Sporq: An Interactive Environment for Exploring Code using Query-by-Example

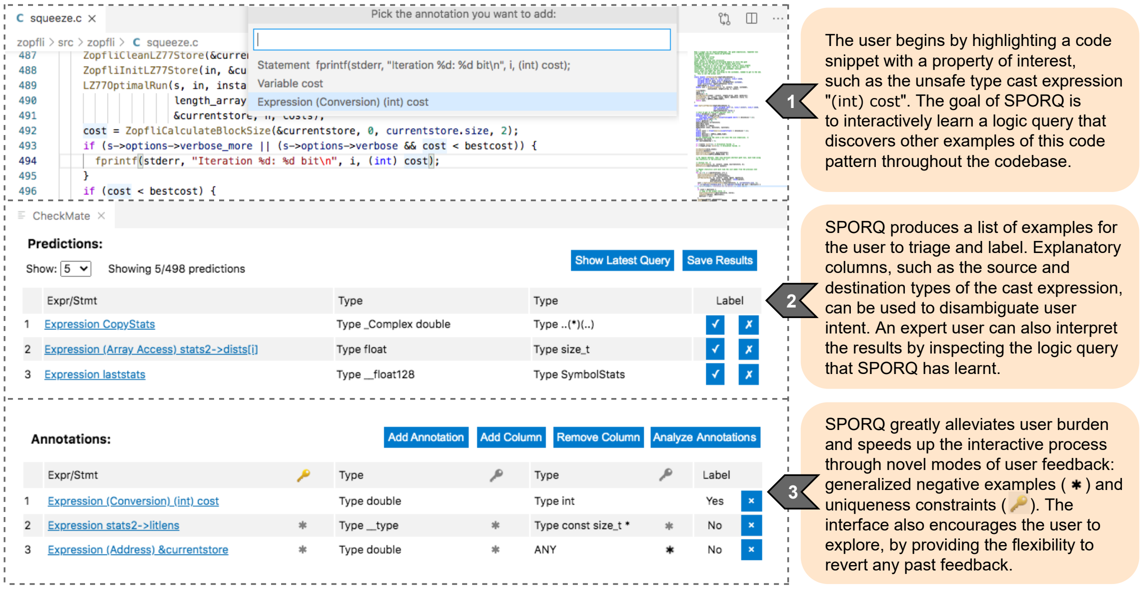
AADITYA NAIK, University of Pennsylvania, USA
JONOTHAN MENDELSON, University of Pennsylvania, USA
NATHANIEL SANDS, University of Southern California, USA
YUEPENG WANG, Simon Fraser University, Canada
MAYUR NAIK, University of Pennsylvania, USA
MUKUND RAGHOTHAMAN, University of Southern California, USA

There has been widespread adoption of IDEs and powerful tools for program analysis. However, programmers still find it difficult to conveniently analyze their code for custom patterns. Such systems either provide inflexible interfaces or require knowledge of complex query languages and compiler internals. In this paper, we present Sporq, a tool that allows developers to mine their codebases for a range of patterns, including bugs, code smells, and violations of coding standards. Sporq offers an interactive environment in which the user highlights program elements, and the system responds by identifying other parts of the codebase with similar patterns. The programmer can then provide feedback which enables the system to rapidly infer the programmer's intent. Internally, our system is driven by high-fidelity relational program representations and algorithms to synthesize database queries from examples. Our experiments and user studies with a VS Code extension indicate that Sporq reduces the effort needed by programmers to write custom analyses and discover bugs in large codebases.

ACM Reference Format:
Aaditya Naik, Jonothan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An Interactive Environment for Exploring Code using Query-by-Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3472749.3474737

1 INTRODUCTION

Programmers often need to explore patterns within their code. These may include patterns that typically raise alarms with compilers, static analyzers, and bug finders. However, they may include more specific patterns as well which may be beyond the scope of these tools. For example, to conform with the C++ Core Guidelines [52], the programmer may want to check that the 'auto' keyword has been used to avoid redundant repetition of type names, except in certain contexts, such as while declaring public class members. Java programmers may want to explore locations in the code where a thread is spawned. This would correspond to searching for indirect calls to the 'run()' method defined in all subclasses of 'java.lang.Thread'. Programmers may also wish to refactor their code. For example, consider a scenario where an API that is heavily used in a codebase received a major update, and as a result, each use of that API must be


C squeeze.c ×
zopfli > src > zopfli > C squeeze.c
Pick the annotation you want to add:
Statement fprintf(stderr, "Iteration %d: %d bit\n", i, (int) cost);
Variable cost
Expression (Conversion) (int) cost
CheckMate ×
Show Latest Query
Save Results
Add Annotation
Add Column
Remove Column
Analyze Annotations
1
2
3

```
    ZopfliCleanLZ77Store(&curre
    ZopfliInitLZ77Store(in, &cu
    LZ77OptimalRun(s, in, insta
                   length_array
                   &currentstore, n, costs);
    cost = ZopfliCalculateBlockSize(&currentstore, 0, currentstore.size, 2);
    if (s->options->verbose_more || (s->options->verbose && cost < bestcost)) {
      fprintf(stderr, "Iteration %d: %d bit\n", i, (int) cost);
    }
    if (cost < bestcost) {
```

Predictions:

Show: 5 Showing 5/498 predictions

	Expr/Stmt	Type	Type	Label	
1	Expression CopyStats	Type _Complex double	Type ..(*)(..)	✓	✗
2	Expression (Array Access) stats2->dists[i]	Type float	Type size_t	✓	✗
3	Expression laststats	Type __float128	Type SymbolStats	✓	✗

Annotations:

	Expr/Stmt	🔑	Type	🔑	Type	🔑	Label	
1	Expression (Conversion) (int) cost		Type double		Type int		Yes	×
2	Expression stats2->litlens	✱	Type __type	✱	Type const size_t *	✱	No	×
3	Expression (Address) &currentstore	✱	Type double	✱	ANY	✱	No	×

1 The user begins by highlighting a code snippet with a property of interest, such as the unsafe type cast expression "(int) cost". The goal of SPORQ is to interactively learn a logic query that discovers other examples of this code pattern throughout the codebase.

2 SPORQ produces a list of examples for the user to triage and label. Explanatory columns, such as the source and destination types of the cast expression, can be used to disambiguate user intent. An expert user can also interpret the results by inspecting the logic query that SPORQ has learnt.

3 SPORQ greatly alleviates user burden and speeds up the interactive process through novel modes of user feedback: generalized negative examples (✱) and uniqueness constraints (🔑). The interface also encourages the user to explore, by providing the flexibility to revert any past feedback.

Fig. 1. Annotated screenshot of the user interface of Sporq.

changed to comply with the latest version. As a result, the programmers maintaining the codebase would want to find all outdated uses of that particular API.

When programmers want to answer such questions about their code, they have access to a range of sophisticated developer tools. Still, they are often forced to compromise between ease of use and flexibility of application. IDEs provide convenient preset interfaces, but are typically limited to locating the declarations and uses of program elements. These presets tend to prove insufficient for tasks requiring a deeper understanding of the programs. Linters and program analyzers allow programmers to access deep facts about their programs. However, these tools are targeted towards general developers and support only common analyses out of the box. In a survey conducted by Microsoft [15], the main obstacle to using static analyzers cited by respondents was the mismatch between their needs and the default settings of the static analyzer. The survey also mentions that 21% of developers who stopped using static analyzers did so because they could not find one that fit their needs. If developers wish to perform analyses that are more specific to their needs, they would need to either customize existing tools or build their own. However, in an earlier study [25], 17 out of 20 participants criticize existing tools for not accommodating the customizations that they want. Additionally, extending such tools for specific use cases requires specialized knowledge, such as of compiler internals. Recent tools like Semmle [45] provide users with a powerful query model which allows users to potentially write their own analyses, but such tools require knowledge of complex query languages and expose unusual user interfaces.

