I'}$ represent the bodies of the let expressions, while s_λ models the body of the function. The P-labeled edges, such as the edge from $s_{I'}$ to s_I, ensure declarations of outer scopes will be visible in inner scopes. Nodes s_x, s_y, and s_f model the declarations of the x, y, and f variables, respectively. Therefore, these scopes map to a datum (e.g. x : ℕ for s_x) that indicates the name and type of the declaration.

References are modeled using *queries* in scope graphs. In the code, a reference x is highlighted. This reference corresponds to the dashed box in the scope graph. The box points to s_λ, because x occurs in the body of the function expression. Eventually, the query resolves to s_x via s_I, which is indeed the declaration of x in the outer let.

For this paper, we are particularly interested in *how* this query result was computed. This is indicated by the numbered, dashed edges. When starting the query in s_λ, the algorithm first traverses the VAR edge to s_y. Then it checks whether s_y is a valid declaration for the reference. Since this is not the case, the algorithm continues by traversing the P edge to s_I (step 2). From there, the VAR edge to s_x is traversed. The algorithm finds that s_x is a valid declaration, and returns that as the environment to which the query evaluates.

In Statix, one would not write a resolution procedure as shown above directly, as Statix is meant to be a declarative specification language. Instead, Statix interprets a high-level description of valid query answers using a *generic* algorithm, yielding the behavior as shown above. So how can we describe the query resolution procedure in a high level fashion?

The given example already shows two of the three *query parameters* that determine how a query resolves to an environment. First, the query resolution algorithm decided that s_y should not be in the environment, while s_x should. That is expressed using a *data well-formedness condition* **D**, which is a unary predicate over datums. A possible declaration is only included in the environment when its datum matches **D**. In this case, the predicate only accepts declarations with name x. Second, the algorithm decided to traverse VAR edges before P edges. This corresponds to the intuition that local declaration are prefered over (i.e., shadow) more distant declarations. In Statix, this is modeled using a strict partial order over labels (refered to as *label order*). For this example, the label order VAR < P indicates that VAR edges should be traversed first.

Example 2. Fig. 2 shows an example program in Language with Modules (LM, introduced by Néron et al. [11]). In this scope graph, scope s represent the *global scope*. Scopes s_A, s_B, s_C, s_D and s_E have a double role: they model the declaration of a module as well as its body. Therefore, they have incoming MOD edges and a datum as well as inner declarations. All modules have a P edge back to their enclosing context. Finally, the imports are modeled using I edges.

When resolving x in s_E, the resolution algorithm first traverses the VAR edge to s_y (step 1). Because that declaration does not match, it continues traversing the I edge to s_B (step 2). That scope does not contain valid declarations but it has P and I edges to s_A, and a MOD edge to s_C. However, reference resolution should not traverse those. Clearly, traversing the P edge would be incorrect. After all, module E only imports A.B, which should not bring declarations from A in scope. Whether traversing the I edge is valid depends on the language. Languages with transitive imports would allow traversing multiple subsequent I edges, while language with non-transitive imports would not. As LM has non-transitive imports, we do not traverse the I edge to s_A. Similarly, the MOD edge to s_C should not be traversed, as modules should not be imported implicitly. Instead, the query resolves to declaration s_2 in s_C via the import edge from s_E (step 3 and 4). Now, we still need to consider whether it is necessary to traverse the P edge to s_D. When imports shadow the surrounding scope, that is not required, as s_2 would shadow any result from s_D. However, for the sake of the example,

```
module A {
  def x = 1
  module B {
    import A
    module C {
      def x = 2
    }
  }
}
```

```
module D {
  def x = 3
  module E {
    import A.B
    import A.B.C
    def y = x
  }
}
```

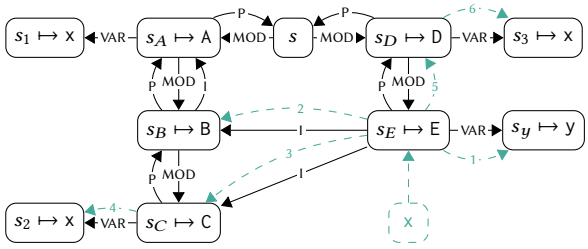


Figure 2. LM example. Types are omitted for brevity.

we assume that imports and the enclosing scope have equal priority. Thus, s_3 is resolved as well (step 5 and 6). Hence, there are multiple declarations the query resolved to, which means that reference x is ambiguous.

This example shows that there can be additional constraints on paths. In Statix, these *path-wellformedness conditions* are expressed as a *regular expression* (RE) on path labels. The query resolution algorithm only traverses paths that are valid with respect to the given RE. In particular, the regular expression that describes this query is P*I?VAR. This regular expression allows looking in lexically enclosing scopes (P*), possibly traversing a single import edge (I?), while ensuring that only variables are resolved (VAR).

Second, we argued that the reference is ambiguous when imports and lexical enclosing scopes have equal priority. This can be modeled by having $\{ \mathsf{VAR} < \mathsf{P}, \mathsf{VAR} < \mathsf{I} \}$ as label order, which has neither $\mathsf{I} < \mathsf{P}$ nor $\mathsf{P} < \mathsf{I}$. Then, the label order does not indicate priority for a particular environment, and the resolution algorithm will return them both.

Summary. These examples show how scope graphs can be used to model the name binding structure of PCF and LM programs. Resolution of references is done using declarative scope graph queries. Valid results of a query are described by three parameters. First, the path well-formedness condition, expressed as a regular expression over labels, describes valid paths. Second, the data well-formedness condition ensures only valid declarations are returned. Finally, the label order condition describes which declarations shadow each other. A query resolution algorithm, integrated in the Statix solver, interprets such parameters to compute the result of a query.

3.2 Query Resolution: Algorithm Outline

When executing a Statix specification, a resolution algorithm resolves queries. This algorithm performs an advanced depth-first search, using the aforementioned query parameters to find correct results. In this section, we explain how each of the query parameters is used by the algorithm.

Path well-formedness. First, consider the use of the path well-formedness RE R. The fact that valid paths need to adhere to R implies that query resolution does not have to traverse the whole graph, but only edges with labels that do not violate R. The labels l that require traversal are precisely those for which the Brzozowski derivative $\partial_l R$ [2] is not equal to the empty language $\emptyset$. We call this set the *head set* of a regular expression (written as $\mathscr{H}(R)$), defined as:

$$\mathscr{H}(R) \triangleq \{ l \in \mathcal{L} \mid \partial_l R \neq \emptyset \}$$

Moreover, after traversing an edge with label l, the resolution algorithm uses $\partial_l R$ for further exploration from the target node. This retains the invariant that full resolution paths will adhere to the initial regular expression.

To illustrate this, recall the example in fig. 2. The query for x has $R_2 = \mathsf{P}^*\mathsf{I}?\mathsf{VAR}$ as initial RE. Because this regular expression has derivatives with respect to VAR, I and P, all edges were traversed (step 1, 2, 3 and 5). However, from s_B, the I and P edges were not traversed because $\partial_\mathsf{I} R_2 = \mathsf{VAR}$ was used as RE after traversing the I edge (step 2).

