Language-Parametric Reference Synthesis

Pelsmaeker, Daniel A.A.; Zwaan, Aron; Bach, Casper; Mooij, Arjan J.

DOI
10.1145/3720481

Publication date
2025

Document Version
Final published version

Published in
Proceedings of the ACM on Programming Languages

Language-Parametric Reference Synthesis

DANIEL A. A. PELSMAEKER*, Delft University of Technology, Netherlands
ARON ZWAAN*, Delft University of Technology, Netherlands
CASPER BACH†, Delft University of Technology, Netherlands
ARJAN J. MOOIJ, TNO-ESI, Netherlands and Zürich University of Applied Sciences, Switzerland

Modern Integrated Development Environments (IDEs) offer automated refactorings to aid programmers in developing and maintaining software. However, implementing sound automated refactorings is challenging, as refactorings may inadvertently introduce name-binding errors or cause references to resolve to incorrect declarations. To address these issues, previous work by Schäfer et al. proposed replacing concrete references with *locked references* to separate binding preservation from transformation. Locked references vacuously resolve to a specific declaration, and after transformation must be replaced with concrete references that also resolve to that declaration. Synthesizing these references requires a faithful inverse of the name lookup functions of the underlying language.

Manually implementing such inverse lookup functions is challenging due to the complex name-binding features in modern programming languages. Instead, we propose to automatically derive this function from type system specifications written in the Statix meta-DSL. To guide the synthesis of qualified references we use *scope graphs*, which represent the binding structure of a program, to infer their names and discover their syntactic structure.

We evaluate our approach by synthesizing concrete references for locked references in 2528 Java, 196 ChocoPy, and 49 Featherweight Generic Java test programs. Our approach yields a principled language-parametric method for synthesizing references.

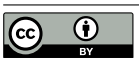
CCS Concepts: • **Software and its engineering** → **Semantics**; *Software maintenance tools.*

Additional Key Words and Phrases: references, synthesis, semantics, scope graphs

ACM Reference Format:
Daniel A. A. Pelsmaeker, Aron Zwaan, Casper Bach, and Arjan J. Mooij. 2025. Language-Parametric Reference Synthesis. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 123 (April 2025), 26 pages. https://doi.org/10.1145/3720481

*Both authors contributed equally to the paper.
†Author's current affiliation: University Of Southern Denmark, Odense, Denmark, casperbach@imada.sdu.dk

Authors' Contact Information: Daniel A. A. Pelsmaeker, Software Technology, Delft University of Technology, Delft, Netherlands, d.a.a.pelsmaeker@tudelft.nl; Aron Zwaan, Software Technology, Delft University of Technology, Delft, Netherlands, a.s.zwaan@tudelft.nl; Casper Bach, Software Technology, Delft University of Technology, Delft, Netherlands, c.b.poulsen@tudelft.nl; Arjan J. Mooij, TNO-ESI, Eindhoven, Netherlands and Zürich University of Applied Sciences, Winterthur, Switzerland, arjan.mooij@tno.nl.



ACM 2475-1421/2025/4-ART123
https://doi.org/10.1145/3720481

1 Introduction

> [P]reserving bindings is at the heart of any refactoring
> that moves, creates, or duplicates code. (Ekman et al. [5, §3])

As software projects evolve, their code is frequently refactored to improve their structure and maintainability. Refactoring often involves copying or moving code from one code unit (such as a class, module, or trait) to another, in a way that preserves the program's behavior. A crucial aspect of behavior-preserving transformations is name binding preservation, to ensure references in refactored code resolve to the same distinct declarations as before. While behavior preservation also needs control- and data flow analysis, name binding preservation can be achieved using only the static semantic analysis of the program. However, due to the sophisticated name binding features found in many modern programming languages, preserving the name resolution semantics of code across transformations is generally challenging.

To illustrate the complexity of reasoning about advanced name binding features, consider the Java program shown in Fig. 1a. If we rename the field `x` (line 2) to `y`, Java's static semantics would cause the reference to `y` on line 7 (in method `foo`) to resolve to the newly renamed field `y` on line 2, rather than the intended declaration of `y` on line 5. This undesired change would alter the name binding structure of the program. To prevent this, the reference to `y` on line 7 should be *qualified* as `Outer.this.y`, as shown in the refactored example in Fig. 1b.

Transformations that require name binding preservation are common across many refactorings, such as those from Fowler's catalog [7]. Yet, manually refactoring code is time-consuming and error-prone. Consequently, many modern Integrated Development Environments (IDEs) provide automated refactorings such as Rename, Inline/Extract Method, and Pull Up/Push Down [7], which attempt to automatically *requalify* references to maintain the program's binding structure.

However, even popular IDEs for mainstream languages struggle to implement sound refactorings. For example, Ekman et al. [5] identify several bugs in Eclipse 3.4 where automated refactorings inadvertently altered the program's binding structure. These errors arise from the difficulty of accurately determining which references need to be fixed and computing the correct requalifications. Not only references in the modified code, but references *throughout the entire code base* may require requalification. Ensuring both *soundness* (preserve name bindings) and *completeness* (finding all possible requalifications) is particularly difficult.

Ekman et al. conclude that these challenges are "not related to the core ingredients of the implemented refactoring, [but] inherent to the complexity of name binding rules in mainstream languages." As a result, existing research on the sound requalification of references is often language-specific, focusing on mainstream languages like Java [31]. Implementing sound automated refactorings for other languages, like Domain-Specific Languages (DSLs) with small language developer teams, can require a prohibitively high effort. As such, a more principled and language-parametric approach to guarantee name binding preservation is needed.

1.1 Locked References

Ekman et al. [5] observe that many bugs in automated refactorings could "be avoided if a set of carefully crafted building blocks were available to refactoring developers." One such building block is *locked references*[1], proposed by Schäfer et al. in previous work [29, 30]. A locked reference is an abstract reference that continues to refer to the same unique declaration even if code is moved or the declaration is renamed. This ensures that transformations cannot cause such a reference to accidentally capture a different declaration.

[1]Terminology introduced by Schäfer et al. [31]. Also referred to as "bound names" [28], "locked names" [29], and "locked bindings" [31]. We use "locked references" throughout this paper.

```
1 class Base {
2   int x = 1;
3 }
4 class Outer {
5   int y = 2;
6   class Inner extends Base {
7     int foo() { return y; }
8   }
9 }
```

(a) Before renaming.

```
1 class Base {
2   int y = 1; // x renamed to y
3 }
4 class Outer {
5   int y = 2;
6   class Inner extends Base {
7     int foo() { return Outer.this.y; }
8   }
9 }
```

(b) After renaming.

Fig. 1. Rename refactoring of a small Java program, where renaming the field x to y on line 2 requires the reference x on line 7 to be appropriately qualified.

```
1 class Base {
2   int x₁ = 1;
3 }
4 class Outer {
5   int y₂ = 2;
6   class Inner extends Base {
7     int foo() { return ⟪→y₂⟫; }
8   }
9 }
```

(a) After locking references, before renaming.

```
1 class Base {
2   int y₁ = 1; // x renamed to y
3 }
4 class Outer {
5   int y₂ = 2;
6   class Inner extends Base {
7     int foo() { return ⟪→y₂⟫; }
8   }
9 }
```

(b) After renaming, before unlocking references.

Fig. 2. Intermediate steps for performing the Rename refactoring from Fig. 1 using locked references. After locking the relevant reference y to declaration y_2 (a) and performing the transformation (b), our approach would synthesize a solution for the locked reference and obtain Fig. 1b.

The following diagram summarizes program transformation with locked references:

$$\mathcal{P} \xrightarrow{\text{lock}} \mathcal{P}_{\text{locked}} \xrightarrow{\text{transform}} \mathcal{P}'_{\text{locked}} \xrightarrow{\text{unlock}} \mathcal{P}'$$

Before refactoring, we first 'lock' each relevant concrete reference by replacing it with a locked reference pointing to the original declaration. In Fig. 2a we replace the concrete reference y (line 7) with a locked reference $\langle\!|{\to}y_2|\!\rangle$ to the declaration y_2 on line 5.[2] (We use subscript indices to distinguish different occurrences of the same name, but the indices are not part of the syntax.)

Next, we 'transform' the program as required for the refactoring, renaming declarations and moving code. In Fig. 2b, the declaration on line 2 is renamed to y. Finally, we 'unlock' each locked reference in the program by replacing it with a *synthesized* concrete reference that unambiguously resolves to the intended declaration. In this case, unlocking replaces the locked reference with `Outer.this.y`, maintaining the name binding semantics of the program (see Fig. 1b).

Every step in this pipeline gives rise to challenges, but in this paper we focus on the key challenge of synthesizing concrete references when unlocking locked references. The program should remain well-typed and synthesized references should resolve to their intended declarations. Separating name binding preservation from the transformation guarantees that refactorings preserve name bindings, and also makes it easier to implement refactorings.

[2]Our syntax for locked references $\langle\!|{\to}d|\!\rangle$ is inspired by the syntax Omar et al. [18] use for holes.

There are numerous potential applications of reference synthesis. In the line of work by Schäfer et al. [30, 31], it can be applied to implement sound (editor) refactorings. Furthermore, it provides a powerful transformation tool for implementing sound transformations of DSL programs or performing large-scale codebase transformations aiming to improve the overall code quality. However, one can also envision *user-extensible refactoring tools* (such as presented by Li and Thompson [15]) or *transformation languages* (such as IntelliJ's structural search and replace) that need to preserve name bindings. Finally, it could be used to build an editor service that suggests fixes for type errors (e.g., QUICK FIX in Eclipse).

1.2 Language-Parametric Reference Synthesis

Following Schäfer et al. [30, 31], a reference synthesis function can be thought of as the right inverse of a reference resolution function. That is, if *QRef* is the set of qualified references, *Decl* is the set of uniquely identified declarations, and the function $\mathsf{resolve}_p : \mathit{QRef} \rightharpoonup \mathit{Decl}$ resolves a reference at some location p in the program, then locked reference synthesis should be a function $\mathsf{synthesize}_p : \mathit{Decl} \rightharpoonup \mathit{QRef}$ such that $\mathsf{resolve}_p(\mathsf{synthesize}_p(d)) = d$ for any p.[3] The lock function uses resolve to obtain the target declaration when replacing a concrete reference with a locked reference, and conversely, unlock uses the synthesize function to replace the locked reference with a concrete reference.

The reference synthesis pipeline shown above is conceptually *language-parametric*. However, as discussed before, implementing correct reference synthesizers manually is error-prone and time-consuming. In this paper, we present a *language-parametric* approach to derive the synthesize function automatically from declarative type system specifications, letting language designers generically synthesize valid concrete references for programs with multiple locked references. The goal of reference synthesis is to find for each locked reference in a program a valid concrete reference that resolves to the intended declaration.

We derive the synthesize function from *only* a declarative specification of the language's static semantics, specified using the *Statix* specification language [27, 36]. Statix allows syntax-directed typing rules to be specified, uses *scope graphs* as a declarative model of static name binding and name resolution [17], and generates executable type checkers by interpreting specifications as constraint programs. In our implementation of synthesize, we reinterpret the *name resolution queries* from the specification to infer syntax for concrete references that resolve to a particular target declaration, guaranteeing name binding preservation. We reuse the Statix solver (see §2) to validate that the syntax we infer is *sound* with respect to the typing rules and represents a reference that resolves to the intended declaration.