In this paper, we introduce Sporq, an interactive environment that makes it easier for programmers to perform custom analyses such as the ones described above. The user begins by highlighting a snippet with some property of interest (Figure 1-1). The desired property can range from specific classes of bugs (such as casting instances of a supertype to variables of its subtype) to code smells (such as checking floating-point numbers for equality) and even more general instances of programming patterns (such as pairs of mutually recursive functions). The system responds by inferring the user intent and identifying other snippets with similar properties (Figure 1-2). By labelling the returned examples as being desirable or undesirable, the user provides feedback which enables the system to refine its hypothesis (Figure 1-3). Over several rounds of feedback, the system converges to an understanding of the user intent and accurately identifies other instances of the same pattern in the codebase.

Notably, the programmer needs no specialized knowledge beyond a general understanding of types, expressions, and statements, since the primary mode of interaction involves the user highlighting, triaging, and labelling examples, as depicted in Figure 1. Still, the user is free to examine the learned query, which is expressed in Datalog [1], a logic programming language which has found a variety of applications in program analysis [51], knowledge discovery [27], and database query engines [14]. This can provide expert users with explanations of predictions made by the system and facilitate trustworthy interaction. This stands in stark contrast to interfaces that employ machine learning models such as deep neural networks that are impractical for humans to interpret.

The flexible interface exposed by Sporq encourages programmers to explore: by freely labelling snippets returned by the system or by withdrawing previous feedback, users can provide speculative labels or even change their minds, and are therefore under no pressure to provide accurate feedback. They are also free to add explanatory columns, which can clarify why a particular snippet of code is interesting, thereby helping the system to disambiguate their intent.

Internally, the system represents the program as a relational database. We build on recent work on Semmle's CodeQL, which analyzes code in a range of languages including C/C++, Python, Java, and JavaScript, and extracts tables that collectively describe diverse aspects of the program, including its syntax tree, expression types, call graphs and data-flow relations [45]. These relational representations enable the use of powerful query formalisms such as Datalog to extract facts and draw conclusions about the program, and have emerged as a popular approach to creating program analysis tools [11, 38, 56]. Sporq analyzes these tables and synthesizes a Datalog query which is consistent with the labels provided by the user.

The central challenge is to scale the synthesis procedure to the large databases produced by Semmle. On moderate-to-large sized C programs comprising 10k–129k lines of code, these databases range in size from 80k to 928k tuples over a relational schema comprising 68 tables. GenSynth [35], a state-of-the-art synthesizer based on evolutionary search, takes ∼ 200 seconds on a database comprising 1k tuples in a single table. We extend GenSynth with three important optimizations: parallelizing the search algorithm by taking into account available resources on the host machine, reusing the results of intermediate computations across the evaluation of different candidate queries, and pruning irrelevant portions of the database using insights from our application domain. Together, these optimizations enable to scale GenSynth to realistic codebases in Sporq's interactive setting with real-time expectations.

An important aspect of Sporq's usability concerns its *sample efficiency*, i.e., the number of labels needed to identify the desired query. To accelerate this process, in addition to the traditional binary labels provided to examples, we introduce two new feedback mechanisms which allow the user to efficiently prune large parts of the search space. The first is a mode which we call *generalized negative feedback*, by which the user can label an entire set of rows as being undesirable. Furthermore, we allow the user to identify specific columns as primary keys, thus allowing the system to reject candidate queries that violate the implied database integrity constraints. Taken together, generalized negative feedback and uniqueness constraints significantly improve the sample efficiency of the learning algorithm, allowing us to discover the target query with an average of less than 6 labels on real programs.

We have implemented a prototype of Sporq as a VS Code extension for C/C++. We evaluate our prototype on a suite of 17 diverse code patterns applied to 5 programs, including well-known open-source projects such as the GNU core utilities, MySQL C++ Connector, and the Fish shell. In all these cases, Sporq identifies the correct pattern in fewer than 7 rounds of feedback, with each round of synthesis requiring less than 50 seconds on average. In addition, we conducted a small user study involving 5 participants and 2 code patterns from our suite. The study indicates that our system is effective at helping users learn patterns in real codebases and is easier to use than two state-of-the-art tools, Semmle [45] and Semgrep [17].

Contributions. To summarize, we make the following contributions in this paper:

(1) We present a new method for programmers to find patterns in their code by interactively highlighting examples and providing feedback on automatically generated suggestions.
(2) We introduce new feedback modes, including generalized negative examples and uniqueness constraints, which significantly improve the sample efficiency of query-by-example techniques.
(3) Using a prototype VS Code extension, we demonstrate the superior usability of our technique through a small user study in which participants find bugs in large programs.
(4) Finally, we conduct an experimental evaluation which shows that Sporq can infer sophisticated checkers on real programs with small amounts of labelled data and quick turnaround times.

The rest of the paper is organized as follows. Section 2 provides an illustrative overview of Sporq's user interface. Section 3 reviews relational program representations and the Datalog query language that underlie Sporq. Section 4 describes the design and implementation of the Sporq system. Section 6 presents findings of a user study and Section 5 empirically evaluates Sporq. Section 8 discusses threats to validity of our studies and limitations of Sporq. Section 7 discusses related work and finally Section 9 concludes.

2 ILLUSTRATIVE OVERVIEW

In this section, we present an overview of Sporq from the perspective of the user who wishes to identify some meaningful pattern in their codebase. For each scenario covered in the discussion in Section 1, the user interaction model is the same. As a running example, we consider an unsafe cast checker for C programs, which finds snippets of code that cast a value of floating-point type (such as `float` or `double`) to an integer type (such as `int` or `long`). Such casts are potentially unsafe because they can cause a loss in precision, produce unexpected results, or even have security implications [16].

Any tool for exploring variations of a code snippet must provide a mechanism for the user to annotate the relevant program elements. In the case of the unsafe cast checker, these elements include at least the cast expression, and possibly also the source and destination types as additional cues to clarify why a particular instance of a cast is unsafe. The mechanisms provided by existing environments to identify such elements are either limited in expressiveness or cumbersome to use, as we will now explain.

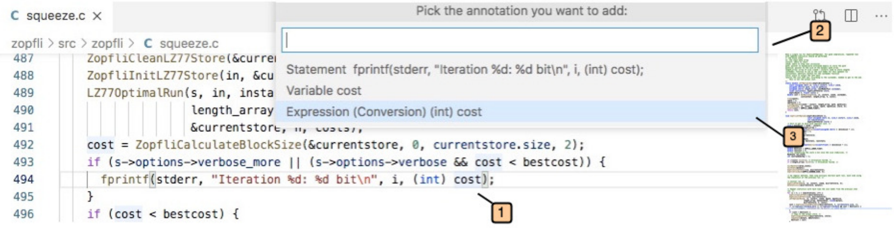
Semgrep [17] allows users to write patterns at the familiar level of their source code. An example of such a pattern is "`(int $A) = (int) (float $B)`". Here, `$A` and `$B` are meta-variables which the Semgrep engine can match against any instance of a program variable. However, this pattern is overly specific, is limited to casts in copy assignment statements such as "`x = (int) y`" and, moreover, is restricted to cases where the types of the variables `x` and `y` are `int` and `float` respectively. Notably, it does not extend even to slight variations of this pattern, such as "`x = (int) (y + 2.0)`".

On the other hand, Semmle [45] requires patterns over the program's abstract syntax tree (AST) to be expressed in a domain-specific language CodeQL [6]. The desired pattern of unsafe casts can be expressed in CodeQL as follows:

```
from Conversion c
where c.getUnspecifiedType() instanceof IntegralType
  and c.getUnspecifiedType() instanceof FloatingPointType
select c
```

In order to construct this query, the user needs to understand CodeQL's extensive and unfamiliar library functions as well as its particular syntax. The intended use case of CodeQL is for security researchers to write broadly applicable checkers, which programmers then apply to their code. It is challenging for non-expert users to write CodeQL queries, and GitHub even awards a bounty for specifying new security vulnerabilities in CodeQL [28].

Sporq provides the best of both worlds—the familiarity of the textual source code representation and the flexibility of the relational AST representation. The interaction between the user and Sporq may be broadly divided into three phases, which we will now describe.