Label Order. The label order $<_l$ is used to model shadowing. This is implemented in the algorithm as follows. The algorithm traverses edges in topological ascending order of their labels. Results obtained from a particular edge are then only included when they are not shadowed by results obtained from an edge with a smaller label.

As an example, consider the label order used in the LM example: $\{ \mathsf{VAR} < \mathsf{P}, \mathsf{VAR} < \mathsf{I} \}$. This order indicates that results over VAR labels shadow results from P and I, but the latter two do not shadow each other. Thus, the algorithm first traverses VAR edges. The result of that traversal is then used to shadow results obtained from P and I edges. That is, when the VAR traversal returned a non-empty environment, declarations reached via P and I edges are ignored. However, as the label order is not necessarily total, some variability in the order of traversing mutually incomparable labels is possible. In the above example, it does not matter whether P edges are traversed before I edges, or the other way around. However, it is important that neither shadows the other.

Internally, the algorithm determines the label traversal order using two helper functions:

$$\mathsf{max}_{<_l}(\hat{L}) \triangleq \{ \hat{l} \in \hat{L} \mid \nexists \hat{l}' \in \hat{L}.\ \hat{l} <_l \hat{l}' \}$$
$$\mathsf{smaller}_{<_l}(\hat{L}, \hat{l}) \triangleq \{ \hat{l}' \in \hat{L} \mid \hat{l}' <_l \hat{l} \}$$

Here, max computes the labels that are highest in the label order, which are hence traversed *last*. For each label $\hat{l}$ in the

```
1 fun TraverseOrdered(L̂)
2     if L̂ = ∅ then return ∅
3     A := ∅
4     foreach l̂ ∈ max_<l̂(L̂) do
5         A_L := TraverseOrdered(smaller_<l̂(L̂, l̂))
6         A_l := Resolve-Î(l̂)
7         A += Shadow(A_L, A_l)
8     return A
```

Figure 3. Edge traversal order

```
1  TraverseOrdered({ VAR, I, P })
2      A := ∅
3      max_<l̂({ VAR, I, P }) = { I, P }
4          smaller_<l̂({ VAR, I, P }, I) = { VAR }
5              TraverseOrdered({ VAR }) = ∅
6              Resolve-Î(I) = { s2 }
7              Shadow(∅, { s2 }) = { s2 }
8              A := ∅ ∪ { s2 }
9          smaller_<l̂({ VAR, I, P }, P) = { VAR }
10             TraverseOrdered({ VAR }) = ∅
11             Resolve-Î(P) = { s3 }
12             Shadow(∅, { s3 }) = { s3 }
13             A := { s2 } ∪ { s3 } = { s2, s3 }
14     return { s2, s3 }
```

Figure 4. Execution trace of the algorithm in fig. 3 applied to s_E in fig. 2. It shows how results obtained over I edges (s_2) and P edges (s_3) are not mutually shadowed.

max set, a smaller set is computed. This set contains precisely those labels that shadow the $\hat{l}$ label. Using these functions, the actual traversal order and shadowing of a set of labels $\hat{L}$ is determined as shown in fig. 3. The `TraverseOrdered` function receives a set of labels it must traverse in the correct order. If the set is empty, an empty environment is returned (line 2), as no edges need to be traversed. Otherwise, the algorithm initializes an empty environment A, which eventually will contain all declarations reachable over labels in $\hat{L}$. Then, it iterates over all labels $\hat{l}$ in the max set of $\hat{L}$. For each max label $\hat{l}$, all declarations that may possibly shadow it are computed by recursively applying `TraverseOrdered` on its smaller set. This ensures labels in the smaller set are traversed before $\hat{l}$. After that, all $\hat{l}$-labeled edges are traversed. The result of the latter operation is shadowed against the result of the former, and the result is added to A.

An astute reader may ask why we do not simply return A_L, and only compute A_l when the former is empty. This is caused by an additional query parameter we do not discuss here. We return to this in sections 3.4 and 5.4.

To illustrate this, consider the execution trace for the example in fig. 2, assuming the traversal is in s_E, which is shown in fig. 4. Here, the initial max set is { I, P } (line 3). The algorithm iterates over these labels in lines 4–8 (where $\hat{l}$ = I) and 9–12 (where $\hat{l}$ = P). Each of these iterations computes { VAR } as its smaller set, and by recursively applying `TraverseOrdered` finds that its corresponding environment is empty. Then the $\hat{l}$ environment is computed, shadowed relative to the VAR-environment (which leaves it unchanged), and added to A. The union of the results of both iterations is then returned (line 13 and 14). This shows how both I and P are shadowed by VAR, but not by each other.

Data well-formedness. Finally, we select valid declarations using the data well-formedness condition **D**. This is simply done by evaluating **D** on the datum of the current scope. However, there is a small subtlety that must be accounted for. The resolution algorithm may visit scopes via a path that does not match the original R, but is only a *prefix* of a sequence of labels matched by R. We should traverse these scopes, but not return them as declaration. Full paths are precisely those on which the language of the derivative at the targets scope R' (written as $\mathscr{L}(R')$) includes the empty word ε. Only full paths are considered by **D** to be included in the query answer.

Returning to the example in fig. 2 again, this means that **D** is not applied to the datum of s_C (among other scopes), as the language of the current RE ({ VAR }) does not include ε. However, after traversing the edge to s_2, the RE becomes $\partial_{\mathsf{VAR}}\mathsf{VAR} = \varepsilon$, which obviously matches ε. Therefore, **D** is applied to the datum of s_2, selecting it as a valid declaration.

3.3 Performance of the Resolution Algorithm

Recall from the introduction that query resolution is slow. Analyzing the algorithm outline can give an intuition why that is the case. First, observe that computing the max and smaller sets, which is executed per scope the resolution algorithm traverses, is quadratic in the number of labels. Similarly, the number of derivatives computed per scope traversal is linear in the number of labels. Therefore, for large scope graphs and label sets, the overhead of these computations can be significant. Profiling the Java Commons IO project shows that 12% of the total query resolution time is spent on computing derivatives and 35% on checking label orders [31]. Therefore, we lift these computations to specification compile-time, ensuring faster execution.

3.4 The Resolution Algorithm

In order to understand the specialization, we first need to understand the generic query resolution algorithm in full detail. Therefore, we present this algorithm now.

For the algorithm, we use the following notation. Scope graphs $\mathcal{G}$ are defined as a three-tuple $\langle S \subset \mathcal{S}, E \subset \mathcal{E}, \rho \rangle$ of scopes s, edges e and a mapping from scopes to data $d \in \mathcal{D}$.