This paper makes the following technical contributions:

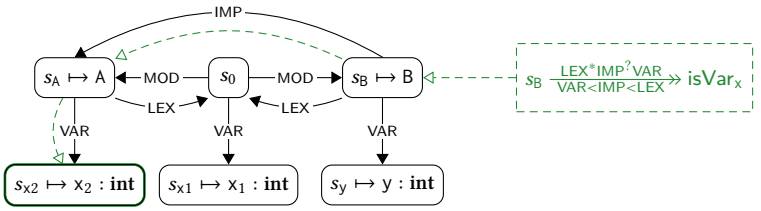
- We present a language-parametric implementation of the synthesize function (see §4 and §5). This function automatically synthesizes concrete references that resolve to the specified declaration, and that are sound with respect to a type system specification written in Statix.
- We evaluate our implementation on 2773 test programs of Java, ChocoPy, and Featherweight Generic Java (§6). Our results demonstrate that our approach applies to mainstream languages with complex name binding semantics without modifying their typing rules.

We first (§2) introduce scope graphs and Statix. Next, in §3 we illustrate our reference synthesis algorithm. Then, in §4 we give an operational semantics of our synthesize function's implementation, and the heuristics we apply in §5. We evaluate our implementation in §6, and discuss related work in §7. We conclude in §8.

[3] Schäfer et al. use "lookup" instead of "resolve" and "access" instead of "synthesize", but the idea is the same.

```
1  var x1 = 42
2  mod A {
3    var x2 = 0
4  }
5  mod B {
6    import A::*
7    var y = x
8  }
```

(a) Program.



(b) Scope graph for the program in Fig. 3a.

Fig. 3. An example LM [17] program and its scope graph, where boxes and arrows represent scopes and reachability relations between scopes, respectively. The dashed box represents a query and the dashed arrows its resolution path to x_2, the second occurrence of a declaration named x.

2 Scope Graphs and Statix

This paper presents a *language-parametric* approach to synthesizing concrete references. To this end, we build on existing work: (1) *scope graphs* as a language-parametric model of name binding, and (2) *Statix* as a uniform representation of typing rules.

In this section we first describe what scope graphs are (§2.1), and how they let us resolve references via graph search (§2.2). Then we provide a high-level introduction to the Statix language (§2.3) and its constraint solver (§2.4).

2.1 Scope Graphs

The example program in Fig. 3a contains two declarations named x, namely x_1 on line 1 and x_2 on line 3, and a named reference x on line 7.[4] The question is: does x refer to declaration x_1 or x_2? Either can be true, depending on the semantics of the programming language.

Scope graphs [17, 27, 34, 36, 42] offer a uniform model for name resolution that supports sophisticated name binding patterns in programming languages. As their name suggests, scope graphs model the scoping structure of programs as graphs. Such graphs let us model both nested and recursive scoping structures, and name resolution policies as graph search queries.

To illustrate this, consider the program and its scope graph in Fig. 3. The nodes in the graph represent scopes: s_0 represents the *global scope*, while s_A and s_B represent the scopes of modules A and B, respectively. Scopes s_{x1}, s_{x2}, and s_y represent named declarations. A scope s may have data d associated with it, written as $s \mapsto d$, such as the name of the module that they correspond to or the name and type of their declaration. As we shall see later, associating scopes with names lets us define *name resolution queries* that resolve module names to their corresponding scopes.

Edges between scopes represent *reachability* relations. Queries can follow these edges to reach other scopes and declarations. The two module scopes are reachable from the global scope via MOD edges, representing the fact that A and B are modules declared in the global scope s_0. The module scopes s_A and s_B are also lexical children of the global scope, so each is connected via a LEX edge to s_0. VAR edges connect scopes to declared names. Due to the wildcard `import A::*` on line 6 (which imports all members of the module A) module scope s_B is connected to module scope s_A via an IMP edge, making all declarations in module A reachable from module B.

[4] We use *subscript indices* to distinguish different *occurrences* of a particular name. For example, x_2 uniquely identifies one of the declarations named x.

2.2 Scope Graph Queries

We define name resolution as *queries* in scope graphs. Resolving a query entails finding all paths from this source scope to matching declarations. To explain the syntax of a name resolution query, we take the query shown in the dashed box on the right of the graph in Fig. 3b:

$$s_{\mathsf{B}} \xrightarrow[\mathsf{VAR}<\mathsf{IMP}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{IMP}^?\mathsf{VAR}} \!\!\!\rightarrow\!\!\!\!\rightarrow \mathsf{isVar}_{\mathsf{x}}$$

Here, s_{B} is the initial scope of the graph search, and $\mathsf{isVar}_{\mathsf{x}}$ is a filter that ensures only declarations with name x are selected. The regular expression $\mathsf{LEX}^*\mathsf{IMP}^?\mathsf{VAR}$ is a *reachability policy* declaring which declarations are reachable; i.e., those declarations we can reach by following a sequence of labeled edges that match the regular expression. The path ordering $\mathsf{VAR} < \mathsf{IMP} < \mathsf{LEX}$ is a *visibility policy* used to disambiguate which reachable names are visible, i.e., to model shadowing. For example, both $s_{\mathsf{x}1}$ and $s_{\mathsf{x}2}$ are reachable in Fig. 3b. However, the order prefers IMP edges over LEX edges, so the only valid path through the graph is the path to $s_{\mathsf{x}2}$.

2.3 Statix Rules and Constraints

In classical typing rules, terms are typed relative to one or more *typing contexts* [25], or typed via *symbol tables* [1] or *class tables* [10]. Following existing work [26, 27, 36], we can define typing rules in a similar style, but with terms typed relative to one or more scopes in a scope graph instead. The constraint language Statix [27, 36, 38] lets us declare such inference rules using a syntax inspired by logic programming. Type system specifications written in Statix have a declarative interpretation, specifying a class of well-typed programs. Alternatively, specifications can be used operationally to type check programs by constructing a scope graph and resolving references by traversing it. This subsection highlights the main features of Statix rules and constraints. For a more detailed breakdown of the syntax, we refer to the discussion in §4.1 and the work of Rouvoet et al. [27].

The rules in Fig. 4 show a representative subset of the Statix rules we derived for LM, a toy language from [17] used throughout this paper. The figure declares rules for five different typing relations: typeOfExpr, memberOk, modOk, importOk, and scopeOfMod. Each rule has a conclusion on the left of an arrow (←), and a premise given by one or more constraints on the right. For example, the rule T-Add states that the expression $e_1 + e_2$ has type **int** in scope s, if both e_1 and e_2 have type **int** under the same scope s. Rule T-QRef is a more complex example, where a qualified module access expression $a.x$ has type T when the a resolves to a module (asserted by the predicate constraint $\mathsf{scopeOfMod}(s, a, s_m)$), and x resolves from that module to a declaration of type T (asserted by the query constraint $s_m \xrightarrow{\mathsf{VAR}} \!\!\!\rightarrow\!\!\!\!\rightarrow \mathsf{isVar}_x$).

The rules in Fig. 4 do not mention the underlying scope graph explicitly. Instead, premises of rules assert requirements on the scope graph structure, such as the existence of scopes with associated data ($\nabla s_x \mapsto x : T$) and edges ($s \xrightarrow{\mathsf{VAR}} s_x$) and the ability to resolve query constraints. A program is well-typed when a *minimal* scope graph exists that satisfies each such assertion and query. Minimality implies that the scope graph only has the scopes and edges asserted by the rules of a program: no extraneous edges or scopes. There exists a solver for Statix constraints that computes this minimal scope graph, which we discuss in the next section.

2.4 Statix Constraint Solver

Following Rouvoet et al. [27], the operational semantics of Statix is given by a constraint solver that soundly constructs and queries scope graphs, and uses unification to solve equality constraints. The reference synthesis approach we illustrate in §3 is sound by construction because it builds on this operational semantics. We defer a deeper discussion of the operational semantics to §4.

$$\mathsf{typeOfExpr}(s, n, \mathbf{int}) \leftarrow \mathsf{emp} \quad \text{(T-Num)}$$

$$\mathsf{typeOfExpr}(s, e_1 + e_2, \mathbf{int}) \leftarrow \mathsf{typeOfExpr}(s, e_1, \mathbf{int}) * \mathsf{typeOfExpr}(s, e_2, \mathbf{int}) \quad \text{(T-Add)}$$

$$\mathsf{typeOfExpr}(s, x, T) \leftarrow s \xrightarrow[\mathsf{VAR}<\mathsf{IMP}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{IMP}^?\mathsf{VAR}} \mathsf{isVar}_x \mapsto \{(\_, x : T)\} \quad \text{(T-Var)}$$

$$\mathsf{typeOfExpr}(s, a.x, T) \leftarrow \exists s_m.\ \mathsf{scopeOfMod}(s, a, s_m) * s_m \xrightarrow{\mathsf{VAR}} \mathsf{isVar}_x \mapsto \{(\_, x : T)\} \quad \text{(T-QRef)}$$

$$\mathsf{memberOk}(s, \mathbf{var}\ x = e) \leftarrow \exists T\, s_x.\ \mathsf{typeOfExpr}(s, e, T) * \nabla s_x \mapsto x : T * s \xrightarrow{\mathsf{VAR}} s_x \quad \text{(M-Var)}$$

$$\mathsf{modOk}(s, \mathbf{mod}\ a\ \{\ \overline{imp}\ \overline{mem}\ \}) \leftarrow \exists s_m.\ \nabla s_m \mapsto a * s_m \xrightarrow{\mathsf{LEX}} s * s \xrightarrow{\mathsf{MOD}} s_m * \mathsf{importsOk}(s_m, \overline{imp}) * \mathsf{membersOk}(s_m, \overline{mem}) \quad \text{(M-Mod)}$$

$$\mathsf{importOk}(s, \mathbf{import}\ a::*) \leftarrow \exists s_m.\ \mathsf{scopeOfMod}(s, a, s_m) * s \xrightarrow{\mathsf{IMP}} s_m \quad \text{(D-ImportOk)}$$

$$\mathsf{scopeOfMod}(s, x, s_m) \leftarrow \exists p.\ s \xrightarrow[\mathsf{MOD}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{MOD}} \mathsf{isMod}_x \mapsto \{(p, x)\} * s_m \stackrel{?}{=} \mathsf{tgt}(p) \quad \text{(S-Mod)}$$

$$\mathsf{scopeOfMod}(s, a.x, s_m) \leftarrow \exists p\, s'_m.\ \mathsf{scopeOfMod}(s, a, s'_m) * s'_m \xrightarrow{\mathsf{MOD}} \mathsf{isMod}_x \mapsto \{(p, x)\} * s_m \stackrel{?}{=} \mathsf{tgt}(p) \quad \text{(S-QMod)}$$

$$\mathsf{isVar}_x \triangleq \lambda x'.\ \exists T.\ (x{:}\ T) \stackrel{?}{=} x'$$

$$\mathsf{isMod}_x \triangleq \lambda x'.\ x \stackrel{?}{=} x'$$

Fig. 4. A subset of the typing rules of LM, a toy language from [17] used for the examples in this paper.