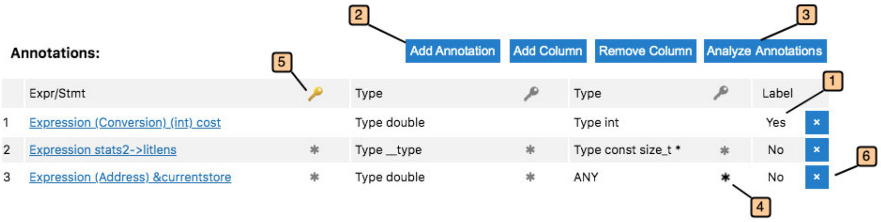

squeeze.c
zopfli > src > zopfli > squeeze.c
Pick the annotation you want to add:
1
2
3

Statement fprintf(stderr, "Iteration %d: %d bit\n", i, (int) cost);
Variable cost
Expression (Conversion) (int) cost

```
487      ZopfliCleanLZ77Store(&curre
488      ZopfliInitLZ77Store(in, &cu
489      LZ77OptimalRun(s, in, insta
490                         length_array
491                         &currentstore, n, costs);
492      cost = ZopfliCalculateBlockSize(&currentstore, 0, currentstore.size, 2);
493      if (s->options->verbose_more || (s->options->verbose && cost < bestcost)) {
494        fprintf(stderr, "Iteration %d: %d bit\n", i, (int) cost);
495      }
496      if (cost < bestcost) {
```

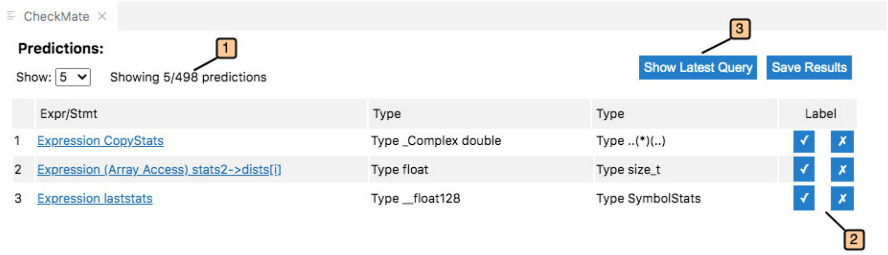
Fig. 2. Identifying the seed snippet within Sporq.

Part 1: Identifying the seed snippet. The user begins by discovering a snippet with the desired property, for example, the `double` to `int` conversion occurring in the expression "`(int) cost`" on line 494 of the file `squeeze.c` (Figure 2-1). As we show in Figure 2, the user conveys their intent to search for unsafe casts by selecting this snippet in the IDE. In case of ambiguity, Sporq presents a list of all program elements anchored to the selected location in the program (Figure 2-2), allowing the user to pick the element most relevant to the downcast (Figure 2-3). The user then proceeds to similarly select the source and destination types, by hovering over the `cost` and `int` sub-parts of the expression, and identifying the relevant program elements from a list of suggestions provided by Sporq. At this point, the user has effectively defined the space of all possible examples of unsafe casts, which can be depicted in tabular form with three columns, corresponding to the location in the program, and the source and target types respectively. We show three such examples in Figure 3.



	Expr/Stmt	🔑	Type	🔑	Type	🔑	Label	
1	Expression (Conversion) (int) cost		Type double		Type int		Yes	×
2	Expression stats2->litlens	*	Type __type	*	Type const size_t *	*	No	×
3	Expression (Address) &currentstore	*	Type double	*	ANY	*	No	×

Fig. 3. Labelled examples within Sporq.



Predictions:

Show: 5

Showing 5/498 predictions

	Expr/Stmt	Type	Type	Label	
1	Expression CopyStats	Type _Complex double	Type ..(*)(..)	✓	✗
2	Expression (Array Access) stats2->dists[i]	Type float	Type size_t	✓	✗
3	Expression laststats	Type __float128	Type SymbolStats	✓	✗

Fig. 4. Examples predicted by Sporq.

Part 2: Disambiguating user intent. At this point, the user has identified the first positive example, as indicated by the label "Yes" (Figure 3-1). By clicking on the button named "Add Annotation" (Figure 3-2), they may proceed to label more examples as positive or negative. At any point, the user can click on the button named "Analyze Annotations" (Figure 3-3) to ask Sporq to extrapolate and find other snippets in the codebase which are similar to the positive examples and dissimilar from the negative examples.

At this point, Sporq typically produces a large number of predictions (such as the 498 predictions made in Figure 4-1). This is because of Sporq's inability to understand the user's intent from a single positive example. The user may now proceed to additionally label any number of the displayed examples as positive by clicking the check button, or negative by clicking the cross button (Figure 4-2), and instruct Sporq to re-analyze the annotations. In each round of analysis, Sporq learns a logic query that is consistent with all the examples labelled so far. In successive rounds, the learned query converges to the desired pattern of unsafe casts. We show the final query in Figure 5a.

Observe that Sporq is designed to allow users to describe the desired pattern *without ever inspecting the underlying query*—throughout the interaction, it suffices to provide binary feedback on examples. However, a key advantage of learning a programmatic query lies in its explainability. Sporq therefore allows the user to inspect the current query by clicking on the button named "Show Latest Query" (Figure 4-3). While we intend this feature to be primarily used by expert users, it may even aid non-expert users to gauge whether Sporq is making progress towards identifying the desired pattern.

Part 3: Generalizing user feedback. Sporq can only be practically useful if it is able to learn the user's intent from a small number of labelled examples. *Sample efficiency* is therefore one of our central considerations in the design of the system and in our evaluation in Sections 5 and 6.

Consider the situation of the user providing a negative label for a prediction, (e, `double`, `int`). This feedback only states that the expression e does not constitute an unsafe cast from type `double` to type `int`. The synthesizer is free to make other suggestions with the same expression, such as (e, `float`, `int`), as long as the learned pattern is consistent with the provided labels. The system is even free to make other spurious predictions, such as (`&currentstore`, `double`, `double`), where the highlighted expression, `&currentstore`, does not even correspond to a type cast. As a result, if the user only labels individual examples as positive or negative, the search space might not be sufficiently constrained, and the system might require a prohibitively large number of examples to learn the target query.

We therefore introduce two new modes of feedback—generalized negative examples and uniqueness constraints—that dramatically reduce the user's annotation burden. We now describe each of these.

While triaging a prediction such as (&currentstore, double, double), the user may conclude that the expression &currentstore does not correspond to an unsafe cast, regardless of the values of the other two columns. They are therefore in a position to not just disallow this prediction, but also any other prediction of the form (&currentstore, ANY, ANY). By clicking on the asterisk in the corresponding columns (Figure 3-4), the user can conveniently provide negative feedback on this entire set of potential predictions, thereby pre-emptively eliminating a large number of candidate queries. We call feedback of this form a *generalized negative example*.

Conversely, Sporq also allows the user to generalize feedback for positive examples by specifying *uniqueness constraints*. Consider a specific expression e. Left unconstrained, the pattern learned by our system may simultaneously produce multiple predictions involving e, such as (e, double, int) and (e, float, int). At least one of these predictions has to be incorrect, since any given cast expression has unique source and target types. We allow the user to describe such properties—which correspond to key constraints in relational databases—by clicking on the icon of the key in the left-most column (Figure 3-5). This constrains Sporq to never produce multiple predictions with the same value of the first column. In other words, the predicted snippets are uniquely determined by the value of the first column. Uniqueness constraints dramatically generalize the information obtained from positive examples, such as from the snippet in the first row, since the unlabeled examples with (source, destination) types other than (double, int) are implicitly labeled negative.

Finally, Sporq allows the user to withdraw any feedback at any point during the interaction process (Figure 3-6). This feature relieves users from the pressure of having to provide accurate feedback upfront. It also encourages them to explore freely and revisit past decisions based on knowledge gained during exploration, such as by discarding the label they provided to an example, adding explanatory columns, or by removing uniqueness constraints.