1 **fun** Resolve($\mathcal{G}, s, R, \mathbf{D}, <_{\hat{l}}, \approx_d$) : $\mathcal{A}$
2 **fun** Resolve-\$(p) $\triangleq$ **if** $\mathbf{D}(\rho_{\mathcal{G}}(\text{tgt}(p)))$ **then** $\{ p \}$ **else** $\emptyset$
3 **fun** Resolve-I(p, l, R') $\triangleq \bigcup \{$ Resolve-All($p \cdot l \cdot s', \partial_l R'$) $\mid \text{tgt}(p) \cdot l \cdot s' \in E_{\mathcal{G}}, s' \notin p \}$
4 **fun** Resolve-Î($p, \hat{l}, R'$) $\triangleq$ **if** $\hat{l} = \$$ **then** Resolve-\$(p) **else** Resolve-I($p, \hat{l}, R'$)
5 **fun** Shadow(A_L, A_l) $\triangleq A_L \cup \{ p \in A_l \mid \nexists p' \in A_L.\ \rho_{\mathcal{G}}(\text{tgt}(p')) \approx_d \rho_{\mathcal{G}}(\text{tgt}(p)) \}$
6 **fun** Resolve-ÎL̂($p, \hat{L}, \hat{l}, R'$) $\triangleq$ Shadow(Resolve-L̂($p, \hat{L}, R'$), Resolve-Î($p, \hat{l}, R'$))
7 **fun** Resolve-L̂($p, \hat{L}, R'$) $\triangleq \bigcup \{$ Resolve-ÎL̂($p, \hat{L}', \hat{l}, R'$) $\mid \hat{l} \in \max_{<_{\hat{l}}}(\hat{L}), \hat{L}' = \text{smaller}_{<_{\hat{l}}}(\hat{L}, \hat{l}) \}$
8 **fun** Resolve-All(p, R') $\triangleq$ Resolve-L̂($p, \mathcal{H}(R') \cup \{ \$ \mid \varepsilon \in \mathcal{L}(R') \}, R'$)
9 **return** Resolve-All(s, R)

Figure 5. Query resolution algorithm (adapted from Van Antwerpen and Visser [27, alg. 5])

Its components can be projected by using $\mathcal{G}$ as subscript (e.g., $S_{\mathcal{G}}$ refers to the scopes of $\mathcal{G}$). An edge $s \cdot l \cdot s'$ consists of a source s, a target s' and an edge label $l \in \mathcal{L}$. We define path labels $\hat{l}$ to be either an edge label or the end-of-path label \$. L denotes a set of edge labels not containing \$, and $\hat{L}$ is a set of labels that might contain \$. A path $p \in \mathcal{P}$ can be a single scope s or a path step $p \cdot l \cdot s$. The source and target of a path can be projected using src$(\cdot)$ and tgt$(\cdot)$. An environment $A \in \mathcal{A}$ is a set of resolution paths.

Algorithm. The full algorithm is shown in fig. 5. It has six inputs: the scope graph, the scope in which to start the query, and the three query parameters. The last argument, called the *data equivalence condition* ($\approx_d$) is a binary predicate over datums. This argument is used for advanced shadowing, such as shadowing based on overload resolution. In short, a declaration d is shadowed by another declaration d' when the path leading to d' has priority, but also when $d \approx_d d'$. For a full treatment, we refer to Van Antwerpen et al. [25] and Rouvoet et al. [17].

The algorithm definition uses a number of inner functions that each have their own arguments. In particular, each function receives a path p, which is the path that the query resolution algorithm traversed so far. Moreover, most functions have an R' argument, which is the derivative of R with respect to the labels in p. The algorithm starts by computing the full environment of the initial scope and R using Resolve-All (line 9). This function computes the labels the regular expression can follow ($\mathcal{H}(R')$), and includes \$ if the empty word ε is in the language described by the regular expression ($\mathcal{L}(R')$). As we will see, including \$ ensures the local scope is checked for being a valid declaration.

The actual, correctly shadowed environment for these labels is computed using Resolve-L̂ and Resolve-ÎL̂. In fact, Resolve-L̂ resembles TraverseOrdered from section 3.2, where Resolve-ÎL̂ is the body of its **for**-loop. Resolve-L̂ computes the max and smaller sets of its $\hat{L}$ argument. The environment of the max-label (computed using Resolve-Î) is shadowed with respect to the environment of its smaller labels (recursively computed using Resolve-L̂). Shadowing is implemented by the Shadow function, which returns the union of A_L and the elements of A_l that are not shadowed by an element in A_L. That is, paths in A_l that are shadowed by a path from A_L are removed from the final environment.

Computing an environment for a single label is done using the Resolve-Î function. When the $\hat{l}$ is the end-of-path label \$, the current path is included in the result when its datum matches $\mathbf{D}$ (Resolve-\$). For an edge label l, Resolve-I retrieves all outgoing edges of the target of the current path. The $s' \notin p$ condition precludes cyclic paths, ensuring termination. For each edge, Resolve-All is invoked. The given arguments are the current path, extended with the traversed edge, and the derivative of the current regular expression R' with respect to the current label. This retains the invariant that the new R' is the derivative of R with respect to the labels in the new path. Therefore, the paths returned by the algorithm are valid according to R.

Finally, we want to highlight a few characteristics of the algorithm. First, note that this algorithm does not enforce local scopes to be considered first. For example, a query that has label order L < \$ will traverse L before checking the local scope. Second, an invocation of Resolve-All after traversing an edge in Resolve-I can very well be seen as an independent query from the target of the p argument. We call these *residual queries*.

4 An Intermediate Resolution Language

Before we define the specializer for queries, we present the intermediate language in which the specialized queries are represented.

4.1 Syntax

The language that allows us to express specialized versions of name resolution queries is shown in fig. 6. First, the syntax of query parameters as discussed in the previous section is formalized. Then, there are names for variables (x) and states (n). Variables can be used in *resolution expressions* (E).

Query Parameters

R	$\in$	$\mathcal{R}$	$\subset$	$\mathcal{L}^*$	Path well-formedness condition
$\mathbf{D}$	$\in$	$\mathcal{D}$	$\subset$	$\mathcal{D}$	Data well-formedness predicate
$<_{\hat{l}}$	$\in$	$\mathcal{O}$	$\subset$	$\hat{\mathcal{L}} \times \hat{\mathcal{L}}$	Strict partial order on labels
$\approx_{\mathrm{d}}$	$\in$	$\mathcal{E}$	$\subset$	$\mathcal{D} \times \mathcal{D}$	Data equivalence condition

Resolution State Machine

x	$\in$	$\mathcal{X}$			Variables
n	$\in$	$\mathcal{N}$			State Names
E	$\in$	$\mathcal{E}$	::=	**resolve** | **subenv** l n	
			|	**merge** $\overline{x}$ | **shadow** x x	Expressions
Y	$\in$	$\mathcal{Y}$	::=	**state** $\overline{x := E}$	States
M	$\in$	$\mathcal{M}$	::=	**state machine** $\overline{n\text{: } Y}$	State Machine

Statix

C	::=	... | **query** $R, \mathbf{D}, <_{\hat{l}}, \approx_{\mathrm{d}}$ **in** $s \mapsto x$		Generic Statix
C^*	::=	... | **query** $\mathcal{M}, \mathbf{D}, \approx_{\mathrm{d}}$ **in** $s \mapsto x$		Compiled Statix

Figure 6. Syntax of the new query resolution language

There are four possible expressions: **resolve** resolves the current path; **subenv** traverses all edges with label l, executing residual queries in state n; **merge** merges a collection of environments; and **shadow** computes the union of two environments, filtering shadowed paths from the second environment. A sequence of assignments of expressions to variables constitutes a state (Y). A state machine (M) consists of a sequence of states, which each are identified by a name. Implicitly, its first state is designated as initial state. Finally, the figure shows how this language is embedded in Statix. There is the traditional variant of Statix (C) [17, 25], that has a generic query constraint, which can be resolved using the resolution algorithm introduced in the previous section. In this paper, we define C^*, which is a variation of C with the generic query removed and a compiled query added. This compiled query does not have a path well-formedness condition nor a label order as arguments, but a state machine instead. After discussing the semantics of query resolution using state machines, we present a compilation scheme from C to C^* in section 5.