The Statix solver will solve as many constraints as possible, yielding either a state with no unsolved constraints (i.e., the program type-checks), a state that derives false (i.e., the program does not type-check), or a *stuck* state, where the solver does not have enough information to solve the remaining constraints. There are two reasons why constraints get stuck: either (1) it is not sufficiently instantiated, or (2) it is a query constraint which is not yet guaranteed to yield a *stable answer*. For (1), the solver will only expand a predicate such as $\mathsf{typeOfExpr}(x, y, z)$ once x, y, and z are sufficiently instantiated such that only a single rule matches. Similarly, it will only run and solve a query constraint once its *source scope* (e.g., s in $s \xrightarrow[\mathsf{VAR}<\mathsf{IMP}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{IMP}^?\mathsf{VAR}} \mathsf{isVar}_x$) and *data well-formedness predicate* (e.g., isVar_x) are ground. In case (2), a query gets stuck when it needs to run but another unsolved constraint might add a scope graph edge that could invalidate the query. The Statix solver implements guards that detect these cases [27].

Our synthesize function runs the Statix solver on a program with holes, where each hole is represented by a free unification variable that maps to a target scope representing the hole's intended target declaration. The unification variables cause the Statix solver to get stuck on predicate and query constraints directly related to the holes. Once a stuck state is reached, our reference synthesis approach extends the usual operational semantics of Statix with the ability to use the typing rules of a language to refine the holes of the program, and use the Statix solver to verify the solution. Once the term of a hole becomes ground and all constraints in the state have been solved, we have successfully synthesized a concrete reference.

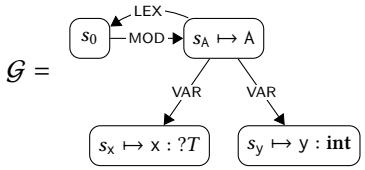
We will illustrate how the Statix solver, scope graph, and typing rules are used in our reference synthesis algorithm in the next section, and discuss the operational semantics of Statix and our extension in more detail in §4.

```
1  mod A {
2    var x = (|→y|)
3    var y = 1
4  }
```

(a) Program with locked reference.

```
1  mod A {
2    var x = ?E
3    var y = 1
4  }
```

(b) Program with hole.



(c) Scope graph.

Fig. 5. Small example program and its scope graph

3 Reference Synthesis by Example

As the name suggests, *reference synthesis* is used to synthesize a concrete (qualified) *reference* to a given declaration. References in many languages take the shape $x_1.x_2.\cdots.x_n$, modulo syntax. Here, the first name x_1 is resolved from the place in the program where the reference occurs, and subsequent names x_i are resolved relative to wherever the previous qualifiers $x_1.\cdots.x_{i-1}$ led. The final name x_n leads to the target declaration.

This informal definition of a reference encompasses many syntactic constructs that we intuitively recognize as (qualified) references across languages, for example `Person.this.name` in Java, `std::option::Option` in Rust, and `ID IN CUSTOMER IN LAST-TRANSACTION` in Cobol. On the other hand, according to our definition, syntax like `List<String>` in Java does not constitute a reference: it is a parameterized type, akin to how a method call `foo(x, y)` would also not be considered a reference. We give a more precise definition of a reference in §4.

$$\mathcal{P} = X[\overline{r}] \xrightarrow{\text{lock}} X[\overline{(|\to d|)}] \xrightarrow{\text{transform}} X'[\overline{(|\to d|)}] \xrightarrow{\text{unlock}} X'[\overline{r'}] = \mathcal{P}'$$

The above diagram reiterates program transformation with locked references. Initially, we have a program $\mathcal{P}$, which can be represented as a context[5] X with references $\overline{r}$ occurring in it. First, we lock relevant references. That is, we replace concrete references with locked references that durably remember which declaration they point to, regardless of where the locked reference occurs in the program ($X[\overline{(|\to d|)}]$). Then we transform the program, possibly moving code around ($X'[\overline{(|\to d|)}]$). Finally, we unlock the locked references: synthesizing their concrete references ($\overline{r'}$) and plugging them into the program to yield the concrete transformed program ($\mathcal{P}' = X'[\overline{r'}]$).

In this section, we use a simple program with a locked reference, shown in Fig. 5, to illustrate the semantics of our synthesize function. The idea is to model locked references as holes given by unification variables, and strategically apply typing rules to infer a substitution for each hole. The strategy used to apply typing rules must guarantee that inferred substitutions correspond to references that resolve as intended.

3.1 Initial Constraint Solving

Consider the example in Fig. 5a, where for the locked reference (|→y|) we want to synthesize a concrete reference that must resolve to variable `y`'s declaration scope in the underlying scope graph. Valid concrete references that our approach could yield include `y` and `A.y`.

The first step of our approach is to replace each locked reference in the program by a hole, represented by a fresh unification variable, shown in Fig. 5b. Then we use the original language's static semantic rules and run the Statix solver on the input program. The presence of holes causes

[5]We can regard a 'context' as a zipper-like structure over an AST.

the solver to yield a stuck state, where the solver neither has enough information to solve all constraints nor can derive false, due to the free unification variables.

In our example, the Statix solver will recursively expand predicates and solve scope graph constraints to yield a solver state, shown below, with the inferred scope graph $\mathcal{G}$ (see Fig. 5c) and a single constraint that is stuck because Statix cannot infer which rule to apply to expand the predicate. Additionally, in the state we record that the unification variable $?E$ is associated with hole h, and that hole h should become a concrete reference that resolves to the scope s_y, which is the scope associated with the locked reference's target declaration y in $\mathcal{G}$.

$$\left\langle \mathcal{G} \;\middle|\; \mathsf{typeOfExpr}(s_A,\ ?E\ ,?T) \;\middle|\; ?E \mapsto h \;\middle|\; h \mapsto (s_y, ?E) \right\rangle$$

The state has the form $\langle\, \mathcal{G} \,|\, \overline{C} \,|\, U \,|\, H \,\rangle$. Following Rouvoet et al. [27], the solver state is given by the (partially constructed) scope graph $\mathcal{G}$ and the set of yet unsolved constraints $\overline{C}$. To support reference synthesis, we have augmented the solver state with U and H. U is a partial function that maps free unification variables to hole identifiers where applicable. H maps each hole identifier in a program to a pair (s_t, t) of the current target scope s_t and the term t synthesized for the hole so far. Note that s_t changes as we synthesize qualifiers for the concrete reference.

3.2 Forking States

By default, the Statix solver only expands a constraint when there is exactly one possible expansion, and otherwise the constraint gets stuck. To support reference synthesis we augment the solver to allow solver states to be *forked*. This way we can obtain the solver state for each possible expansion of a constraint. This way we can speculatively apply each possible expansion of a constraint, obtaining a solver state. Forked solver states that fail are discarded, but those that can be successfully solved represent programs for which we have synthesized a valid reference.

For the stuck state discussed above, we can speculatively expand the stuck typeOfExpr predicate constraint and fork the state for each of the typeOfExpr rules shown in Fig. 4, yielding different states. However, some of those rules will not lead to well-formed references, and therefore we only expand to rules that could yield a reference. For the simple LM language shown in Fig. 4, the relevant rules are T-Var and T-QRef. We fork the solver state, and discuss how each of these two rules leads to a synthesized reference.

3.3 Expanding Query Constraints

Applying the rule T-Var and solving the constraints as far as possible yields the first forked solver state with one stuck query constraint:

$$\left\langle \mathcal{G} \;\middle|\; s_A \xrightarrow[\mathsf{VAR}<\mathsf{IMP}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{IMP}^?\mathsf{VAR}} \gg \mathsf{isVar}_{?x} \mapsto \{(\_, ?x : ?T)\} \;\middle|\; ?x \mapsto h \;\middle|\; h \mapsto (s_y,\ ?x\ ) \right\rangle$$

As the free variable $?x$ in the stuck constraint is related to hole h, we can infer that the query is also related to hole h: attempting to solve the constraint could be fruitful for synthesizing a reference to the target scope s_y. Therefore, reference synthesis searches for valid scope graph paths from s_A to the intended target the scope s_y, while respecting the reachability regex and visibility ordering of the query. There is exactly one such path, namely the one-step path traversing the VAR edge from s_A to s_y. This implies that $?x$ = y and $?T$ = **int**. This solves all constraints and results in the mapping $h \mapsto (s_y, \mathsf{y})$. As all constraints have been solved and the term for the hole is ground, the unqualified reference y is returned as a solution.

3.4 Qualified Reference

In the second forked solver state, applying the rule T-QRef and solving constraints yields the following solver state with a stuck predicate constraint and a stuck query constraint, where the term for the hole is representing a possible *qualified* reference $?a\,.\,?x$:

$$\left\langle \mathcal{G} \;\middle|\; \begin{array}{c} \mathsf{scopeOfMod}(s_\mathsf{A}, ?a, ?s_m) \\ ?s_m \xrightarrow{\mathsf{VAR}} \mathsf{isVar}_{?x} \mapsto \{(\_, ?x : ?T)\} \end{array} \;\middle|\; \begin{array}{l} ?a \mapsto h \\ ?x \mapsto h \end{array} \;\middle|\; h \mapsto (s_\mathsf{y},\ ?a\,.\,?x) \right\rangle$$

Next, we expand the scopeOfMod predicate. One of the possible expansions (using rule S-Mod) yields the following state:

$$\left\langle \mathcal{G} \;\middle|\; \begin{array}{ll} s_\mathsf{A} \xrightarrow[\mathsf{MOD}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{MOD}} \mathsf{isMod}_{?a} \mapsto \{(?p, ?a)\} & (1) \\ ?s_m \stackrel{?}{=} \mathsf{tgt}(?p) & (2) \\ ?s_m \xrightarrow{\mathsf{VAR}} \mathsf{isVar}_{?x} \mapsto \{(\_, ?x : ?T)\} & (3) \end{array} \;\middle|\; \begin{array}{l} ?a \mapsto h \\ ?x \mapsto h \end{array} \;\middle|\; h \mapsto (s_\mathsf{y}, ?a\,.\,?x) \right\rangle$$

The solver state has two stuck query constraints: the module query (1) and the variable query (3). Both query constraints have unification variables that relate to the hole h, so we cannot know which of these query should resolve to the target scope s_y. Therefore, we fork the solver state again: one branch where we attempt to expand query (1) and one where we attempt to expand query (3). Only the fork that attempts to expand the variable query (3) to the target s_y will succeed, so in this example we continue with that branch.