3 RELATIONAL PROGRAM REPRESENTATIONS

The choice of program representation is crucial to the design of any compiler, program analyzer, or IDE. Sporq employs a relational representation which represents a program P as a set of tabular relations I. Two key procedures interest us: (a) compiling the program to a database that conforms to a schema for the program's language (e.g. C or Python), and (b) querying the database using a declarative logic programming language akin to SQL. These procedures correspond to the tasks of data ingestion and query evaluation respectively. We denote them by the functions compileProgram and evaluateQuery respectively, and describe each of them in order below.

Codebases as Databases: The compileProgram **Function.** In designing Sporq, we build on Semmle's industrial-strength CodeQL which defines schemas for common programming languages including C/C++, C#, Go, Java, Javascript, and Python. Sporq also leverages Semmle's custom compilers which parse codebases written in these languages and generate databases that conform to their corresponding schemas.

As an example, we depict the portions of Semmle's schema for the C language which are relevant to our unsafe cast checker in Figure 5b. For instance, the Expr relation contains a row (tuple) for each expression that occurs in the parse tree of the C program. The row includes the following columns: *id*, an integer value that uniquely identifies the expression, which is underlined to indicate that it is a key of the relation; *kind*, an integer value which indicates the kind of expression; and *location*, an integer value which indicates the expression's location in the codebase.

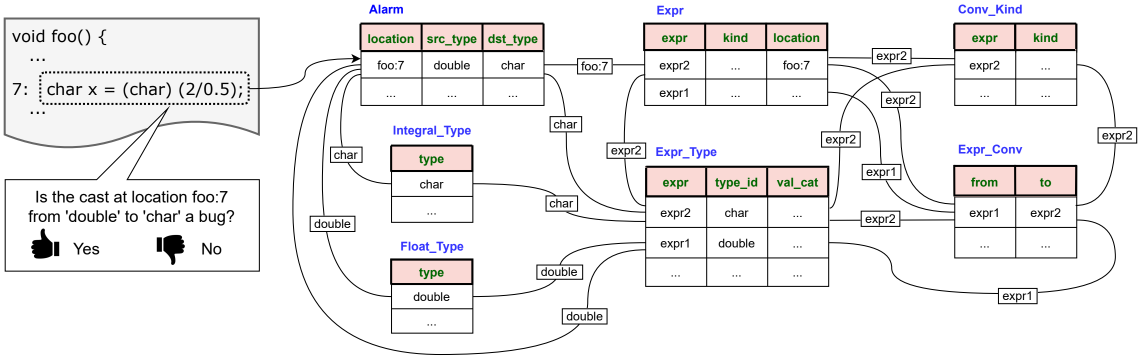
The relations are populated from a given C program using Semmle's C compiler. Figure 5c shows relevant parts of these relations generated from a small code snippet (we defer explaining the meaning of the Alarm relation and the labeled edges between rows to later in this section). The database generated for a full-fledged program can comprise

```
Alarm(location, src_type, dst_type) :-
    Expr(expr2, _, location),
    Conv_Kind(expr2, _),
    Expr_Conv(expr1, expr2),
    Expr_Type(expr1, src_type, _),
    Expr_Type(expr2, dst_type, _),
    Float_Type(src_type),
    Integral_Type(dst_type).
```

(a) Unsafe cast checker in Datalog.

```
Expr(int id : @expr, int kind : int ref, int location : @location_expr ref);
Conv_Kind(int expr_id : @cast ref, int kind : int ref);
Expr_Conv(int converted : @expr ref, int conversion : @expr ref);
Expr_Type(int id : @expr ref, int typeid : @type ref,
          int value_category : int ref);
Float_Type(int id : @type ref);
Integral_Type(int id : @type ref);
... // 192 more relations
```

(b) Relational schema of C programs (partial).



```
void foo() {
  ...
7: char x = (char) (2/0.5);
  ...
```

Is the cast at location foo:7 from 'double' to 'char' a bug?

Yes No

Alarm

location	src_type	dst_type
foo:7	double	char
...	...	...

Expr

expr	kind	location
expr2	...	foo:7
expr1	...	...

Conv_Kind

expr	kind
expr2	...
...	...

Integral_Type

type
char
...

Expr_Type

expr	type_id	val_cat
expr2	char	...
expr1	double	...
...	...	...

Expr_Conv

from	to
expr1	expr2
...	...

Float_Type

type
double
...

(c) Relational representation of the inputs (program facts) and outputs (example labels) of the unsafe cast checker.

Fig. 5. Illustration of how programs, examples, and queries are represented in Sporq.

hundreds of thousands of tuples. For instance, the GNU coreutils project comprising 90K lines of C source code results in a database comprising 367,628 tuples.

Querying Code in Datalog: The evaluateQuery **Function.** Representing codebases as databases enables the use of logical formalisms to query them, write custom bug checkers, and even specify entire program analyses. Sporq employs the logic programming language Datalog [1] which has emerged as a popular query language in the program analysis literature due to its concise and declarative nature, rich expressivity, and efficient off-the-shelf solvers [4, 44, 56].

A Datalog query Q comprises a set of rules that specify how to compute a set of output relations from a set of input relations. As an example, Figure 5a shows a Datalog query specifying the unsafe cast checker. It comprises a single rule, seven input relations (Expr, Conv_Kind, etc.), and a single output relation Alarm. A rule is a Horn clause of the form:

$$R_h(\bar{x_h}) :- R_1(\bar{x_1}), ..., R_n(\bar{x_n}).$$

where the $\bar{x_i}$'s are vectors of variables of appropriate arity. The rule is read from right-to-left as a universally quantified implication: for all variable valuations $\bar{x}$, if each of tuples $R_1(\bar{x_1}), \ldots, R_n(\bar{x_n})$ are derivable, then so is $R_h(\bar{x_h})$.

The evaluation of a query begins with the tuples in the input relations, and repeatedly applies the rules in arbitrary order to derive new tuples, until no further tuples can be derived. In the case of our unsafe cast checker, which is non-recursive, a single application of the single rule suffices. Sporq uses the Souffle solver [44] which incorporates sophisticated optimizations to efficiently evaluate Datalog queries. For instance, evaluating the unsafe cast checker on the GNU coreutils project takes only 0.029 seconds.

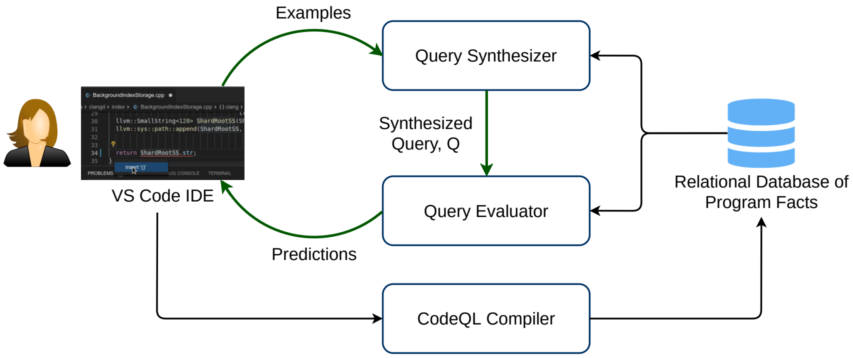


Fig. 6. System architecture of Sporq.

4 DESIGN AND IMPLEMENTATION OF THE SPORQ SYSTEM

In this section, we describe the main technical contributions in developing Sporq, whose architecture we describe in Figure 6. We formally present the interaction loop in Algorithm 1. We start with a statement of the Datalog query synthesis problem in Section 4.1, describe our optimizations to the GenSynth evolutionary program synthesizer in Section 4.2, and conclude with a description of our prototype implementation in Section 4.3.