4.2 Semantics

In this section, we explain how queries in our intermediate language are interpreted (see in fig. 7). Where appropriate, we explain how it relates to the algorithm in fig. 5. We use the following notation. First, $M(\cdot) : \mathcal{N} \rightharpoonup \mathcal{Y}$ retrieves the state with a particular name from M. Second, we use $\mathsf{init}(\cdot) : \mathcal{M} \to \mathcal{Y}$ to retrieve the initial state of a state machine. Finally, Σ represents a *store* that maps variables (x) to environments (A). ϵ represents the empty store, Σ; (x, A) denotes adding a mapping $x \mapsto A$ to Σ, and $\Sigma(\cdot) : \mathcal{X} \rightharpoonup \mathcal{A}$ retrieves the value (an environment) of a variable.

Following Rouvoet et al. [17], we define the semantics of evaluating a compiled query in a small-step style: a solver state that consists of a scope graph and a set of unsolved constraints ($\overline{C^*}$) steps to a new state. This is shown in the Op-Query-SM rule, which defines the answer set of the query to be the result of evaluating the initial state with the initial scope as path (similar to line 9 of fig. 5). The remaining constraints $\overline{C^*}$ in the output state have the environment substituted in free positions of the result variable x. A state is evaluated by sequentially executing all its steps (Eval-State). Initially, it starts with an empty store, and each step extends the store with a new mapping. The rule returns the value of the last expression. Exp-Resolve defines that a **resolve** expression returns the current path if its target datum is well-formed according to $\mathbf{D}$. This is similar to the body of `Resolve-$`. Next, **subenv** expressions are evaluated using the Exp-Subenv rule. This rule defines P to be all valid extensions of path p with label l in $\mathcal{G}$. For each path in P, we evaluate the new state Y. The result is the union of the resolved residual queries. This is similar to `Resolve-l`. However, instead of computing a derivative of a regular expression, we pass a new state. As our compilation scheme is designed to ensure that a state is equivalent to `Resolve-L̂` specialized to its R' argument, the behavior of this rule matches the `Resolve-l` function. The **merge** primitive simply does a lookup of all variables in the current store, and returns the union of them. This operation will be a part of the compilation scheme for specializing `Resolve-L̂`. Finally, the **shadow** construct resolves its argument environments, and returns the first environment, combined with the second environment with all shadowed paths removed. This is similar to the `Shadow` function.

5 Specializing Declarative Queries

Now that we have defined the language in which we can represent our partially specialized queries, we can define the specializer. We first discuss a few examples, and then present its full definition.

5.1 Examples

In fig. 9, a few examples of specialized queries are shown. The first example shows a state machine that represents a query that traverses a single label L without shadowing. In this machine, state n_0 has a single **subenv** expression that traverses L labels while transitioning to state n_1. In state n_1, the current path is resolved. So how was this translation performed? First, each state in the state machine corresponds to a possible derivative of R. State n_0 is derived from the original regex L, while state n_1 is derived from $\partial_{\mathsf{L}}\mathsf{L} = \varepsilon$. Each derivative has a singleton head set ({ L } and { $ }, respectively). Therefore, each state is implemented by resolving that label. The **subenv** expression in n_0 transitions to n_1 because state n_1 corresponds to the derivative with respect to L of the regex of state n_0. In the state machine of the second example, there is only one state, as $\partial_{\mathsf{L}}\mathsf{L}^* = \mathsf{L}^*$. However, the

Expression Evaluation Semantics $\boxed{\Sigma, \mathcal{M}, \mathcal{G}, \mathcal{D}, \mathcal{E} \vdash \mathcal{P}, \mathcal{E} \Rightarrow \mathcal{A}}$

$$\textsc{Exp-Resolve}\ \frac{A = \{\, p \mid \mathbf{D}(\rho_{\mathcal{G}}(\mathrm{tgt}(p))) \,\}}{\Sigma, M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p, \mathbf{resolve} \Rightarrow A} \qquad \textsc{Exp-Subenv}\ \frac{\begin{array}{c} Y = M(n) \qquad P = \{\, p \cdot l \cdot s' \mid \mathrm{tgt}(p) \cdot l \cdot s' \in E_{\mathcal{G}} \,\} \\ A = \bigcup \{\, A' \mid p' \in P,\ M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p', Y \Rightarrow A' \,\} \end{array}}{\Sigma, M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p, \mathbf{subenv}\ l\ n \Rightarrow A}$$

$$\textsc{Exp-Merge}\ \frac{A = \bigcup \{\, \Sigma(x) \mid x \in \overline{x} \,\}}{\Sigma, M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p, \mathbf{merge}\ \overline{x} \Rightarrow A} \qquad \textsc{Exp-Shadow}\ \frac{\begin{array}{c} A_1 = \Sigma(x_1) \qquad A_2 = \Sigma(x_2) \\ A = A_1 \cup \{\, p_2 \in A_2 \mid \nexists p_1 \in A_1.\ \rho_{\mathcal{G}}(\mathrm{tgt}(p_1)) \approx_{\mathrm{d}} \rho_{\mathcal{G}}(\mathrm{tgt}(p_2)) \,\} \end{array}}{\Sigma, M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p, \mathbf{shadow}\ x_1\ x_2 \Rightarrow A}$$

State Evaluation Semantics $\boxed{\mathcal{M}, \mathcal{G}, \mathcal{D}, \mathcal{E} \vdash \mathcal{P}, \mathcal{Y} \Rightarrow \mathcal{A}}$

$$\textsc{Eval-State}\ \frac{\Sigma_0 = \epsilon \quad \forall_{i \in 1 \ldots n}.\, \Sigma_{i-1}, M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p, E_i \Rightarrow A_i \ \ \Sigma_i = \Sigma_{i-1}; (x_i, A_i)}{M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash p, \mathbf{state}\ \overline{x_i := E_i}_{i=1\ldots n} \Rightarrow A_n}$$

Compiled Statix Evaluation Semantics $\boxed{\langle \mathcal{G} \mid \overline{C^*} \rangle \rightarrow \langle \mathcal{G} \mid \overline{C^*} \rangle}$

$$\textsc{Op-Query-SM}\ \frac{Y = \mathsf{init}(M) \qquad M, \mathcal{G}, \mathbf{D}, \approx_{\mathrm{d}} \vdash s, Y \Rightarrow A}{\langle \mathcal{G} \mid \mathbf{query}\ M, \mathbf{D}, \approx_{\mathrm{d}}\ \mathbf{in}\ s \mapsto x; \overline{C} \rangle \rightarrow \langle \mathcal{G} \mid \overline{C}[x/A] \rangle}$$