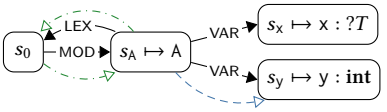
Because of the mapping $?x \mapsto h$, query (3) may be relevant for resolving to the target scope s_y. Thus, we inspect the scope graph and search for well-formed paths to s_y. Since the query in question has the unification variable $?s_m$ as its source, we need to look for paths with *any* possible source. For the graph Fig. 5c in the solver state, only the one-step path from s_A to s_y matches the regular expression of the query. Hence, the query is resolved by substituting s_A for the scope variable $?s_m$. Next, we make s_A the new target for the hole h, since we assume that the source scope was not ground because the query forms part of a qualified reference; i.e., a sequence of paths. Eventually, we solve the constraint by substituting $?s_m \mapsto s_\mathsf{A}$ and $?x \mapsto \mathsf{y}$.

$$\left\langle \mathcal{G} \;\middle|\; \begin{array}{c} s_\mathsf{A} \xrightarrow[\mathsf{MOD}<\mathsf{LEX}]{\mathsf{LEX}^*\mathsf{MOD}} \mathsf{isMod}_{?a} \mapsto \{(?p, ?a)\} \\ s_\mathsf{A} \stackrel{?}{=} \mathsf{tgt}(?p) \end{array} \;\middle|\; ?a \mapsto h \;\middle|\; h \mapsto (\,s_\mathsf{A} \cdot s_\mathsf{y}, ?a\,.\,\mathsf{y}\,) \right\rangle$$

In addition to refining the hole term to $?a\,.\,\mathsf{y}$, the target scope was also refined to s_A. The new problem to be solved is to find the qualifier that resolves to s_A. Using the same principles as illustrated above, the remaining stuck query can be expanded. This will solve the remaining constraints and make the hole term ground, yielding $\mathtt{A.y}$ as the solution.

$$\left\langle \mathcal{G} \;\middle|\; \emptyset \;\middle|\; \emptyset \;\middle|\; h \mapsto (\,s_\mathsf{A} \cdot s_\mathsf{A} \cdot s_\mathsf{y}, \mathtt{A.y}) \right\rangle$$

Following to our definition of a reference, the solution `A.y` is a composite path in the scope graph from the source scope s_A, $s_A \dashrightarrow s_0 \dashrightarrow s_A$ to the scope of qualifier `A`, followed by $s_A \dashrightarrow s_y$ to the scope of the intended target declaration `y`, as shown here:



The next section formally defines the approach illustrated above. In §5 we describe the heuristics we apply to make the approach usable in practice. We evaluate our synthesize function on test programs with locked references in §6.

4 Operational Semantics

The previous section illustrated our approach to synthesize concrete references using an extension of the Statix solver. This section presents an operational semantics that defines that extension.

4.1 Syntax of Statix

The syntax of Statix terms and constraints, defined in Fig. 6, follows Rouvoet et al. [27] and has a separation-logic-inspired flavor, as the declarative semantics of Statix constraints is defined using separation logic. We refer to Rouvoet et al. for the details of this declarative semantics, and focus on the operational semantics instead in §4.2 and §4.3.

The syntax uses these distinct enumerable sets of symbols: *TermConstructor* for term constructor symbols, *Var* for term variables, *SetVar* for set variables, *PredSymbol* for names of predicates (such as typeOfExpr in Fig. 4), *Scope* for scope graph node identifiers, *Label* for scope graph edge labels, *RegEx* is the set of regular expressions over words comprised of label symbols, and *PartialOrd* is the set of partial orders on label symbols.

In the *Constraint* syntax, emp is the trivially satisfiable constraint, akin to a true constraint in a traditional logic. Conversely, false is never satisfiable. $C_1 * C_2$ is a *separating conjunction*, where the declarative and operational semantics of Statix guarantees C_1 and C_2 construct separate scope graph fragments. However, the reader can approximately think of $C_1 * C_2$ as traditional logic conjunction. The constraint $\exists x.\, C$ asserts the existence of some term named x, which may be referenced and constrained by C. $\mathsf{single}(t, \underline{t})$ asserts that set term $\underline{t}$ is a singleton set whose element is equal to t, while $\forall x$ in $\underline{t}.\, C$ asserts that C holds for all its elements x. $\nabla t_1 \mapsto t_2$ asserts that the scope graph contains a scope identified by t_1 and with associated data t_2, whereas $t_1 \xrightarrow{l} t_2$ asserts that the scope identified by t_1 is connected via an l-labeled edge to the scope identified by t_2.

The syntax of queries is $t \xrightarrow[o]{r} \!\!\!\!\gg \lambda x.\, E \mapsto z.\, C$. Here, the term t represents a source scope term; r represents a regular expression that determines reachability; o is a partial order that determines visibility; $\lambda x.\, E$ is a *data-well-formedness constraint* which characterizes whether a target scope and its associated data matches the query; z is a set variable that will be bound in C to the result of the query. The syntax used by Rouvoet et al. provides a separate constraint for applying the visibility ordering o. We include this ordering as a part of the query, following the syntax used by the implementation of Statix found in the Spoofax Language Workbench [11].[6]

Also in contrast to Rouvoet et al., we distinguish equality constraints (ranged over by E) from plain constraints (C). This way, data well-formedness predicates of queries ($\lambda x.\, E$) use constraints that can only inspect terms and data, but cannot extend the scope graph. Another difference from Rouvoet et al. is that we define the semantics of predicate constraints. The constraint $P(t^*)$

[6] https://spoofax.dev/

$$
\begin{array}{llll}
f \in \mathit{TermConstructor} & & s \in \mathit{Scope} \\
x \in \mathit{Var} & & l \in \mathit{Label} \\
z \in \mathit{SetVar} & & r \in \mathit{RegEx} \\
P \in \mathit{PredSymbol} & & o \in \mathit{PartialOrd}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Term} \ni t & ::= & x \mid f(t^*) \mid l \mid s \\
\mathit{SetTerm} \ni \underline{t} & ::= & z \mid \zeta \\
\mathit{SetLit} \ni \zeta & ::= & \emptyset \mid \{t\} \mid \zeta \sqcup \zeta \\
\mathit{Constraint} \ni C & ::= & \mathsf{emp} \mid \mathsf{false} \mid C * C \mid \exists x.\, C \mid \mathsf{single}(t, \underline{t}) \mid \forall x \text{ in } \underline{t}.\, C \\
& \mid & \nabla t \mapsto t \mid t \xrightarrow{l} t \mid t \xrightarrow[o]{r}\!\!\!\!\rightarrow \lambda x.\, E \mapsto z.\, C \mid E \mid P(t^*) \\
\mathit{EqConstraint} \ni E & ::= & t \stackrel{?}{=} t \mid \mathsf{dataOf}(t, t) \mid E * E \mid \exists x.\, E
\end{array}
$$

Fig. 6. Syntax of Statix terms and constraints.

$$
\begin{array}{rcl}
& & h \in \mathit{Hole} \\
\mathit{Configuration} \ni \kappa & ::= & \left\langle \mathcal{G} \,\middle|\, C^* \,\middle|\, U \in (\mathit{Var} \rightharpoonup \mathit{Hole}) \,\middle|\, H \in (\mathit{Hole} \rightharpoonup (s^* \times t)) \right\rangle \\
\mathit{ScopeGraph} \ni \mathcal{G} & ::= & \langle S \subseteq \mathit{Scope},\ R \subseteq (\mathit{Scope} \times \mathit{Label} \times \mathit{Scope}),\ \rho \in (\mathit{Scope} \rightharpoonup \mathit{Term}) \rangle
\end{array}
$$

Fig. 7. Syntax of Statix configurations and scope graphs.

represents an invocation of a user-specified predicate, such as those from Fig. 4. The rules we discuss next are parameterized by a specification $\mathbb{S}$ comprising rules of the form $P(t^*)$.

In the syntax of constraints and terms in Fig. 6 and throughout the paper, we use $\overline{\bullet}$ notation to represent (possibly empty) sequences, and $\bullet^*$ notation to represent their syntax. For example, $P(t^*)$ represents the syntax of a predicate symbol followed by a parenthesized sequence of terms. We use $x; y$ for sequences that can be freely reordered (e.g., $x; y \approx y; x$) and $x \cdot y$ for sequences that cannot (e.g., $x \cdot y \not\approx y \cdot x$). We overload notation and use $x; y$ and $x \cdot y$ to represent sequences both when x is an element and when x is a sequence, and similarly for y.

4.2 Operational Semantics of Statix with Hole State Tracking

The operational semantics in Fig. 8 also follows Rouvoet et al. [27]. The transition relation uses the configurations whose syntax is given in Fig. 7. A configuration $\left\langle \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$ comprises:

- $\mathcal{G}$: the currently constructed scope graph.
- $\overline{C}$: the current set of constraints.
- $U \in (\mathit{Var} \rightharpoonup \mathit{Hole})$: associates unification variables with holes. This lets us determine to which hole a given constraint might relate.
- $H \in (\mathit{Hole} \rightharpoonup (s^* \times t))$: maps each hole in the program to its state, consisting of a list of traversed scopes s^* and the term t constructed so far.

A main difference from Rouvoet et al. is that we extended the configuration to track the state of reference synthesis *holes* via the entities U and H. While the rules in Fig. 8 never access them, U and H are explicitly propagated by these rules such that substitutions resulting from unification get applied to them. We return to the role of U and H in §4.3.

$\mathbb{S} \vdash \kappa \to \kappa'$ — Configuration κ steps to κ' using specification $\mathbb{S}$

$$\textsc{Op-Conj}\quad \mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, (C_1 * C_2); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, C_1; C_2; \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$$

$$\textsc{Op-Emp}\quad \mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, \mathsf{emp}; \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$$

$$\textsc{Op-Exists}\ \frac{y \text{ is fresh}}{\mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, (\exists x.\, C); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, C[y/x]; \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle}$$

$$\textsc{Op-Eq}\ \frac{\mathsf{mgu}(t_1, t_2) = \theta}{\mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, (t_1 \stackrel{?}{=} t_2); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \theta}$$

$$\textsc{Op-Singleton}\quad \mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, \mathsf{single}(t, \{t'\}); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, (t \stackrel{?}{=} t'); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$$

$$\textsc{Op-Forall}\quad \mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, (\forall x \text{ in } \zeta.\, C); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, \{C[t/x] \mid t \in \zeta\}; \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$$

$$\textsc{Op-New-Scope}\ \frac{s \notin S}{\mathbb{S} \vdash \left\langle \langle S, R, \rho \rangle \,\middle|\, (\nabla x \mapsto t); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \langle s; S, R, \rho[s \mapsto t] \rangle \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle [s/x]}$$

$$\textsc{Op-New-Edge}\quad \mathbb{S} \vdash \left\langle \langle S, R, \rho \rangle \,\middle|\, (s_1 \xrightarrow{\ell} s_2); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \langle S, (s_1, \ell, s_2); E, \rho \rangle \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$$

$$\textsc{Op-Data}\ \frac{\rho_{\mathcal{G}}(s) = t_2}{\mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, \mathsf{dataOf}(s, t_1); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, (t_1 \stackrel{?}{=} t_2); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle}$$

$$\textsc{Op-Query}\ \frac{A = \mathsf{Ans}(\mathcal{G}, s \xrightarrow[o]{r}\!\!\!\!\gg \lambda x.\, E) \qquad \mathsf{guard}(\mathcal{G}, (s \xrightarrow[o]{r}\!\!\!\!\gg \lambda x.\, E \mapsto z.\, C), \overline{C})}{\mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, (s \xrightarrow[o]{r}\!\!\!\!\gg \lambda x.\, E \mapsto z.\, C); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, C[A/z]; \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle}$$

$$\textsc{Op-Pred}\ \frac{\exists!(P(\overline{t_2}) \leftarrow C) \in \mathbb{S}.\ \exists\theta.\ \mathsf{mgu}(\overline{t_1}, \overline{t_2}) = \theta}{\mathbb{S} \vdash \left\langle \mathcal{G} \,\middle|\, P(\overline{t_1}); \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \to \left\langle \mathcal{G} \,\middle|\, C; \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle \theta}$$