4.1 The Datalog Query Synthesis Problem

The computational core of Sporq involves the synthesis of Datalog queries from examples provided by the user. As the user provides positive and negative labels to individual snippets, we add or remove the corresponding tuples from the `Alarm` relation. For example, consider the snippet shown at the far left of Figure 5c. If the user labels the type cast on line 7 of `foo()` as positive (i.e., a bug), then the corresponding tuple (`foo:7`, `double`, `char`) is added to the `Alarm` relation. Then, clicking the button named "Analyze Annotations" in the user interface (Figure 1) triggers a call to a Datalog query synthesizer, which we denote by the synthesizeQuery function and explain below.

Programming-by-example (PBE) systems have traditionally expected the user intent to be described using a set of concrete positive and negative examples. In our setting, a *concrete example* is a tuple t, and may be labelled as either positive or negative, indicating whether it is desirable or undesirable respectively. Since our examples are provided by the user in response to a previous synthesis attempt, we synonymously refer to these example tuples as *feedback*. We may formalize synthesis from concrete examples as the following computational problem:

Problem 1 (Synthesis from concrete examples)**.** *Given the set of input relations I, the set of concrete positive examples O_c^+ and concrete negative examples O_c^-, find a query Q which: (a) produces all positive examples, i.e., $O_c^+ \subseteq \mathrm{evaluateQuery}(I, Q)$, and (b) which does not produce any negative examples, i.e., $O_c^- \cap \mathrm{evaluateQuery}(I, Q) = \emptyset$.*

The problem of synthesizing logic queries from relational input-output data forms the subject of a sub-field of AI called Inductive Logic Programming (ILP). We refer the reader to Cropper et al. [18] for a comprehensive survey.

As described in Section 2, one of the central challenges in building a system such as Sporq is to infer the user's intent from small amounts of labelled data. To achieve this, we would like to enable the user to provide as much information about the target concept as is conveniently possible. We therefore introduce the concept of a *generalized negative example*, which is a tuple t^* where some of the columns have been replaced with a special don't-care term, "$*$". We say that a concrete tuple t *matches* a generalized tuple t^* if the two tuples agree on all columns i where $t_i^* \neq *$. We denote this by the notation $t \sim t^*$, formally defined as $\forall$ column indices i, $t_i^* \neq * \implies t_i = t_i^*$.

Consider the tuple $t = (\texttt{foo:7}, \texttt{double}, \texttt{char})$, and the generalized tuple obtained by replacing its second and third columns with $*$: $t^* = (\texttt{foo:7}, *, *)$. Since the two tuples agree on the first column, we write $t \sim t^*$. Consider another tuple $t' = (\texttt{foo:10}, \texttt{double}, \texttt{char})$. Since t' and t^* do not agree on the first column, $t' \nsim t^*$. If the user indicates that they are uninterested in line 7 of the file `foo.c`, then by enforcing the generalized negative example t^*, they eliminate not just t, but other matching tuples, such as $t'' = (\texttt{foo:7}, \texttt{char}, \texttt{double})$.

The positive and negative examples (concrete or generalized) respectively provide lower and upper bounds on the output of the synthesized query Q. Users can sometimes provide additional information about the structure of their intended query. For example, they know that the first component—indicating the location of the cast—is the central component of the output, and that the remaining two columns simply indicate the source and destination types, both of which are uniquely determined by the first column. By building on the closely related concept of integrity constraints in relational databases, we introduce the idea of *uniqueness constraints* for query-by-example systems.

Formally, let $K \subseteq S$ be a set of user-specified columns, where S is the schema of the output relation. We say that a query Q satisfies the uniqueness constraints imposed by K with respect to the input relations I if each pair of output tuples, $t, t' \in \text{evaluateQuery}(I, Q)$ disagree on some element of K: $\exists$ some column $i \in K$ such that $t_i \neq t'_i$.

By adding generalized negative examples and uniqueness constraints to the previous problem statement, we obtain the generalized QBE problem that we solve in Sporq. We refer to any procedure which solves this problem with the function $\text{synthesizeQuery}(I, O_c^+, O_n^-, K)$.

Problem 2 (Synthesis from generalized examples)**.** *Given the set of input relations I, the set of concrete positive examples O_c^+, generalized negative examples O_n^-, and key constraints K, find a query Q which: (a) produces all positive examples, i.e., $O_c^+ \subseteq \text{evaluateQuery}(I, Q)$, (b) none of whose output tuples match any generalized negative example, i.e., $\forall t \in \text{evaluateQuery}(I, Q), \forall t^* \in O_n^-, t \nsim t^*$, and (c) which satisfies the uniqueness constraints imposed by K with respect to I.*

4.2 Synthesizing Queries Using Evolutionary Search

Algorithm 1 Sporq(P, t). Given a program P and seed example t, runs the main interaction loop.

```
(1) Let $I = \text{compileProgram}(P)$. Compile the program using CodeQL and initialize the input relations.
(2) Initialize the synthesis constraints, with positive feedback, $O_c^+ := \{t\}$, negative feedback, $O_n^- := \emptyset$, and key
    constraints, $K = S$, where the schema $S$ is the set of all columns in the output.
(3) Repeat until the user stops responding:
    (a) Let $Q = \text{synthesizeQuery}(I, O_c^+, O_n^-, K)$. Find a query which is consistent with the user's feedback.
    (b) Let $O = \text{evaluateQuery}(I, Q)$. Evaluate the synthesized query to obtain the list of predictions.
    (c) Assert that $O$ is consistent with the synthesis constraints $(O_c^+, O_n^-, K)$.
    (d) Highlight the predictions in $O$ for triaging by the user.
    (e) Collect the new feedback provided by the user, $(\Delta^+, \Delta^-, \Delta_K)$, and update: $O_c^+ := O^+ \cup \Delta^+$, $O_n^- := O^- \cup \Delta^-$,
        and $K := K'$.
```

Sporq relies on GenSynth to solve Problem 2 [35]. At its core, GenSynth uses a genetic algorithm that maintains a population of Datalog programs, and operates by repeatedly mutating the programs of this population with small changes. After each batch of mutations, the programs which best fit the synthesis constraints are preserved for further mutation. We visualize this process in Figure 7. We represent the best performing programs of each generation with

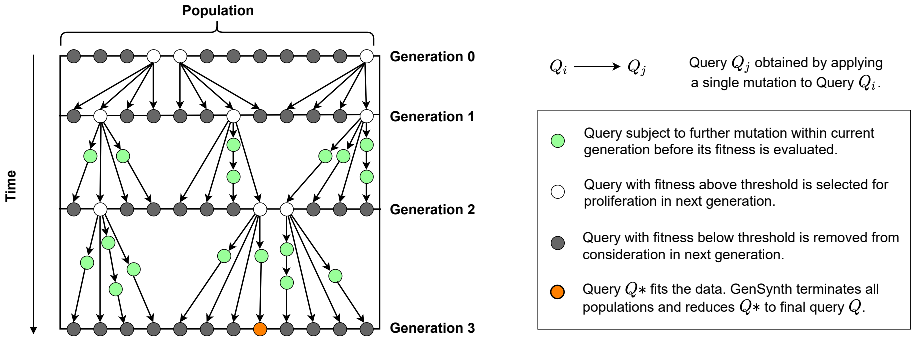

Population
Generation 0
Generation 1
Generation 2
Generation 3
Time
$Q_i \longrightarrow Q_j$
Query Q_j obtained by applying a single mutation to Query Q_i.
🟩 Query subject to further mutation within current generation before its fitness is evaluated.
○ Query with fitness above threshold is selected for proliferation in next generation.
● Query with fitness below threshold is removed from consideration in next generation.
🟠 Query $Q*$ fits the data. GenSynth terminates all populations and reduces $Q*$ to final query Q.