Figure 7. Operational semantics of the intermediate query resolution language

$$\begin{aligned}
\mathrm{spec}_{\mathsf{M}} &: \mathcal{R} \times \mathcal{O} \rightarrow \mathcal{M} \\
\mathrm{spec}_{\mathsf{M}}(R, <_{\hat{l}}) &:= \mathbf{state\ machine}\ \overline{n\colon Y}\ \textit{where} \\
\overline{\langle n, R', \mathrm{id} \rangle} &= \mathsf{gen_states}(R) \\
\overline{n\colon Y} &= \{\, n\colon \mathrm{spec}_{\mathsf{Y}}(R', <_{\hat{l}}, \mathrm{id}) \mid \langle n, R', \mathrm{id} \rangle \in \overline{\langle n, R', \mathrm{id} \rangle} \,\} \\
\\
\mathrm{spec}_{\mathsf{Y}} &: \mathcal{R} \times \mathcal{O} \times (\mathcal{L} \rightharpoonup \mathcal{N}) \rightarrow \mathcal{Y} \\
\mathrm{spec}_{\mathsf{Y}}(R, <_{\hat{l}}, \mathrm{id}) &:= \mathbf{state}\ \overline{x := E}\ \textit{where} \\
\langle \overline{x := E}, x \rangle &= \mathrm{spec}_{\mathsf{L}}(\mathcal{H}(R) \cup \{\, \$ \mid \varepsilon \in \mathcal{L}(R) \,\}, <_{\hat{l}}, \mathrm{id}) \\
\\
\mathrm{spec}_{\mathsf{L}} &: \mathcal{P}(\hat{\mathcal{L}}) \times \mathcal{O} \times (\mathcal{L} \rightharpoonup \mathcal{N}) \rightarrow \overline{x := E} \times \mathcal{X} \\
\mathrm{spec}_{\mathsf{L}}(\hat{L}, <_{\hat{l}}, \mathrm{id}) &:= \langle (\oplus\, \overline{\overline{x := E}}) \oplus x_L := \mathbf{merge}\ \overline{x}, x_L \rangle\ \textit{where} \\
\overline{\langle \hat{l}, \hat{L}' \rangle} &= \{\, \langle \hat{l}, \hat{L}' \rangle \mid \hat{l} \in \max_{<_{\hat{l}}}(\hat{L}), \hat{L}' = \mathrm{smaller}_{<_{\hat{l}}}(\hat{L}, \hat{l}) \,\} \\
\langle \overline{\overline{x := E}}, \overline{x} \rangle &= \mathsf{unzip}(\{\, \mathrm{spec}_{\mathsf{lL}}(\hat{L}', \hat{l}, <_{\hat{l}}, \mathrm{id}) \mid \langle \hat{l}, \hat{L}' \rangle \in \overline{\langle \hat{l}, \hat{L}' \rangle} \,\}) \\
x_L &= \mathsf{fresh_id}()
\end{aligned}$$

$$\begin{aligned}
\mathrm{spec}_{\mathsf{lL}} &: \mathcal{P}(\hat{\mathcal{L}}) \times \hat{\mathcal{L}} \times \mathcal{O} \times (\mathcal{L} \rightharpoonup \mathcal{N}) \rightarrow \overline{x := E} \times \mathcal{X} \\
\mathrm{spec}_{\mathsf{lL}}(\hat{L}, \hat{l}, <_{\hat{l}}, \mathrm{id}) &:= \langle \overline{x := E}', x_{lL} \rangle\ \textit{where} \\
\langle \overline{x := E}, x_L \rangle &= \mathrm{spec}_{\mathsf{L}}(\hat{L}, <_{\hat{l}}, \mathrm{id}) \\
E_l &= \mathrm{spec}_{\mathsf{l}}(\hat{l}, \mathrm{id}) \\
x_l &= \mathsf{fresh_id}() \\
x_{lL} &= \mathsf{fresh_id}() \\
\overline{x := E}' &= \overline{x := E} \oplus x_l := E_l \oplus x_{lL} := \mathbf{shadow}\ x_L\ x_l \\
\\
\mathrm{spec}_{\mathsf{l}} &: \hat{\mathcal{L}} \times (\mathcal{L} \rightharpoonup \mathcal{N}) \rightarrow \mathcal{E} \\
\mathrm{spec}_{\mathsf{l}}(\$, \mathrm{id}) &:= \mathbf{resolve} \\
\mathrm{spec}_{\mathsf{l}}(l, \mathrm{id}) &:= \mathbf{subenv}\ l\ \mathrm{id}(l)
\end{aligned}$$

Figure 8. Specializer that translates a path well-formedness condition (R) and a label order ($<_{\hat{l}}$) to a state machine (M).

$\mathrm{spec}_{\mathsf{M}}(\mathsf{L}, \emptyset)$	$\mathrm{spec}_{\mathsf{M}}(\mathsf{L}^*, \emptyset)$	$\mathrm{spec}_{\mathsf{M}}(\mathsf{L}^*, \{\, \$ < \mathsf{L} \,\})$

state machine
  n_0:
  e_0 := **subenv** L n_1
  n_1:
  e_0 := **resolve**
	state machine
	  n_0:
	  e_0 := **resolve**
	  e_1 := **subenv** L n_0
	  e_2 := **merge** e_0 e_1
		state machine
		  n_0:
		  e_0 := **resolve**
		  e_1 := **subenv** L n_0
		  e_2 := **shadow** e_0 e_1

Figure 9. Examples of compiled queries, demonstrating how an RE and a label order are translated to a state machine.

head set of that regular expression is { L, $ }. Thus, state n_0 computes both sub-environments, and combines them with the **merge** operator, since there is no ordering between the labels. Compared to `Resolve-L̂`, the lack of an ordering means that both labels are in the max set. Therefore, both environments are computed by a call to `Resolve-Î L̂` with an empty $\hat{L}$ set. This translates to single calls to `Resolve-$` and `Resolve-l`, of which the union is returned by `Resolve-L̂`. In the third example however, there is an ordering between the labels. Therefore, the sub-environments are combined using the **shadow** operator instead. When we compare this with the execution of `Resolve-L̂` again, we see that the max set is { L }, with a smaller set of { $ }. In `Resolve-Î L̂`, this translates to the result of traversing L being shadowed with respect to the local environment, which again corresponds with the behavior of the specialized query. In summary, these examples show how states correspond with derivatives of regular expressions, and how **merge** and **shadow** are used to model different label orders.