$\mathcal{G} \vdash E \rightsquigarrow \theta$ — Equality constraint E produces a unifier θ

$$\textsc{E-Eq}\ \frac{\mathsf{mgu}(t_1, t_2) = \theta}{\mathcal{G} \vdash t_1 \stackrel{?}{=} t_2 \rightsquigarrow \theta} \qquad \textsc{E-DataOf}\ \frac{\mathsf{mgu}(\rho_{\mathcal{G}}(s), t) = \theta}{\mathcal{G} \vdash \mathsf{dataOf}(s, t) \rightsquigarrow \theta}$$

$$\textsc{E-Conj}\ \frac{\mathcal{G} \vdash E_1 \rightsquigarrow \theta_1 \qquad \mathcal{G} \vdash E_2 \rightsquigarrow \theta_2}{\mathcal{G} \vdash E_1 * E_2 \rightsquigarrow \theta_1\theta_2} \qquad \textsc{E-Exists}\ \frac{y \text{ is fresh} \qquad \mathcal{G} \vdash E[y/x] \rightsquigarrow \theta}{\mathcal{G} \vdash \exists x.\, E \rightsquigarrow \theta}$$

Fig. 8. Operational semantics of constraints in Statix

Predicate Constraints. Rule Op-Pred defines the semantics of *predicate expansion*. To support this rule, the transition judgment in Fig. 8 is parameterized by a specification $\mathbb{S}$. This specification is given by a set of predicate rules where each rule has the shape $P(\bar{t}) \leftarrow C$. Each rule is closed (i.e., $FV(P(\bar{t}) \leftarrow C) = \emptyset$), and we assume that the domain of every predicate rule is disjoint from all other rules; i.e.:

$$\forall P\ \overline{t_1}\ \overline{t_2}\ C_1\ C_2.\ \left(P(\overline{t_1}) \leftarrow C_1\right) \in \mathbb{S} \wedge \left(P(\overline{t_2}) \leftarrow C_2\right) \in \mathbb{S} \wedge (\exists \theta.\, \mathsf{mgu}(\overline{t_1}, \overline{t_2}) = \theta) \Rightarrow \overline{t_1} \equiv \overline{t_2} \wedge C_1 \equiv C_2$$

We also assume that premises that access rules in a specification $\mathbb{S}$ (e.g., $(P(\bar{t}) \leftarrow C) \in \mathbb{S}$) are automatically α-renamed to avoid variable capture. Rule Op-Pred thus expands a predicate only when there exists a *unique* rule $P(\overline{t_2}) \leftarrow C$ in specification $\mathbb{S}$ whose head matches the predicate constraint $P(\overline{t_1})$. In case the predicate constraint matches multiple rules in $\mathbb{S}$, rule Op-Pred does not apply. For example, if P is a predicate symbol, f and g are constructors, x, y are variables, and we have a specification with rules $P(f(x)) \leftarrow C_1$ and $P(g(x)) \leftarrow C_2$, then the Op-Pred rule does not apply to the predicate constraint $P(y)$ because y can be instantiated to both $f(x)$ and $g(x)$.

The substitution yielded by mgu in the premise of Op-Pred is applied to the entire configuration after unfolding a predicate. Here and in the rest of the paper, mgu is the partial function computing the most general unifier (i.e., a substitution). We use $\bot$ to denote failure in partial functions. We use $\theta, \theta_1, \theta_{result}, \ldots$ to range over substitutions of variables by terms ($Var \rightharpoonup Term$) or set variables by set terms ($SetVar \rightharpoonup SetTerm$). The type of substitution will be clear from the context. The substitution functions for constraints, terms, and scope graphs are standard and elided for brevity, except for the reference entity U which we describe in §4.3.

Logic Constraints. The other rules in Fig. 8 are directly adapted from Rouvoet et al. [27]. Rule Op-Conj splits a separating conjunction constraint into two constraints. Op-Emp dispatches the vacuously satisfiable constraint emp. Op-Exists unpacks an existentially quantified constraint by choosing a fresh variable name, which may get unified using, for example, Op-Eq.

Set Constraints. Queries yield *sets* of results, so the rules in Fig. 8 include two dedicated constraints for matching on sets. The semantics of $\mathsf{single}(t_1, \underline{t_2})$ is given in rule Op-Singleton which asserts that t_2 must be a singleton set $\{t'\}$, such that t_1 unifies with t'. The semantics of constraint $\forall x \text{ in } \underline{t}.\, C$ is given in rule Op-Forall. The rule asserts that $\underline{t}$ must be some set literal ζ; i.e., a union of singleton sets. The rule expands $\forall x \text{ in } \zeta.\, C$ into as many constraints as ζ has singleton sets, in each case substituting x for the singleton set inhabitant.

Scope Graph Constraints. The rules Op-New-Scope and Op-New-Edge create new scopes and edges in the scope graph, respectively. Rule Op-Data asserts that a constraint $\mathsf{dataOf}(t_1, t_2)$ can be solved when t_1 is a scope s, and t_2 unifies with the term associated with scope s. Rule Op-Query has two premises. The function Ans returns the set of all paths that match the query parameters (see §2.1). The guard predicate ensures that the query is *guarded* in the sense that the constraints $C; \overline{C}$ do not add a new edge to the scope graph that would cause the query to yield a different answer. Both Ans and guard are discussed in detail by Rouvoet et al. [27, §3.1 and §5.3].

4.3 Operational Semantics of Reference Synthesis

The usual operational semantics of Statix in Fig. 8 is conservative about solving query and predicate constraints. As discussed before, Op-Pred solves a predicate constraint only when there is exactly one possible expansion, otherwise it is *stuck*. Similarly, Op-Query only solves a query constraint when the source scope of the query is ground (i.e., it is a scope rather than a variable), and the guard premise holds.

As we can observe from the example discussed in §3, speculatively solving predicate and query constraints allows the Statix solver to *infer* what syntax of valid references to substitute for each hole of a program. To admit such inference, the rules in Fig. 9 let us *speculatively* expand *predicate* and *query constraints* in states that would otherwise be stuck. We achieve this by introducing two new relations: $\rightarrowtail$ performs a speculative expansion step, and $\rightarrowtail\!\!\!\!\!\rightarrow$ relates sets of potentially speculatively expanded configurations.

The $\rightarrowtail\!\!\!\!\!\rightarrow$ Relation. As long as the regular Statix constraint solving rules can make progress, the Op-Solve rule applies. Once the solver gets stuck, Op-Expand applies. The set comprehension in the bottom premise of the rule lets us speculatively solve stuck query constraints and expand predicates. Configurations for which neither Op-Solve nor Op-Expand apply are truly stuck, and will be pruned by the set comprehension premise of Op-Expand.

Speculative Predicate Expansion. The rule Op-Expand-Pred augments the plain Statix constraint solving rules from Fig. 8 to support selecting an *arbitrary* rule for expanding a predicate constraint.

Speculative Query Expansion. The rule Op-Expand-Query augments the plain Statix constraint solving rules from Fig. 8 with support for solving a stuck query constraint, by synthesizing a path from a (possibly unknown) source scope to the current target scope of the related hole. The rule assumes that we are resolving a reference given by a composite path, and attempts to "prepend" a step to the composite path. Intuitively, if we think of composite paths as (qualified) references, this corresponds to attempting to prepend a qualifier.

Op-Expand-Query uses the U and H components of the state, to determine which hole it is expanding. For each free variable in the program U tracks to which hole it is related. For this reason, its substitution function $U[t/x]$ has a guard that checks that each free variable in t are either not related to a hole, or are related to the same hole as x. As shown previously in Fig. 7, the state of a hole $H(h)$ is given by a pair $(\overline{s}, t)$. Here, t is a term representing the inferred syntax for the reference, while $\overline{s}$ represents a non-empty sequence of *query-connected scopes* that form its composite path.

Definition 4.1 (Query-Connected Scopes). For a given scope graph $\mathcal{G}$, two scopes s_1 and s_2 in $\mathcal{G}$ are *query-connected* by a query $q = s_1 \xrightarrow[o]{r}\!\!\!\!\!\rightarrow \lambda x.\, E$, which we denote $s_1 \xrightarrow{q} s_2$, when there exists an p such that $p \in \mathsf{Ans}(\mathcal{G}, s_1 \xrightarrow[o]{r}\!\!\!\!\!\rightarrow \lambda x.\, E)$ where either $s_2 = \mathsf{tgt}(p)$ or $s_2 \in \rho_{\mathcal{G}}(\mathsf{tgt}(p))$.

This uses the notation $s_2 \in \rho_{\mathcal{G}}(\mathsf{tgt}(p))$ to mean that s_2 is a syntactic sub-term of the data associated with $\mathsf{tgt}(p)$. Using the definition we can define composite paths, which model references.

Definition 4.2 (Composite Path). For a given scope graph $\mathcal{G}$, a sequence of scopes $s_0 \ldots s_n$, and a sequence of queries $q_i \in \overline{q}$, a composite path in the scope graph is given by a series of query-connected scopes; i.e.: $s_0 \xrightarrow{q_1} s_1 \cdots \xrightarrow{q_n} s_n$

The first premise of rule Op-Expand-Query asserts that the data well-formedness predicate $(\lambda y.\, E)$ has a variable occurring in it which is relevant for a hole h. The head scope s_t of the composite path component in the state of h represents the scope that we need to connect to, in order to prepend a step to a query-connected path. The remaining premises assert that we choose a source scope s' and a target scope s'' that is connected to s_t, such the query resolves from s' to s''.

Accepting States. The $\mathsf{Accept}(\overline{C}, H)$ premise of Op-Expand holds iff (1) $\overline{C}$ is empty (all constraints are solved), and (2) the state of each hole in H has a composite path component of length > 1 (we have constructed a composite path for each hole).