Fig. 7. Illustration of a single population of queries evolving over time in GenSynth.

white circles, which survive to the next generation, while the remaining programs, shaded in gray, are discarded. As a result, the fitness of the best program in the population monotonically increases over time, until it perfectly matches the user's constraints. By running multiple populations in parallel, GenSynth significantly reduces the time needed for a population to discover a solution. The synthesis algorithm used by GenSynth may be summarized as follows:

(1) Run $b = 64$ populations in parallel. Within each population:
   (a) Initialize the population P by sampling $n = 8$ seed queries from a distribution $\mathcal{D}$.
   (b) While $\max_{Q \in P}$ score$(Q) \lesssim 1$:
      (i) Construct the subsequent generation P' by mutating each query $s \sim$ Unif(1,7) times:

$$P' = \{\textsc{mutate}(Q) \mid Q \in P, i \in \{1, 2, \ldots, s\}\}.$$

      (ii) Update P to be the n queries in P' with highest scores:

$$P := \underset{Q' \in P'}{\text{top-}n}\,(\text{score}(Q')). \tag{1}$$

   (c) Kill the remaining populations and return the query $Q = \arg\max_{Q \in P}(\text{score}(Q))$.

GenSynth defines the scoring function score(Q) as the F_1 score of the query output evaluateQuery(I, Q) with respect to the positive and negative examples (O_c^+, O_c^-). To add support for generalized examples and key constraints, we modify the computation of the scoring function to compute the F_1 score with respect to the generalized examples, (O_c^+, O_n^-, K). We have also made extensive improvements to the tool to enable it to scale to the massive datasets produced by Semmle. We will now describe these modifications.

Scheduling multiple populations over physical worker threads. One important challenge with using GenSynth as the query synthesizer involves adapting it to run on commodity hardware used by programmers. Therefore, instead of running all $b = 64$ populations in parallel, which would involve a large number of processor threads, we instead schedule these logical populations across a set of $k = 4$ physical threads. We spawn these k worker threads, each of which is responsible for b/k populations. The threads in turn cycle through their assigned populations, running one generation of each before switching to the next. This allows us to achieve the benefits of a large number of populations—which greatly reduces the time to reach a solution—while still running on most developer machines, which typically only have 2 or 4 physical cores.

Population-wide batching of query evaluation. Computing the score of the queries in Equation 1 requires evaluating the queries $Q' \in P'$ with calls to $\text{evaluateQuery}(I, Q')$. GenSynth uses Souffle, a state-of-the-art Datalog solver for query evaluation. In large datasets such as those extracted by Semmle, a significant fraction of the time is spent by the Datalog solver in loading the relations I from disk. Our second optimization therefore involves batching all of the queries of a generation into a single call to Souffle, thereby amortizing the time needed to load the relations.

The idea is to uniquely rename the output relation produced by each candidate query $Q' \in P'$ so the output produced by each query can be recognized. For example, say that $P' = \{Q_1, Q_2\}$ consists of two queries as follows:

$$Q_1 : \ \texttt{Alarm}(x_0, x_1, x_2) :\!- \texttt{Expr}(x_0, x_2, x_1), !\texttt{Float\_Type}(x_2), \text{ and}$$

$$Q_2 : \ \texttt{Alarm}(x_0, x_1, x_2) :\!- \texttt{Expr}(x_0, x_2, x_1), \texttt{Expr\_Conv}(x_3, x_0), \texttt{Integral\_Type}(x_2).$$

Instead of separately invoking $\text{evaluateQuery}(I, Q_1)$ and $\text{evaluateQuery}(I, Q_2)$, we rename the output relations of the two queries as follows:

$$Q_1' : \ \texttt{Alarm1}(x_0, x_1, x_2) :\!- \texttt{Expr}(x_0, x_2, x_1), !\texttt{Float\_Type}(x_2), \text{ and}$$

$$Q_2' : \ \texttt{Alarm2}(x_0, x_1, x_2) :\!- \texttt{Expr}(x_0, x_2, x_1), \texttt{Expr\_Conv}(x_3, x_0), \texttt{Integral\_Type}(x_2),$$

and evaluate both of them with a single call to Souffle, as $\text{evaluateQuery}(I, \{Q_1', Q_2'\})$. Sharing computation across multiple queries in this manner greatly reduces the amount of time spent within Souffle.

Pruning the input database for scalable synthesis. Despite the previous optimizations, scaling the synthesis procedure remains an challenge. Our solution in Sporq is to perform aggressive pruning of the input relations I. We perform this pruning at three levels: First, at the level of the source code, we restrict our attention to files, functions, and code units which are of interest to the programmer, and exclude libraries and other source units which are unlikely to contain bugs. Second, at the level of the schema, we only consider those relations which are likely to be relevant to the problem. Finally, we assume that the target concept can be explained with a short query, and that all input tuples which are necessary to explain the positive examples are within $\beta = 2$ edges from the positive tuples in Figure 5c. Together, these three pruning methods—source-level pruning, schema-level pruning, and β-pruning—are necessary to enable our system to return results in a real-time manner. On the other hand, the architecture of Sporq is agnostic of the underlying synthesizer, and we can therefore naturally leverage research advances in efficiently solving Problem 2.

4.3 Implementing the IDE Extension

We chose to implement Sporq in TypeScript as an extension to the Visual Studio Code IDE. The interface sits side-by-side with the code editor, allowing users to easily compare their code with the results returned by Sporq without switching windows. Clicking on any of the links in the predictions or annotations pane immediately brings up the relevant section of code in the editor.

Internally, Sporq represents programs as relational databases, so the interface was designed to be similar to that of a visual database management system. The results are presented in tabular format in which columns can be added or removed. The user adds a uniqueness constraint by clicking on a key icon, which is identical to the icon denoting a primary key value within MySQL Workbench. The all-caps 'ANY' label which marks generalized negative examples is taken directly from standard SQL syntax, and the asterisk button is analogous to the wildcard character used in SQL 'SELECT' statements. One advantage of embedding Sporq within VS Code is that the IDE's own code suggestions can be used as prompts to craft new queries. For example, suppose a light bulb icon appears next to a line of code with

Pattern	Description	# Preds	# Cols	# Vars	# Negs
bool-compare	Boolean expression compared with integer expression.	5	9	5	1
cast-to-signed	Cast of int-types from unsigned to signed.	6	12	7	0
complex-conds	Overly complicated boolean conditions.	2	5	4	0
const-conds	Use of a boolean or integer literal as a condition.	3	6	4	0
const-void-funcs	Const member functions with void return type.	3	3	1	0
dangerous-funcs	Use of unsafe localtime function (CWE-676).	3	6	4	0
eq-bool-consts	Unnecessary comparison of boolean variable with boolean literal.	2	5	3	0
float-conversion	Cast from double-type to float-type.	6	12	7	0
long-inline-funcs	Inline-qualified functions with more than 10 SLOC.	2	3	2	0
mutual-recursion	Mutually recursive functions.	5	10	4	1
pointer-arith	Use of pointer arithmetic.	3	9	7	0
sign-compare	Comparison of signed types with unsigned types.	8	16	9	1
string-compare	Improper comparison of C-strings.	7	14	8	2
unsafe-casts	Unsafe cast from float-type to int-type.	6	21	15	0
unsigned-ge-zero	Greater-than or equal comparison of unsigned variable with zero.	3	6	4	0
unused-locals	Local variables that are never used after declaration.	3	4	2	1
use-goto	Use of goto statement.	2	4	3	0

Table 1. Description and statistics of code patterns in our evaluation.

a suggested refactoring action. The programmer may wish to explore whether similar lines of code exist elsewhere. Sporq can use this particular example to construct a matching query. In contrast to an IDE's own suggestions, which may be limited to the file that is currently open, Sporq can return results across the entire codebase.

Upon activation in a new project, we compile the codebase using CodeQL to construct the relational representation, and use a set of extractor queries to obtain comma-separated files that describe the input relations. We also use these extractor queries to obtain mappings from numeric identifiers used within CodeQL (such as to reference expressions, functions, and other program elements) to explicit locations in the source code for highlighting and to present to the user. Because the schema used by CodeQL is language-specific, adding support for a new language inside Sporq involves writing a new set of extractor queries. Our current prototype supports C and C++, but providing support for other languages is conceptually straightforward.

After providing the initial example and after each subsequent batch of feedback, the user clicks on the button titled "Analyze Annotations". At this point, Sporq maps concrete source locations to the underlying numeric identifiers, and invokes GenSynth on the current set of examples. Once synthesis is complete, the learned query is applied to the program to obtain the new set of predictions, which are in turn translated back to concrete source locations and presented to the user for triaging.