5.2 Specializer

In this section, we present the specializer that generates a state machine M for a regular expression R and a label order $<_{\hat{l}}$. We use the following notation. First, the $\oplus$ infix operator appends an item to a list. Similarly, its larger variant flattens a list of lists by concatenating its sublists. In addition, we assume the following helper functions:

$$\begin{aligned} \mathsf{unzip}(\cdot) &: \forall\, \mathcal{T}_1\, \mathcal{T}_2.\ \overline{\mathcal{T}_1 \times \mathcal{T}_2} \to \overline{\mathcal{T}_1} \times \overline{\mathcal{T}_2} \\ \mathsf{fresh\_id}() &: () \to \mathcal{X} \\ \mathsf{gen\_states}(\cdot) &: \boldsymbol{\mathcal{R}} \to \overline{\mathcal{N} \times \boldsymbol{\mathcal{R}} \times \mathcal{L} \rightharpoonup \mathcal{N}} \end{aligned}$$

The unzip function translates a list of 2-tuples in a tuple of lists. Next, the fresh_id primitive generates a fresh variable name at each invocation. Finally, the gen_states function expands a regular expression into its state machine. This state machine is defined as an array of three-tuples, where each tuple (a state) contains (1) a unique name, (2) the derivative of the original regular expression that corresponds to current state, and (3) a partial transition function (defined for the head set of (2)) that maps labels to the identifier of the state. The result of gen_states has two invariants. First, the first state should be the initial state. That is, its second component should be equal to the original input of the function. Second, given two states $\langle n, R, \mathsf{id}\rangle$ and $\langle n', R', \mathsf{id}'\rangle$, and a label l such that $\mathsf{id}(l) = n'$, then $\partial_l R = R'$. This invariant ensures the transitions in the state machine correspond to the original regular expression. We implemented gen_states as follows. We construct a DFA for the input regular expression [19], assign each node in the DFA a name, and then construct an entry for each node. Each entry contains the generated name, the regular expression corresponding to the node [2, 13] and a transition function based on the transitions in the DFA.

```
e0 := subenv L1 n        e0 := subenv L1 n
e1 := subenv L2 n        e1 := subenv L2 n
e2 := shadow e0 e1       e2 := shadow e0 e1
e3 := subenv L1 n
e4 := subenv L3 n        e4 := subenv L3 n
e5 := shadow e3 e4       e5 := shadow e0 e4
e6 := merge e2 e5        e6 := merge e2 e5
```

Figure 10. Example of optimization based on CSE

Fig. 8 shows spec_M, which specializes a regular expression and a label order into a state machine. In spec_M, gen_states generates a DFA, which is then compiled using spec_Y on each state. This function generates a sequence of expressions that computes an environment for each label in the head set of the regular expression argument, including the $ label if the RE matches the empty word (similar to `Resolve-All`). The code for this environment is generated using spec_L, which, similar to `Resolve-L̂`, first computes pairs of max-labels and their set. For each entry in this set, code that computes its shadowed environment is generated. This yields an array of statement sequences and an array of variables. The first array is flattened to obtain a sequence of statements that computes all sub-environments. At the end, a **merge** operation is appended that stores the merged environment in the fresh x_L variable. This sequence is returned, together with x_L, ensuring callees can refer to the new environment. Code to compute a shadowed environment is generated using $\mathrm{spec}_{\mathsf{lL}}$. This function generates code for the environment of its $\hat{L}$ argument, and an expression that creates the environment of its $\hat{l}$ argument. Then, it appends a statement that stores the latter in the fresh x_l variable, and next a statement that creates a shadowed environment. It returns this sequence and the result variable x_{lL}. Finally, code that resolves single labels is generated by $\mathrm{spec}_{\mathsf{l}}$. For the $ label, **resolve** is returned, while for an edge label l, a **subenv** construct is used. The id function is used to find the state the query resolution needs to transition to when traversing l edges. The second invariant on gen_states ensures that $\mathsf{id}(l)$ is the state that implements `Resolve-All` specialized to $\partial_l R$. Therefore, the behavior of this expression is equal to `Resolve-l`.

5.3 Eliminating Common Sub-Environments

Having an intermediate language opens up some additional opportunities for optimization as well. First, consider a case where the label order is defined as $\mathsf{L}_1 < \mathsf{L}_2, \mathsf{L}_1 < \mathsf{L}_3$. Applying the specializer yields the result from the left side of fig. 10. However, the values of e_0 and e_3 are equal. Therefore, we can eliminate the calculation of e_3, and simply use e_0 instead. In general, this corresponds to applying *common sub-expression elimination* (CSE). As this can save redundant computation of sub-environments, this optimization can have a significant impact on the total run time.

$$E \in \mathcal{E} ::= \ldots \mid x \textbf{ else } E \quad \text{Expressions}$$

$$\textsc{Exp-Else-L}\ \frac{\Sigma(x) = A \qquad A \neq \emptyset}{\Sigma, M, \mathcal{G}, \mathbf{D}, \approx_d \vdash p, x \textbf{ else } E \Rightarrow A}$$

$$\textsc{Exp-Else-R}\ \frac{\Sigma(x) = \emptyset \qquad \Sigma, M, \mathcal{G}, \mathbf{D}, \approx_d \vdash p, E \Rightarrow A}{\Sigma, M, \mathcal{G}, \mathbf{D}, \approx_d \vdash p, x \textbf{ else } E \Rightarrow A}$$

$$\begin{aligned}
\text{spec}_{\text{lL}}(\hat{L}, \hat{l}, <_{\hat{l}}, \text{id}) &:= \langle \overline{x := E}', x_{lL} \rangle \textit{ where} \\
\langle \overline{x := E}, x_L \rangle &= \text{spec}_{\text{L}}(\hat{L}, <_{\hat{l}}, \text{id}) \\
E_l &= \text{spec}_{\text{l}}(\hat{l}, \text{id}) \\
x_{lL} &= \text{fresh_id}() \\
\overline{x := E}' &= \overline{x := E} \oplus x_{lL} := x_L \textbf{ else } E_l
\end{aligned}$$

Figure 11. Resolution language extension

5.4 Skipping Fully Shadowed Environments

For the second optimization, recall that **shadow** evaluates to

$$A_1 \cup \{\, p_2 \in A_2 \mid \nexists p_1 \in A_1.\ \rho_{\mathcal{G}}(\text{tgt}(p_1)) \approx_d \rho_{\mathcal{G}}(\text{tgt}(p_2)) \,\}$$

where A_1 is the environment that can shadow declarations in A_2. Now consider the case that all declarations can shadow each other (i.e., $\forall\, d_1\, d_2.\ d_1 \approx_d d_2$). As we have seen in section 3.2, that is the most common situation. In this case, any element in A_1 will shadow A_2 completely. Thus, the **shadow** operator can be simplified to choosing A_1 if it is not empty, and A_2 otherwise. However, that means that we do not need to compute A_2 at all when A_1 is not empty.

This optimization is implemented using a small extension of the query resolution language. An **else** operator, as shown in fig. 11, is added, which has a variable and a sub-expression as operands. When the environment of the variable is non-empty, it is returned (Exp-Else-L). In this case, the right-hand expression is not evaluated. When the variable is empty, that expression is evaluated and its value returned (Exp-Else-R).

The bottom part of fig. 11 shows an alternative version of spec_{lL} that uses this expression. In our specializer, this version is used when $\approx_d$ is trivially satisfied. Similar to the other version, it generates code that computes the environment of $\hat{L}$, an expression that computes the environment of $\hat{l}$ ($E_{\hat{l}}$), and a result variable x_{lL}. To the sequence of code, it appends a statement that stores either x_L or $E_{\hat{l}}$ in x_{lL}. This sequence and the result variable x_{lL} is then returned.