$\mathbb{S} \vdash \overline{\kappa} \rightarrowtail\!\!\!\!\rightarrow \overline{\kappa'}$ — Configurations $\overline{\kappa}$ step to $\overline{\kappa'}$ using specification $\mathbb{S}$

$$\textsc{Op-Solve}\ \frac{\mathbb{S} \vdash \kappa \rightarrow \kappa'}{\mathbb{S} \vdash \kappa; \overline{\kappa} \rightarrowtail\!\!\!\!\rightarrow \kappa'; \overline{\kappa}} \qquad \textsc{Op-Expand}\ \frac{\begin{array}{c}\mathbb{S} \vdash \left\langle\, \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \,\right\rangle \not\rightarrow \qquad \neg\mathsf{Accept}(\overline{C}, H) \\ \overline{\kappa'} = \left\{ \left\langle\, \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H' \,\right\rangle \theta \;\middle|\; \mathbb{S} \vdash \left\langle\, \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \,\right\rangle \rightarrowtail \theta, H' \right\}\end{array}}{\mathbb{S} \vdash \left\langle\, \mathcal{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \,\right\rangle; \overline{\kappa} \rightarrowtail\!\!\!\!\rightarrow \overline{\kappa'}; \overline{\kappa}}$$

$\mathbb{S} \vdash \kappa \rightarrowtail \theta, H$ — For κ, synthesize substitution θ and hole state H, using specification $\mathbb{S}$

$$\textsc{Op-Expand-Pred}\ \frac{(P(\overline{t_2}) \leftarrow C) \in \mathbb{S} \qquad \mathsf{mgu}(\overline{t_1}, \overline{t_2}) = \theta}{\mathbb{S} \vdash \left\langle\, \mathcal{G} \,\middle|\, P(\overline{t_1}); \overline{C} \,\middle|\, U \,\middle|\, H \,\right\rangle \rightarrowtail \theta, H}$$

$$\textsc{Op-Expand-Query}\ \frac{\begin{array}{c} h \in \{h \mid x \in FV(\lambda y.\, E), h = U(x)\} \qquad H(h) = \left(s_t \cdot \overline{s'_t}, t'\right) \\ s', s'' \in S_{\mathcal{G}} \qquad \mathsf{mgu}(t, s') = \theta_1 \qquad (s_t = s'' \vee s_t \in \rho_{\mathcal{G}}(s'')) \qquad \mathcal{G} \vdash E[s''/y]\theta_1 \rightsquigarrow \theta_2 \\ \exists p \in \mathsf{Ans}(\mathcal{G}\theta_1\theta_2, s' \xrightarrow{r}_{o}\!\!\!\!\rightarrow \lambda y.\, E\theta_1\theta_2).\, \mathsf{tgt}(p) = s'' \qquad \mathsf{guard}(\mathcal{G}\theta_1\theta_2, (s' \xrightarrow{r}_{o}\!\!\!\!\rightarrow \lambda x.\, E \mapsto z.\, C)\theta_1\theta_2, \overline{C}) \end{array}}{\mathbb{S} \vdash \left\langle\, \mathcal{G} \,\middle|\, t \xrightarrow{r}_{o}\!\!\!\!\rightarrow \lambda y.\, E \mapsto z.\, C; \overline{C} \,\middle|\, U \,\middle|\, H \,\right\rangle \rightarrowtail \theta_1\theta_2, H\left[h \mapsto \left(s' \cdot s_t \cdot \overline{s'_t}, t'\right)\right]}$$

Auxiliary functions

$$U[t/x] = \begin{cases} U & \text{if } x \notin \mathsf{dom}(U) \\ U[\{y \mapsto U(x) \mid y \in FV(t)\}] & \text{if } x \in \mathsf{dom}(U) \text{ and } \forall y \in FV(t), h'.\, U(y) = h' \Rightarrow h' = U(x) \\ \bot & \text{otherwise} \end{cases}$$

Fig. 9. Operational semantics of reference synthesis.

4.4 Building the Synthesize Function

Now, we have all the pieces to build the synthesize function:

$$\frac{\begin{array}{c} D = \{x \mapsto d \mid (\!|\rightarrow d|\!) \in \mathcal{P}_{\text{locked}}, x \text{ fresh}\} \qquad U = \{x \mapsto h \mid x \in \mathsf{dom}(D), h \text{ fresh}\} \\ \theta_1 = \{(\!|\rightarrow d|\!) \mapsto x \mid D(x) = d\} \qquad \mathcal{P}_0 = \mathcal{P}_{\text{locked}}\theta_1 \qquad \left\langle\, \emptyset \,\middle|\, P_0(\mathcal{P}_0) \,\middle|\, U \,\middle|\, \emptyset \,\right\rangle \rightarrow^* \left\langle\, \mathcal{G}_0 \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, \emptyset \,\right\rangle \\ H = \left\{h \mapsto (s_d, x) \mid U(x) = h, D(x) \in \rho_{\mathcal{G}_0}(s_d)\right\} \qquad \left\langle\, \mathcal{G}_0 \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \,\right\rangle \rightarrowtail\!\!\!\!\rightarrow^* \overline{\kappa} \\ \left\langle\, \mathcal{G} \,\middle|\, \overline{C'} \,\middle|\, U' \,\middle|\, H' \,\right\rangle \in \overline{\kappa} \qquad \mathsf{Accept}(\overline{C'}, H') \qquad \theta_{result} = \{x \mapsto t \mid H'(U(x)) = (s_t, t)\} \end{array}}{\mathsf{synthesize}(\mathcal{P}_{\text{locked}}) = \mathcal{P}_0\theta_{result}}$$

We first create a fresh unification variable and hole for each locked reference in the program (D and U, respectively), and replace each locked reference by a fresh unification variable in the program (θ_1). Then, we solve the initial constraint (P_0) on the program with holes. From the partial scope graph in the result state, we initialize the hole state H for each hole. Then, we synthesize the references, and extract an accepted state. From this state, we build a substitution θ_{result} that substitutes each hole variable with the synthesized reference term.

4.5 Properties

In this section, we discuss some properties of our operational semantics: soundness, completeness, and confluence; and we discuss liveness.

Soundness. First, we consider the *soundness* of our approach. Soundness consists of two components: (1) the resulting solutions are well-typed; and (2) each solution corresponds to a composite path in the scope graph to the target declaration.

Theorem 4.3 (Soundness 1). *Programs with synthesized references are well-typed.*

Theorem 4.4 (Soundness 2). *Every synthesized reference corresponds to a composite path (definition 4.2) to the target it was initially locked to.*

Formal definitions and proofs of theorems 4.3 and 4.4 can be found in the extended version of this paper [24], appendices A.2 and A.3, respectively.

Completeness. Ideally, we would conjecture *completeness* as well. However, completeness is hard to define, as it must rely on a generic (language-independent) definition of a reference. For the purpose of this paper, we consider a well-formed reference in scope s_0 to declaration d to be a composite path (definition 4.2) through scope graph $\mathcal{G}$, from s_0 to s_d where $d \in \rho_{\mathcal{G}}(s_d)$ (i.e., s_d is associated with declaration d). However, this definition is an over-approximation, as queries could be connected 'by accident'. For example, a Java expression `a.m(b)` where `a` is an instance of the class in which this expression occurs could be considered a reference `a.b` by our definition, as the target of the query for `a` would match the source of the query of `b`. Thus, our definition is not suitable to state a completeness theorem. In §6.3 we show experimental evidence that our approach is practically complete.

Confluence. We also consider the property of *confluence*: if two different expansions are possible, eventually, the final state sequence will be equivalent.

Theorem 4.5 (Confluence). *If* $\overline{\kappa} \rightarrowtail\!\!\!\!\rightarrow \overline{\kappa}_1$ *and* $\overline{\kappa} \rightarrowtail\!\!\!\!\rightarrow \overline{\kappa}_2$, *then* $\exists \overline{\kappa'}.\ \overline{\kappa}_1 \rightarrowtail\!\!\!\!\rightarrow^* \overline{\kappa'} \wedge \overline{\kappa}_2 \rightarrowtail\!\!\!\!\rightarrow^* \overline{\kappa'}$

Proof. This is a proof by case analysis on the expanded state:

- If different states were expanded in $\overline{\kappa}_1$ and $\overline{\kappa}_2$, the step made to obtain $\overline{\kappa}_1$ can be applied on $\overline{\kappa}_2$ as well, and vice versa. This yields equivalent states again.
- If the same state was expanded in $\overline{\kappa}_1$ and $\overline{\kappa}_2$, either one of the following is true:
  - An Op-Solve-step was made in both cases. In this case, confluence holds by virtue of $\longrightarrow$ being confluent [27, Theorem 4.5].
  - An Op-Expand-step was made in both cases. As this rule ranges over all possible expansions, $\overline{\kappa}_1 = \overline{\kappa}_2$, so confluence trivially holds.

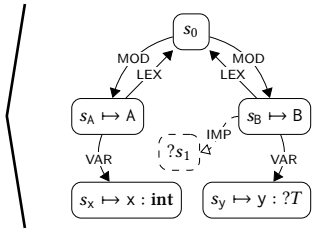
  The same state cannot be expanded with both Op-Solve and Op-Expand, as the first premise of Op-Expand requires the state to be stuck (i.e., Op-Solve does not apply). □

This confluence result is especially important for our next section, as we exploit this property to design heuristics that reduce the huge search space of possible expansions.

Liveness. Finally, albeit not a property of the operational semantics, we discuss *liveness* here as well. Liveness consists of two components: (1) the synthesis finds each solution in finite time, and (2) when no (new) solution is available, the synthesis terminates. Property (1) can be guaranteed by scheduling the expansion steps 'fairly'; i.e. in a breadth-first manner. Property (2) requires special care for predicates that are (potentially) infinitely expanding without yielding a solution. We return to this in §5.5.

```
1 mod A {
2   var x = 1
3 }
4 mod B {
5   import (⟶A) : : *
6   var y = (⟶x)
7 }
```

$$H = \begin{cases} ?A_1 & \mapsto h_A \\ ?x & \mapsto h_x \\ ?A_2 & \mapsto h_x \end{cases} \qquad U = \begin{cases} h_A & \mapsto (s_\mathsf{A}, ?A_1) \\ h_x & \mapsto (s_\mathsf{x}, ?A_2.?x) \end{cases}$$



$$\begin{array}{ll} \mathsf{scopeOfMod}(s_\mathsf{B}, ?A_1, ?s_1) & (1) \\ s_\mathsf{B} \xrightarrow{\mathsf{IMP}} ?s_1 & (2) \\ \mathsf{scopeOfMod}(s_\mathsf{B}, ?A_2, ?s_2) & (3) \\ ?s_2 \xrightarrow{\mathsf{VAR}} \!\!\!\rightarrow \mathsf{isVar}_{?x} \mapsto \{(\_, ?x : ?T)\} & (4) \end{array} \quad \Big| \; U \; \Big| \; H \; \Big\rangle$$

Fig. 10. Example program (top-left) and intermediate synthesis state (below and top-right).

5 Heuristics

The operational semantics is highly non-deterministic. A direct, naive implementation would perform duplicate and unnecessary work. To reduce work, our implementation of reference synthesis uses heuristics to guide the search. These heuristics have three goals: (i) to guide the search toward results, (ii) to avoid duplicate work, and (iii) to cut search branches early that do not lead to results. For all these heuristics, we argue that they do not yield solutions that are not derivable from the operational semantics (soundness), and preserve all derivable solutions as well (completeness).

In this section, we discuss these heuristics, using the example in Fig. 10. Performing reference synthesis on the program on the top-left eventually reaches the state shown in the figure. The first two constraints are obtained from expanding rule D-ImportOk on the initial hole on line 5 (h_A), represented by the variable $?A_1$. Resolving this reference yields a scope, currently represented by the variable $?s_1$. Once this scope is resolved, an incoming edge from s_B can be created. Until then, this edge is not present in the scope graph (hence it is indicated with a dashed line). The other two constraints correspond to the hole on line 6 (h_x), which is expanded to a reference with a single qualifier $?A_2.?x$. The qualifier should resolve to a scope $?s_2$, in which afterward, a query resolving to the target scope s_x will be performed.

5.1 Selecting Constraints

As discussed in §4.5, our system is *confluent*. For that reason, we can choose one single order in which to expand constraints, instead of trying all possible orders. We choose the constraint to expand according to the following criteria (considered in order): (i) prefer queries over predicate constraints; (ii) prefer predicate constraints that can lead to queries over those that cannot; and (iii) prefer older constraints over newer constraints. The rationale behind this order is that we try to expand queries as soon as possible. Expanding queries typically blows up the search space less than expanding predicates, and queries often reach terminal states (ground references). Using age as a tiebreaker, we try to explore the remaining state space in a breath-first manner, to ensure we reach all possible references. In our example, this implies that we first expand the query constraint (4), inferring x to be the value of $?x$, and s_A to be the target $?s_2$ of constraint (3). As we eventually expand all constraints, we preserve soundness and completeness.