5 EXPERIMENTAL EVALUATION

To evaluate Sporq, we conduct experiments that aim to answer the following research questions:

RQ1 Is Sporq able to learn diverse code patterns on large real-world codebases?
RQ2 How quickly does Sporq synthesize the query between successive rounds of interaction?
RQ3 How much user feedback does Sporq need to learn the desired code pattern?

Codebase	Description	Language	SLOC	# Tuples
coreutils	File, shell, and text manipulation utilities of GNU operating systems	C	90k	367,628
fish-shell	Smart and user-friendly command line shell for macOS and Linux	C++	51k	558,094
mysql-conn	C/C++ database connector for MySQL servers	C++	129k	927,871
snap	Stanford network analysis platform	C++	84k	191,500
zopfli	Google library to perform good but slow deflate or zlib compression	C	10k	79,713

Table 2. Description and statistics of codebases in our evaluation.

5.1 Benchmarks

Patterns. To obtain realistic code exploration tasks for our evaluation, we collect existing code patterns from two sources: (1) static analysis frameworks like Semmle and Semgrep, and (2) warning rules from compilers like GCC. Overall, we collect 13 patterns from static analysis frameworks and 4 patterns from compiler warnings. These code patterns can match a diverse set of code snippets, ranging from potential bugs (e.g., comparing two strings using the equality operator), security vulnerabilities (e.g., use of dangerous functions like `localtime`), to coding styles (e.g., comparing boolean expressions against literals `true` and `false`). The characteristics of all 17 patterns are summarized in Table 1. In order to characterize the complexity of the patterns, we manually write a ground-truth Datalog query for each pattern and collect statistics about the ground-truth query. Specifically, the "# Preds" column in Table 1 shows the number of predicates in the body of the Datalog query. "# Cols" shows the total number of columns in the body, which is equal to the sum of the arities of all predicates in the body. The next column "# Vars" presents the number of variable occurrences in the body, and the last column "# Negs" shows the number of negated predicates.

Codebases. To find codebases that contain snippets matching the collected patterns, we write CodeQL queries for all of the 17 patterns and use the Semmle framework to search over Github repositories. We choose a set of 5 representative codebases as described in Table 2. The size of collected codebases ranges from 10k to 129k lines of source code, and the number of tuples in their corresponding databases ranges from 80k to 928k.

Experimental setup. To evaluate Sporq on a diverse set of patterns and real-world codebases, we make a list of 17 benchmarks, where each benchmark is a codebase-pattern pair (C_i, P_i) for $i \in \{1, \ldots, 17\}$ such that

(1) each pattern P_i is unique
(2) codebase C_i contains code snippets that match pattern P_i

For each pair of codebase C_i and pattern P_i, one of the authors ran Sporq on codebase C_i with uniqueness constraints and generalized negative examples as necessary, aiming to find all code snippets that match pattern P_i.

All of our experiments are performed on a Macbook Pro laptop with a 8-core 2.3GHz CPU and 16 GB of physical memory. We set the maximum rounds of interaction between the user and Sporq as 20 and stopped exploring the codebase when this bound was exceeded. We also set a timeout as 5 minutes for the underlying synthesizer. If the synthesis time goes beyond the limit in a particular run, we immediately concluded failure.

5.2 Main Results

Table 3 summarizes the evaluation results of all 17 benchmarks. Recall from Section 4.2 that we perform three levels of pruning on the input database. The column "# Tuples" in Table 3 describes the number of tuples in the pruned database. The next four columns present the results of interacting with Sporq. Specifically, "# Pos" and "# Neg" show

Codebase	Pattern	# Tuples	# Pos	# Neg	# Rounds	Avg Time (s)
coreutils	bool-compare	6,837	1.7	4.0	4.0	5.3
	mutual-recursion	4,503	1.0	0.0	1.0	6.4
	use-of-goto	14	1.7	1.0	2.0	2.1
	dangerous-funcs	14,711	1.0	1.0	2.0	2.1
fish-shell	cast-to-signed	1,253	1.0	4.3	5.3	17.1
	const-void-funcs	534	1.0	2.0	3.0	2.1
	const-conds	66	1.0	3.3	4.3	2.2
mysql-conn	sign-compare	4,229	1.3	4.7	5.7	46.8
	string-equality	2,361	1.3	4.3	5.3	5.5
snap	long-inline-funcs	23	1.3	0.7	1.7	1.2
	unsigned-ge-zero	644	1.3	2.3	3.3	2.7
	eq-bool-consts	58	1.3	1.7	2.7	1.9
	unused-locals	384	1.0	1.0	2.0	2.3
	complex-conds	95	2.0	2.0	3.0	1.7
zopfli	float-conversion	435	1.3	4.3	6.3	5.1
	unsafe-casts	1,278	2.0	4.0	5.0	10.1
	pointer-arith	432	1.0	0.0	1.1	3.5

Table 3. Main evaluation results of Sporq.

the number of positive and negative examples. "# Rounds" describes the rounds of interaction between the user and Sporq. "Avg Time" is the average synthesis time between two successive rounds of interaction. Since the synthesizer of Sporq performs evolutionary search that exhibits randomness in the process of generating mutations, we execute each benchmark for three times. The numbers reported in Table 3 are the average over three runs.

As shown in Table 3, Sporq is able to solve all of the 17 benchmarks with users providing feedback as needed. Given that all five codebases are collected from Github repositories with 10k – 129k lines of source code, and the 17 code patterns cover a variety of applications from compilers and static analysis frameworks, we believe that Sporq is able to learn diverse code patterns on realistic codebases (RQ1).

Furthermore, we observe that the average synthesis time is less than 10 seconds for 14 out of the 17 benchmarks, 10 – 20 seconds for 2 benchmarks, and more than 20 seconds for only 1 benchmark (46.8 seconds). All of the queries in the experiments are synthesized in less than one minute. These results demonstrate that Sporq can quickly synthesize the query between successive rounds of interaction (RQ2).

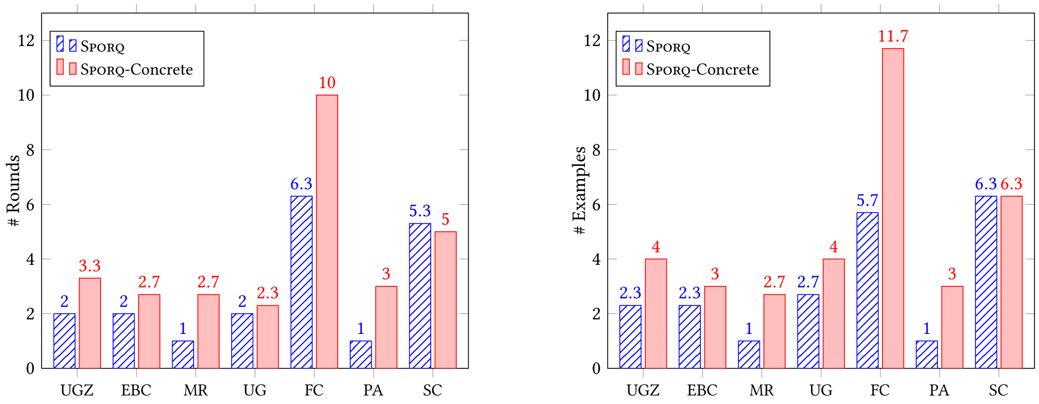
In addition, the evaluation results show that Sporq only a needs small amount of user feedback to learn the desired code pattern (RQ3). Specifically, Sporq learns the pattern in less than 5 rounds of interaction for 12 out of 17 benchmarks and 5 – 7 rounds of interaction for the remaining 5 benchmarks. Besides, the number of examples needed from the user is reasonably small. We provide less than 5 (positive plus negative) examples for 11 benchmarks and 5 – 7 examples for the remaining 6 benchmarks.