Although not shown in fig. 5, our implementation of the query resolution algorithm contained this optimization as well. However, lifting it to specification compile-time removes the overhead of checking whether the data equivalence condition is trivially true.

Table 1. Benchmark summary. The third and fourth column give the total run times in seconds when using the generic algorithm versus compiled queries, respectively.

Project	#Queries	RT_{gen}	RT_{com}	Speedup
CSV 1.7	14328	7.3	4.5	39%
IO 2.6	73843	19	12	38%
Lang3 3.11	288883	88	46	48%

6 Evaluation

To evaluate the correctness and performance of our optimization, we applied a Statix specification for a subset of Java with pre-compiled queries on the Apache Commons CSV, IO and Lang3 projects. We used these projects as evaluation corpus because they have scope graphs of significant size, and use queries with complex path well-formedness conditions and label orders. Thus, for these projects performance problems are most urgent. All benchmarks are executed on a Linux system with 2 AMD EPYC 7502 32-Core Processors (1.5GHz, 2 threads) and 256GB RAM. All results should be reproducible using the artifact accompanying this paper [31].

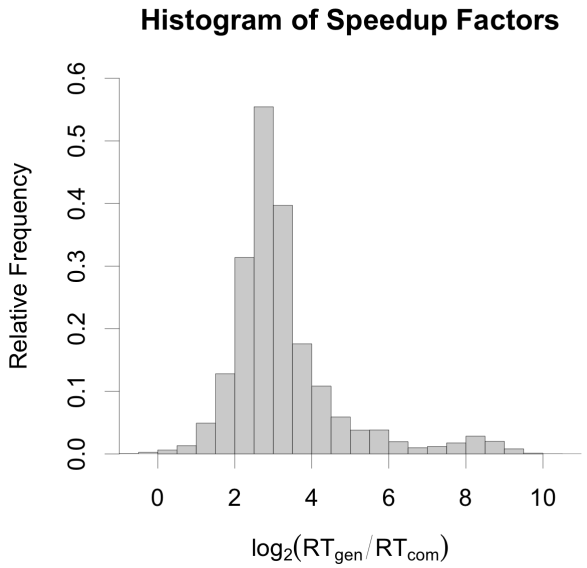
6.1 Correctness

Essential for the validity of an optimization is its correctness. This especially holds for Statix, where soundness with respect to its declarative semantics is essential [17, 25]. Therefore, we validated that for each of the 377054 queries executed by the evaluation projects, the generic algorithm yielded the same answer as the specialized version. As this holds for many complex queries in large scope graphs, we have enough confidence that our approach is correct.

6.2 Performance

In addition, we considered how much our optimization improved performance of query resolution. To this end, we traced all individual queries of the CSV project. For each individual query, we separately measured the performance of the generic and the compiled version. This benchmark used 8 warmup iterations of 500ms, and 5 measurement iterations of 2000ms with 4 parallel threads in throughput mode. A summary of the results is shown in fig. 12. As the speedup distribution is skewed left, but has a relatively heavy tail, the histogram is plot in log-scale, and the upper fence is shown. The individual speedups range from 0.52 (a slowdown) to 1985, with the most weight around the mean of 7.7. This shows that our approach was able to lift much of the computation to specification compile time.

Executing a Statix specification entails more than executing queries. Therefore, these results cannot be interpreted as speedups for Statix-based type checkers. Thus, we benchmarked the total speedup of executing the Java specification [27], using both generic and compiled queries. We used 5



Min	*Q1*	*Q2*	*Q3*	*Max*	Upper fence ($Q2 + 1.5IQR$)
0.52	5.61	7.7	12.9	1985	17.2

Figure 12. Histogram and five-number summary with upper fence of individual query speedup factors. The x-axis of the histogram displays speedup factors in logarithmic scale. In the table, *Q1* – *Q3* represent the quartiles. *IQR* is the interquartile range (*Q3* - *Q1*).

warm-up iterations and 20 measurement iterations in single-shot mode on four cores. The results are shown in table 1 as well. We see that the overall performance of the Java specification improved with 38 – 48%. This is partially due to the elimination of computing label orders and regular expression derivatives, but also to the new optimization discussed in section 5.3, which our intermediate language allowed to implement. Although this does not attain performance comparable to `javac`, it is a significant step in that direction.

6.3 Compilation

In addition, we need to assess the compile-time overhead query specialization introduces. Thus, we measured the time required for specializing queries for the Java specification. We found that it is responsible for 4.6% of the total compilation time, remaining under 11% for all individual files. We consider this overhead acceptable.

6.4 Threats to Validity

There are three threats to the validity of the evaluation that we discuss now. First, regarding the correctness of our approach (section 6.1), we could not verify whether the recorded queries cover all aspects of the algorithm and the intermediate language interpreter. Although we consider it unlikely, it might be the case that queries of the Java specification do not exercise particular important code paths. This limits the guarantees on correctness we provide.

Second, Statix allows interleaving scope graph construction and querying [17, 27]. This is required to support type-dependent name resolution and module systems. To ensure that query answers are valid, some internal scheduling is done. In addition, evaluating data well-formedness conditions can be delayed when a datum contains free unification variables. The benchmarks for individual queries were executed on complete scope graphs, where query scheduling and unification was not needed. Hence, these benchmarks do not account for the overhead that causes. This threat does not concern the type checker benchmarks, thus the conclusion of a significant speedup on Java programs remains valid.

Third, while our Java benchmark set shows the performance characteristics of the approach clearly, it might not be fully representative. Statix is often used for language prototyping and DSLs, which often give simpler specifications and smaller codebases. Still, our evaluation shows that our approach improves the run time of specifications for which performance problems are most relevant.

7 Related Work

In this section, we discuss related work on scope graphs, declarative type checkers and partial-evaluation-based approaches to optimizing interpreters.

7.1 Scope Graphs

Scope graphs were introduced by Néron et al. [11] as a language-parametric model of name binding for languages with non-trivial binding structures. This model was embedded in NaBL2, which is a DSL for declarative specification of type checkers [24]. NaBL2 employed a strict two-phase approach of constraint generation and solving. To extend support to structural types and parameterized types, the scope graph model was generalized and embedded in a new DSL: Statix [25]. Statix allows the definition of user-defined constraints, which, unlike NaBL2, allows interleaving of constraint solving and introduction of new constraints. Rouvoet et al. [17] present a formal operational semantics for Statix, and proved it sound with respect to its declarative semantics. To prove the correctness of query resolution in incomplete scope graphs, the concept of *critical edges* was introduced. Van Antwerpen and Visser [27] generalized this notion to the *scope states* protocol. Using this protocol, they present an actor-based concurrent semantics for Statix.

As scope graphs provide uniform program representation model, they have been used to provide language-parametric editor services, such as semantic completion [15], renaming [9, 10] and inlining [28]. In addition, it has been shown that scope graphs can be used to describe *frames*, which model the structure of run time heaps [16, 30].