5.2 Expanding Queries

When expanding a query, a source and target scope must be chosen (s' and s'' in Op-Expand-Query in Fig. 9). Instead of trying the query with all possible combinations of source and target scopes, we can be smart about the source and target scopes we pick. First, we choose s'', based on the premise that it should be equal to the current target scope, s_t, or contain it. Then, we infer a unifier from the query, $\mathcal{G} \vdash E[s''/y] \rightsquigarrow \theta$. If no such unifier exists, we stop the search for this branch. Finally, we traverse the scope graph backwards, starting at s''. We use the inverted regular expression (e.g., $\mathsf{inv}(\mathsf{LEX}^*\,\mathsf{VAR}) = \mathsf{VAR}\,\mathsf{LEX}^*$) to guide the graph traversal to only find source scopes that have a valid path to the target scope. This approach only over-approximates when there is another declaration, reachable from s', that shadows s''. It is sound, as eventually we check all the premises of the rule. It is also complete, as all other choices for s'' do not satisfy the premise that relates it to s_t, and all other choices for s' do not yield valid query answers.

In our example, when expanding the query constraint (4), the only reasonable choice for s'' is s_{x}. Then, solving E yields $\{?x \mapsto \mathsf{x}\}$. Next, following the inverted regular expression (VAR) traversing the edge $s_{\mathsf{A}} \xrightarrow{\mathsf{VAR}} s_{\mathsf{x}}$ backwards, we choose s' to be s_{A}. This instantiation is a valid query expansion.

5.3 Expanding Predicates

When selecting a predicate (§5.1) we fork the computation and try each possible expansion of a predicate to a rule concurrently. We prioritize expansions that lead to a query, as those might converge to a result quicker. Additionally, we prioritize rules that have fewer free variables, as those add less freedom to the problem; i.e., tell us more about the final solution. As we only prioritize certain rules over others, but never discard any, we will eventually try all rules. Therefore, this does not affect completeness. In our example, this implies that we prioritize expanding the $\mathsf{scopeOfMod}$ constraints using rule S-Mod before rule S-QMod.

5.4 Isolating Holes

The schemes in §5.1 and §5.3 reduce the search space significantly, but may skew the search to holes with less complicated solutions. Therefore, when starting the computation, we fork the state for each hole, which we call the *focus hole* of that search branch. We only expand constraints that are related to this focus hole, ensuring we make progress on this hole specifically.

However, this approach is incomplete, as sometimes the solution of one hole can only be computed after another hole is solved. For example, for the program in Fig. 10, x is a valid solution for h_x (although in a different branch than the state presented there). However, this can only be computed when the solution for h_A is known, as it requires the edge $s_{\mathsf{B}} \xrightarrow{\mathsf{IMP}} s_{\mathsf{A}}$ to be present in the scope graph. We need a way to ensure such 'composite' solutions are computed as well.

We ensure this as follows. Suppose query expansion (§5.2) traverses an l-labeled edge. When there is a scope s such that future steps *might* create an l-labeled edge in s (i.e., $\overline{C} \hookrightarrow (s, l)$; see Rouvoet et al. [27, §5.3]), we might miss edges. In this case, we find the constraint that is responsible for the missing edge, and all constraints that (transitively) share a variable with this constraint. Some of these constraints may correspond to a different hole. Then, we peek at that hole for solutions and try insert those solutions in our current state. Solving this state should create the missing edge. Next, we can resume our backwards traversal. Since we over-approximate the missing edges ($\hookrightarrow$ over-approximates, and the future edges may also have different target scopes), we preserve completeness. In addition, we do not change the traversal itself, so we also preserve soundness.

In the example, we detect that constraint (2) is responsible for the missing edge. As this constraint shares a unification variable with constraint (1), we detect it is related to h_A. Thus, we insert a solution for h_A (e.g., A) in the state, after which we can find the solution x for h_x.

5.5 Recursive Qualifiers

Another heuristic concerns *recursive qualifiers*. We consider a reference recursive if multiple qualifiers in the reference resolve to the same declaration. For example, consider a Java class `A` with field `int x` and a field `A a`. In this case, a hole referring to `x` has an infinite number of possible solutions: `x`, `a.x`, `a.a.x`, etc. In these cases, each `a` refers to the same declaration.

To explain how we optimize synthesizing these references, we consider two of the states the synthesis of `a.a.x` goes through. After some steps, the synthesis reaches the following state:

$$\kappa_1 = \left\langle \mathcal{G} \,\middle|\, \mathsf{resolveQVar}(s_\mathsf{A}, ?q_1, ?s_1) \,\middle|\, \{?q_1 \mapsto h, \ldots\} \,\middle|\, \{h \mapsto (s_\mathsf{A} \cdot s_\mathsf{x}, ?q_1.\mathsf{x}), \ldots\} \right\rangle$$

Some forks of this branch will arrive at `a.x` as a solution for this hole. However, on the path to synthesizing `a.a.x`, the following state will also be reached:

$$\kappa_2 = \left\langle \mathcal{G} \,\middle|\, \mathsf{resolveQVar}(s_\mathsf{A}, ?q_2, ?s_2) \,\middle|\, \{?q_2 \mapsto h, \ldots\} \,\middle|\, \{h \mapsto (s_\mathsf{A} \cdot s_\mathsf{A} \cdot s_\mathsf{x}, ?q_2.\mathsf{a.x}), \ldots\} \right\rangle$$

These states are very similar: they share the same scope graph, have α-equivalent constraints, and the same target scope in the hole state of h. For that reason, we can conclude that *each solution* for $?q_1$ is *also a solution* for $?q_2$, as any derivation starting in κ_1, is also valid in κ_2. Therefore, we can avoid synthesizing the same solution multiple times by reusing solutions for $?q_1$ when synthesizing $?q_2$. As the traces are equivalent, we preserve soundness and completeness.

Algorithmically, we achieve this in the following three steps: (1) We detect pairs of solver states where (a) the constraints are equivalent up to α-renaming, (b) the hole's term in the recursive solver state is an instantiation of the term in the base state, and (c) the hole has the same target scope. For these states, we stop further synthesis on the recursive state. (2) For each ground solution for the variable in the base state, we emit a solution derived from the recursive state. (3) Repeat from step 2, using this new solution.

In the example above, this detects that κ_2 is a recursive state with respect to κ_1. Thus, solutions from other branches rooted in κ_1 (such as `a.x`) are reused in the variable $?q_2$ in κ_2, yielding `a.a.x`.

This expansion is interleaved with executing other search branches, in order to guarantee liveness. In the case no base solutions exist, this immediately terminates a search branch that would otherwise run infinitely without returning results. This ensures that we are terminating on recursive instances that match this pattern. When we assume the predicates that model references do not generate new scopes, only a finite number of recursive instances can be generated (because there exist a finite number of scopes in the scope graph, and a finite number of rules in a specification; hence only a finite number of states that are not equivalent according to the definition in step 1 above exist.). That implies that we can only have non-termination when the recursive reference predicate generates fresh scopes, which typical specifications do not do.

The next section evaluates our implementation, which is based on these heuristics, providing evidence that we indeed preserve soundness and completeness.

6 Experimental Evaluation

In evaluating reference synthesis, we focus on three key criteria: (1) ensuring that synthesized references are valid according to the static semantic specification of the language and resolve to the intended declaration (*soundness*), (2) verify the ability to find a valid reference if it exists (*completeness*), and (3) assess the efficiency of the synthesis process (*performance*). In this section we discuss how we evaluated these aspects of reference synthesis.

	Java	ChocoPy	FGJ
Included	2528 (62%)	196 (64%)	49 (94%)
Negative	0 (0%)	55 (18%)	0 (0%)
Java 8 incompat.	1076 (26%)	N/A	N/A
Spec incompat.	197 (5%)	10 (3%)	0 (0%)
No references	304 (7%)	47 (15%)	3 (6%)
Total	4105 (100%)	308 (100%)	52 (100%)

Fig. 11. Test selection.

	Java	ChocoPy	FGJ
Success	2382 (94%)	155 (79%)	38 (78%)
Timeout	146 (6%)	41 (21%)	11 (22%)
Failure	0 (0%)	0 (0%)	0 (0%)
Total	2528 (100%)	196 (100%)	49 (100%)

Fig. 12. Test outcomes.

6.1 Languages

We evaluated our synthesize function on three larger programming languages: Java, ChocoPy, and Featherweight Generic Java (FGJ).

(1) *Java* is a mainstream language with sophisticated name binding features. We evaluated our approach using an existing Statix specification of Java 8 from the artifact [37] associated with the work of van Antwerpen and Visser [38]. This specification of Java unfortunately does not support generics or method references. We derived test cases from Java files used to validate the implementation of refactorings in the JetBrains IntelliJ IDE.[7]
(2) *ChocoPy* [20] is a statically-typed dialect of Python. We used an existing Statix specification and ChocoPy files for our evaluation.
(3) *Featherweight Generic Java* [10] (FGJ) is a functional Java core language with full generics support. We used the Statix specification from the artifact [35] of van Antwerpen et al. [36].

6.2 Method

For each language we used the existing Statix semantic specifications *without modification*, as our synthesize function works directly with these specifications. We collected a set of test cases: single-file programs where we locked variable names, type names, and qualified member names that occur in it. As a result, many test cases contain multiple locked references. Each locked reference only has the target declaration as a parameter. The original syntax of the reference is erased. We excluded negative test cases (tests that validated that the specification or implementation gives an error on incorrect programs), those that are incompatible with our Statix specification or Java 8 (in the case of Java tests), and those that had no references to lock. The resulting selection is shown in Fig. 11. Our reference synthesis algorithm was then applied to propose concrete references until the original reference in the program was recovered. We set a timeout of 60 seconds per test case and ran the evaluation on a MacBook Pro 2019 with a 2.4 GHz Intel Core i9 processor and 64 GB memory.

6.3 Results

A summary of the results of the evaluation is shown in Fig. 12. The results confirm the *soundness* of our reference synthesis algorithm. Substituting the synthesized references back into the original programs only produced well-typed programs (theorem 4.3).

Our evaluation demonstrates a high level of *completeness*, as it successfully synthesized each reference encountered in our test cases, including non-trivial references such as Java's `A.super.x` and references with ambiguous qualifiers [8, §6.5.2]. Nevertheless, formally proving *completeness* of the algorithm is challenging (§4.5).

[7] https://github.com/JetBrains/intellij-community/tree/idea/233.14808.21/java/java-tests/testData/refactoring

While *performance* was not the main focus of our approach, we also measured the time taken to synthesize each reference, shown as a violin plot in Fig. 13. The plot illustrates the distribution of the synthesis time per hole as a density curve, where for all three languages most results lie between 10 and 200 milliseconds. A small box plot is depicted inside each density curve, highlighting the median, range of the central 50% results, and range of the remaining data. For the majority of locked references the algorithm proposed a solution within one second. In around 7% of the test cases, reference synthesis failed to find a solution for all locked references within the 60-second timeout. This typically occurred when references were strongly interdependent, as reference synthesis will exhaustively explore all combinations of solutions for the dependent and the dependency (see §5.4).

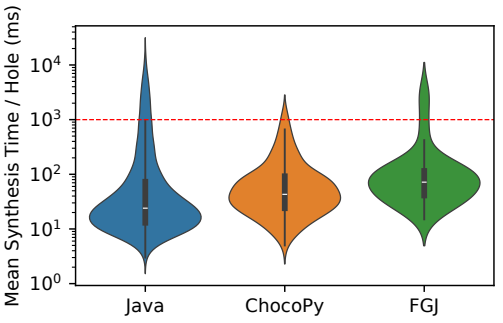


Fig. 13. Logarithmic plot of the time spent synthesizing references per hole in each of 2575 successful test cases. The dashed line marks 1 second.

6.4 Threats to Validity

To mitigate bias in the references we lock, or the alternatives we try, we make two conservative assumptions that are not generally true for a refactoring: (1) *all* references in the program must be locked; and (2) the original reference syntax is unknown. These assumptions make our test cases more challenging and may result in a worse performance than it would be in practical scenarios.

Instead of locking all references in the program, a typical refactoring would only need to lock a subset of those references, namely those that could get changed due to the program transformation. Consequently, the likelihood of interdependent locked references would be lower. Furthermore, most existing references are likely to remain valid despite the transformation. Therefore, reference synthesis could prioritize verifying that the existing reference syntax still resolves to the intended declaration before synthesizing a new concrete reference. As a result, only a handful of references need to be synthesized.

Another potential threat to validity is that our approach is parameterized by the language's static semantics written in Statix. As shown by Rouvoet et al. [27], van Antwerpen et al. [36], Zwaan and Poulsen [41], Statix can express many interesting name binding and type system concepts, but it is yet unclear what language concepts Statix cannot express. The features of Statix that we heavily rely on are the solver interface, the presence of predicate constraints and query constraints, and conservative scheduling. Given that we do not expect significant changes in any of these, we expect that our reference synthesis algorithm will work without fundamental modifications being required by possible future extensions to Statix.

7 Related Work

Refactoring as a discipline and subject goes back to the pioneering works by Griswold [9] and Opdyke [19]. Since then, refactoring has become a well-established field that has grown exponentially. Steimann [32] notes that the growth has been so large that a recent attempt to update the Mens and Tourwè's survey [16] was abandoned as there were too many works to be considered. Consequently, we focus our discussion on prior work most relevant to reference synthesis.

A distinguished line of work on implementing refactorings is Thompson et al.'s work on refactoring tools for Haskell [12, 14] and Erlang [13], which provides support for scripting a general-purpose code transformations with possibilities to specify pre- and post-conditions that ensure the transformation preserves certain properties explicitly. This support is realized via name-binding APIs implemented from scratch for each individual language.

As discussed in the introduction, our work builds upon previous work by Ekman et al. [5], Schäfer and de Moor [29], Schäfer et al. [30, 31], who introduced the idea of tracking name-binding dependencies by "locking" references to declarations and then synthesizing concrete references. Schäfer and de Moor applied this concept to synthesizing references for Java but deemed their reference synthesis algorithm to be beyond the scope of their paper [29, §2]. Our reference synthesis algorithm is applicable to any language whose typing rules are defined using Statix [36].

In other closely related previous work, de Jonge and Visser [3] describe a language-generic API for name-binding preserving refactorings. Their approach is inspired by the work of Ekman et al. [5] on JastAdd [4], which they generalize by giving operations for querying name-binding information and requalifying names. They demonstrate that this approach could be applied to multiple languages, including Stratego and a subset of Java [39, 40]. The main difference between their generic API and our work is that de Jonge and Visser require the requalification function to be explicitly provided. Our work provides this function.

Tip et al. [33] demonstrate that the problem of checking that a refactoring preserves well-typedness in a language like Java can be expressed as a *type constraint problem* [21]. Their type constraint framework supports a wide range set of refactorings for a large subset of Java. While their work focuses on Java-specific constraints, we address the broader problem of guaranteeing that references resolve to the same declaration for *any* programming language whose semantics of name resolution is defined using scope graphs, with Java serving as one of the case studies.

Steimann [32] generalizes the work of Tip et al. to also consider binding preservation and provides a more general and language-independent foundation for constraint-based refactoring. Although this foundation could in principle support refactorings in terms of name-binding constraints that would guarantee binding preservation in any language, the question of how to provide a language-parametric semantics for such constraints is left open. Our language-parametric algorithm might provide an answer to this question.

An important aspect of refactorings that move code is to avoid accidental name capture. A common approach to avoiding name capture is *renaming* (following, e.g., the Barendregt convention [2]). While our reference synthesis algorithm currently focusses on synthesizing (qualified) references, it does not produce suggestions where name capture could be avoided by renaming declarations. The language-parametric Name-Fix algorithm due to Erdweg et al. [6] provides an interesting solution to this problem.

Despite serving a very different purpose, Pelsmaeker et al. [22] define a language-parametric code completion algorithm that also relies on the Statix specification. Like our approach, they insert a free unification variable in a placeholder position, and partially type-check the program. By inspecting the stuck constraints, they find and propose type-sound suggestions for code completion. Their work, together with ours, shows that having a declarative but executable specification of the static semantics is essential to deriving sound, language-parametric editor services.

8 Conclusion

We have presented a novel approach reference synthesis that is *automatic*, *language-parametric*, and *sound*—generating only well-typed references to the intended declarations. This approach is applicable to any language whose static semantics is defined using typing rules in Statix [36]. It works out-of-the-box for such languages, is *sound* by construction, and uses non-deterministic search to provide a high degree of *completeness*. Our evaluation demonstrates that our algorithm works on practical examples, but also reveals that our generic approach comes with a high performance cost, which we intend to explore in future work.

9 Data Availability

This paper is accompanied by an artifact [23], a Docker container that includes our reference synthesis tool, source code, and its dependencies. Our tool can be executed in *evaluation* mode to reproduce the data presented in §6. The evaluation utilizes a set of test programs with locked references, that are all included in the artifact. Additionally, the original test sets from which the test cases were derived are also provided for further reference.

Acknowledgments

We would like to thank Luka Miljak and the anonymous reviewers for their comments and feedback on previous versions of this paper. This work is supported by the Programming and Validating Software Restructurings project (17933, NWO-TTW, MasCot).

References

[1] Alfred V. Aho and Jeffrey D. Ullman. 1972. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[2] Hendrik Pieter Barendregt. 1984. *The Lambda Calculus - Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland.

[3] Maartje de Jonge and Eelco Visser. 2012. A language generic solution for name binding preservation in refactorings. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, Anthony Sloane and Suzana Andova (Eds.). ACM, 2. doi:10.1145/2427048.2427050

[4] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 1–18. doi:10.1145/1297027.1297029

[5] Torbjörn Ekman, Max Schäfer, and Mathieu Verbaere. 2008. Refactoring is not (yet) about transformation. In *Second ACM Workshop on Refactoring Tools, WRT 2008, in conjunction with OOPSLA 2008, Nashville, TN, USA, October 19, 2008*. ACM, 5. doi:10.1145/1636642.1636647

[6] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. 2014. Capture-Avoiding and Hygienic Program Transformations. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard Jones (Ed.). Springer, 489–514. doi:10.1007/978-3-662-44202-9_20

[7] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley. http://martinfowler.com/books/refactoring.html

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java Language Specification - Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/html/

[9] William G. Griswold. 1992. *Program Restructuring As an Aid to Software Maintenance*. Ph. D. Dissertation. Seattle, WA, USA.

[10] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. doi:10.1145/503502.503505

[11] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 237–238. doi:10.1145/1869542.1869592

[12] Huiqing Li, Claus Reinke, and Simon J. Thompson. 2003. Tool support for refactoring functional programs. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. ACM, 27–38. doi:10.1145/871895.871899

[13] Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. 2006. Refactoring Erlang Programs. In *The Proceedings of 12th International Erlang/OTP User Conference*. Stockholm, Sweden. http://www.cs.kent.ac.uk/pubs/2006/2455

[14] Huiqing Li, Simon Thompson, and Claus Reinke. 2005. The Haskell Refactorer, HaRe, and its API. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 29–34. doi:10.1016/j.entcs.2005.02.053

[15] Huiqing Li and Simon J. Thompson. 2012. Let's make refactoring tools user-extensible!. In *Fifth Workshop on Refactoring Tools 2012, WRT '12, Rapperswil, Switzerland, June 1, 2012*, Peter Sommerlad (Ed.). ACM, 32–39. doi:10.1145/2328876.2328881

[16] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Trans. Software Eng.* 30, 2 (2004), 126–139. http://csdl.computer.org/comp/trans/ts/2004/02/e0126abs.htm

[17] 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. doi:10.1007/978-3-662-46669-8_9

[18] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. http://dl.acm.org/citation.cfm?id=3009900

[19] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph. D. Dissertation. University of Illinois, Urbana-Champaign, IL, USA. Advisor(s) Ralph E. Johnson.

[20] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3358711.3361627

[21] Jens Palsberg and Michael I. Schwartzbach. 1994. *Object-oriented type systems*. Wiley.

[22] 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. doi:10.1145/3527329

[23] Daniel A. A. Pelsmaeker, Aron Zwaan, Casper Bach, and Arjan J. Mooij. 2025. *Language-Parametric Reference Synthesis (Artifact)*. doi:10.5281/zenodo.14592164

[24] Daniel A. A. Pelsmaeker, Aron Zwaan, Casper Bach, and Arjan J. Mooij. 2025. Language-Parametric Reference Synthesis (Extended). (2025). doi:10.48550/arXiv.2502.19143 arXiv:arXiv:2502.19143 [cs.PL]

[25] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts.

[26] Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages* 2, POPL (2018). doi:10.1145/3158104

[27] 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). doi:10.1145/3428248

[28] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. 2009. Stepping Stones over the Refactoring Rubicon – Lightweight Language Extensions to Easily Realise Refactorings. In *23rd European Conference on Object-Oriented Programming (ECOOP '09)*.

[29] Max Schäfer and Oege de Moor. 2010. Specifying and implementing refactorings. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 286–301. doi:10.1145/1869459.1869485

[30] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Sound and extensible renaming for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 277–294. doi:10.1145/1449764.1449787

[31] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. 2012. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Trans. Software Eng.* 38, 6 (2012), 1233–1257. doi:10.1109/TSE.2012.13

[32] Friedrich Steimann. 2018. Constraint-Based Refactoring. *ACM Transactions on Programming Languages and Systems* 40, 1 (2018). doi:10.1145/3156016

[33] Frank Tip. 2007. Refactoring Using Type Constraints. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4634)*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 1–17. doi:10.1007/978-3-540-74061-2_1

[34] 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. doi:10.1145/2847538.2847543

[35] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Case Studies for Article: Scopes as Types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). doi:10.1145/3276915

[36] 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). doi:10.1145/3276484

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

[38] 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. doi:10.4230/LIPIcs.ECOOP.2021.1

[39] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2051)*, Aart Middeldorp (Ed.). Springer, 357–362. doi:10.1007/3-540-45127-7_27

[40] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, Baltimore, Maryland, United States, 13–26. doi:10.1145/289423.289425

[41] Aron Zwaan and Casper Bach Poulsen. 2024. Defining Name Accessibility Using Scope Graphs. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.47

[42] 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. doi:10.4230/OASIcs.EVCS.2023.32

Received 2024-10-15; accepted 2025-02-18