7.2 Declarative Type System Specification

In addition to scope graphs, there is other research into declarative type system specification. *Attribute Grammars*, such as JastAdd [5] and Silver [29] allow the definition of attributes on AST nodes. Functions that compute attribute values can reference attributes of other nodes, hence abstracting from traditional AST traversal. However, language implementers need to define the traversal strategy manually, whereas scope graph query resolution derives that from declarative queries. To the best of our knowledge, there is no research about applying partial evaluation to attribute grammar systems.

Although it is more often used for other types of analysis [20–22], *datalog* has been used for the specification of type systems as well [3, 14]. This research was especially aimed at leveraging the good incremental performance of datalog solvers to type checkers. Datalog-based type systems attain a high level of declarativity and good performance, although scope graphs allow easier encoding of complicated name binding patterns. Scholz et al. [18] apply partial evaluation of datalog specifications (targeting C++) to gain efficient program analyzers. This setup is rather similar to ours, although we targeted a tailored intermediate language.

7.3 Optimizing Interpreters by Partial Evaluation

Kleene's *s-m-n theorem* [8] essentially proved partial evaluation possible [7]. Futamura applied this concept on interpreters, establishing fundamental relations between specializers, interpreters, compilers, source programs and executables [4]. The first of his three projections was to specialize an interpreter to a source program, yielding an executable. In section 2, we generalize over this by introducing partial specialization that changes a program in another (interpreted) program of lower complexity. Although they do not name it, Thibault et al. [23] argue that the first Futamura projection can be applied at compile-time as well as at run time.

Brady and Hammond [1] show how to use partial evaluation on DSLs embedded in dependently-typed languages, arguing it is possible to have both correctness and efficiency. Our paper differs in the fact that our query resolution language is (1) not embedded in a dependently-typed language and (2) has a more complicated interpreter, due to the declarative nature of scope graph queries. In addition, our specializer targets a tailored intermediate language, instead of the host language. This allowed us to carefully consider which parts of the algorithm we specialize, but makes the approach harder to transfer to other languages.

Humer et al. [6] introduce *Truffle*, which is an embedded DSL for self-optimizing interpreters. It allows language implementers to annotate specialization possibilities on operations, which are dynamically applied when required. This allows efficient interpretation of (especially) dynamic languages. Truffle-based interpreters are often executed on the *Graal VM* [12], which is an optimizing just-in-time Java compiler. However, Vergu et al. [30] argue that *meta-interpreting* specifications of dynamic semantics "introduces runtime overhead that is difficult to remove by using interpreter optimization frameworks such as the Truffle/Graal Java tools." Using scopes and frames however, optimization of meta-interpreters beyond straightforward application of Truffle can be done [30]. In general, Truffle is particularly aimed at optimization of AST interpreters, which have a very syntax-directed evaluation style. Whether Truffle provides speedup for more algorithmic interpreters is an interesting question for future research.

8 Conclusion

In this paper, we have seen how scope graphs can be used to give high-level encodings of name binding and resolution patterns in programming languages. In addition, we discussed the algorithm that interprets these declarative queries, yielding actual environments in scope graphs of real programs. However, this algorithm turns out to impose significant run time overhead. To eliminate that, we apply partial evaluation to Statix, yielding a specializer that translates declarative queries into a more low-level intermediate representation. These queries can be executed up to 7.7x faster, yielding a speedup of Statix-based type checkers of 38% – 48%. This is a step toward deriving type checkers from declarative specifications that have performance comparable to hand-written type checkers.

Our work suggests that partial evaluation is a powerful technique to optimize execution of programs written in declarative languages. Because interpreters of such languages generally perform complex computations on programs, specialization might reduce run time even more compared to interpreters of more imperative languages. Further establishing the relation between partial evaluation and interpreters for declarative languages seems a promising topic for further research.

Acknowledgments

I would to thank the anonymous reviewers for their helpful feedback, and Casper Bach Poulsen for his extraordinary support when working towards this publication. This paper was written in remembrance of Eelco Visser, who with a few sentences sparked the idea that resulted in this research.

References

[1] Edwin Brady and Kevin Hammond. 2010. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 297–308. https://doi.org/10.1145/1863543.1863587

[2] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494.

[3] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277

[4] Yoshihiko Futamura. 1982. Partial Computation of Programs. In *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings (Lecture Notes in Computer Science, Vol. 147)*, Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa (Eds.). Springer, 1–35. https://doi.org/10.1007/3-540-11980-9_13

[5] Görel Hedin and Eva Magnusson. 2003. JastAdd–an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

[6] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A domain-specific language for building self-optimizing AST interpreters. In *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, Ulrik Pagh Schultz and Matthew Flatt (Eds.). ACM, 123–132. https://doi.org/10.1145/2658761.2658776

[7] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science. ISBN number 0-13-020249-5 (pbk).

[8] Stephen Cole Kleene. 1952. *Introduction to Metamathematics.* North Holland.

[9] Phil Misteli. 2021. *Renaming for Everyone: Language-parametric Renaming in Spoofax.* Master's thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5

[10] Philippe D. Misteli. 2020. Towards language-parametric refactorings. In *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*, Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix (Eds.). ACM, 213–214. https://doi.org/10.1145/3397537.3398476

[11] 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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[12] Oracle. 2021. Graal project. https://www.graalvm.org/

[13] Scott Owens, John H. Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *Journal of Functional Programming* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090

[14] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428195

[15] Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages* 6, OOPSLA (2022), 1–30. https://doi.org/10.1145/3527329

[16] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.20

[17] 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. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428248

[18] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 196–206. https://doi.org/10.1145/2892208.2892226

[19] Michael Sipser. 2012. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.

[20] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[21] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276509

[22] Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. 2018. Incremental overload resolution in object-oriented programming languages. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, Julian Dolby, William G. J. Halfond, and Ashish Mishra (Eds.). ACM, 27–33. https://doi.org/10.1145/3236454.3236485

[23] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. 2000. Static and Dynamic Program Compilation by Interpreter Specialization. *Higher-Order and Symbolic Computation* 13, 3 (2000), 161–178. https://doi.org/10.1023/A:1010078412711

[24] Hendrik van Antwerpen, Pierre Néron, Andrew P. 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, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

[25] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276484

[26] Hendrik van Antwerpen and Eelco Visser. 2021. Scope States (Artifact). *DARTS* 7, 2 (2021). https://doi.org/10.4230/DARTS.7.2.1

[27] Hendrik van Antwerpen and Eelco Visser. 2021. Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.1

[28] Loek Van der Gugten. 2022. *Function Inlining as a Language Parametric Refactoring.* Master's thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:15057a42-f049-4321-b9ee-f62e7f1fda9f

[29] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1-2 (2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[30] Vlad A. Vergu, Andrew P. Tolmach, and Eelco Visser. 2019. Scopes and Frames Improve Meta-Interpreter Specialization. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2019.4

[31] Aron Zwaan. 2022. *Specializing Scope Graph Resolution Queries (Artifact).* Zenodo. https://doi.org/10.5281/zenodo.7189413

[32] Aron Zwaan. 2022. Specializing Scope Graph Resolution Queries: Extended Edition. (2022). https://doi.org/10.48550/arXiv.2210.06121