5.3 Impact of Uniqueness Constraints and Generalized Examples

To evaluate the impact of uniqueness constraints and generalized examples on the amount of needed feedback, we perform another set of experiments on the same benchmarks. However, in the new experiments, we do not use

	No Impact	Both ≤ 20 Rounds	Concrete > 20 Rounds	Concrete Timeout
# Benchmarks	6	7	2	2

Table 4. Comparison between concrete and generalized examples.



(a) Rounds of Interaction

(b) Number of Examples

Fig. 8. Impact of uniqueness constraints and generalized examples.

uniqueness constraints or generalized examples. Instead, we only provide concrete positive and negative examples as the feedback to Sporq.

The comparison results are summarized in Table 4. As shown in the table, there are 6 benchmarks where uniqueness constraints and generalized examples do not affect the user interaction. The reason is because those benchmarks only return one column as output, so uniqueness constraints and generalized examples are not applicable. For the remaining 11 benchmarks, uniqueness constraints and generalized examples have significant impact on learning desired code patterns using Sporq. In particular, for 7 out of 11 benchmarks, Sporq is able to learn the code pattern purely from *concrete* examples within 20 rounds of interaction. It requires more than 20 concrete examples to learn the desired pattern on 2 benchmarks. The underlying synthesizer in Sporq times out on a laptop after 5 minutes on the remaining 2 benchmarks after obtaining more than 10 concrete examples.

For the benchmarks where Sporq can learn the pattern with or without uniqueness constraints and generalized examples, Figure 8 shows the impact in more detail. The x-axis denotes different benchmarks by abbreviated names, e.g. UGZ for unsigned-ge-zero, EBC for eq-bool-consts, etc. The y-axis represents rounds of interaction in Figure 8(a) and total number of examples in Figure 8(b). Sporq-Concrete denotes the new experiment where we only provide concrete positive and negative examples as feedback.

As shown in Figure 8, Sporq only needs an average of 2.8 rounds of interaction with 3.0 examples to learn the desired pattern using uniqueness constraints and generalized examples, while it needs 4.1 rounds of interaction with 5.0 examples if the feedback only consists of concrete examples. Combined with the results from Table 4, we conclude that uniqueness constraints and generalized examples help to reduce the amount of user feedback for learning many desired code patterns.

6 USER STUDY

We perform a small-scale user study to evaluate Sporq on the following research questions:

RQ4 How effective is Sporq in helping users find code snippets of interest in realistic settings?

RQ5 How does Sporq compare to baseline tools Semgrep and Semmle in terms of ease of use?

6.1 Tasks

To design realistic code exploration tasks for the user study, we refer to the patterns and codebases described in Section 5.1. Due to limited resources, we select one representative pattern to create the user study task from each of the two sources, respectively.

Task 1: Unsafe cast checker. This task is adapted from Semmle [1]. It corresponds to our running example that finds casts from floating point types to integral types. There are three unsafe casts in the provided codebase.

Task 2: Bool integer comparison. This task is adapted from the GCC warning option `-Wbool-compare`. It identifies comparison operations between boolean expressions and integral expressions because such comparisons are more likely to be bugs despite being technically allowed by the language standard. There are nine such comparisons in the provided codebase.

We choose these tasks for two reasons. First, the code patterns in these tasks cover a set of concepts (e.g., expressions, types) that are commonly supported by all three tools. Second, these two problems are accessible to participants because they are familiar with the underlying concepts.

6.2 Participants

We choose 5 graduate students majoring in computer science as participants in the user study. We excluded one additional participant because of failure to follow our experimental instructions. The pre-study survey shows that the average familiarity of C/C++ among all participants is 4.2 on a 0–5 scale, with 0 being not familiar at all and 5 being an expert. Using the same scale, the familiarity score for Visual Code IDE is 3.4, and familiarity with program analysis techniques is 4.0. Note that all of the participants had some experience with program analysis. Since program analysis is hard, and since all three tools are targeted to programmers who wish to perform some form of analysis, it was necessary to choose participants who had some experience in the field. Of the 5 participants, 3 report having some experience with the CodeQL query language and the average familiarity score is 3.0, while the rest report not having any experience. Similarly, 2 participants have experience with Semgrep (average score: 1.5) but the rest have no experience. This choice of participants allowed us to gain some insight into the ease of use of Sporq compared to people with and without experience with Semmle and Semgrep.

6.3 Methodology

To answer RQ4 and RQ5, we ask each participant to conduct Task 1 and Task 2 using three different tools: Sporq, Semmle, and Semgrep. To ensure that experiments are conducted in the same setting for all participants, we provide a virtual machine with all of the tools installed and ask participants to perform experiments on the virtual machine.

To instruct participants how to use the three tools, we provide each participant with a tutorial and a demo video of Sporq, where the underlying codebase is different from those used in Task 1 and Task 2. We also provide the

[1] https://github.com/github/codeql/blob/main/cpp/ql/examples/snippets/castexpr.ql

Task	**Sporq**			**Semmle**		**Semgrep**	
	# Users	**# Interactions**	**Avg Time (m)**	**# Users**	**Avg Time (m)**	**# Users**	**Avg Time (m)**
Task 1	5	8.2	22.75	3	18.25	1	35.00
Task 2	3	6	13.81	2	13.47	1	19.21

Table 5. Results of user study involving 5 participants. **# Users** indicates the number of users who completed the task, while **# Interactions** indicates the average number of interactions needed by each user who completed the task. **Avg Time** is the average time in minutes taken by these users to complete the task.

documentation of the CodeQL query language used by Semmle and the instructions of Semgrep provided online. Participants are required to read the documents before they use each tool. They can also refer to online resources (e.g., Google or StackOverflow) if needed during the user study. We do not put a hard time limit for each task in the user study and do not specify the order in which participants use different tools. Each participant decides which tool to use first at their own discretion.

At the end of the user study, participants are asked to finish a post-study survey that investigates the interaction experience of each tool, as well as usefulness of different features supported by Sporq. They also provided comments and feedback for improving Sporq.

6.4 Results

Table 5 summarizes the main results of our user study. The "# Users" column in Table 5 refers to the number of users who successfully completed the task using the corresponding tool. The "# Interactions" column mentions the average number of interactions between the user and Sporq, while the "Avg Time" column reports the average time taken by the users in minutes to complete the task using the corresponding tool. As shown in Table 5, all 5 participants manage to solve Task 1, and 3 participants solve Task 2 using Sporq. Overall, participants spend an average of 22.75 minutes on Task 1 with 8.2 rounds of interaction with Sporq. For Task 2, it takes participants 13.81 minutes on average with 6 rounds of interactions. Therefore, we conclude that Sporq can help users to explore programs and find code snippets of interest in real codebases.

The reason why 2 of the users could not get the correct results for Task 2 using Sporq can be attributed to some form of user error. In both cases, the users labeled a correct prediction as false, and one of them had labeled an incorrect prediction as true, possibly due to not understanding the code snippet corresponding to the prediction they labeled.

By contrast, only 3 participants succeed in solving Task 1, and 2 in solving Task 2 using Semmle. We manually inspected all the CodeQL queries submitted by the users and found two reasons for the failure. First of all, finding relevant relations and libraries in a query is difficult for people unfamiliar with CodeQL. For example, one participant who does not have any experience with CodeQL wrote the following query:

```
from AssignExpr a
where a.getRValue().getType() instanceof FloatingPointType
  and a.getLValue().getType() instanceof IntegralType
select a
```

where `AssignExpr` is not related to the desired cast expressions. Second, even people experienced with CodeQL may not know the subtle difference between different API functions. As an example, one participant who is familiar with CodeQL (scored 4 out of 5) wrote the query:

Question	Metric	Answer
How easy is it to use Sporq?	0: hard - 5: easy	3.6
How easy is it to use Semgrep?	0: hard - 5: easy	1.2
How easy is it to use CodeQL/Semmle?	0: hard - 5: easy	3.4
How helpful are uniqueness constraints in Sporq?	0: not at all - 5: very helpful	2.6
How helpful are generalized negative examples in Sporq?	0: not at all - 5: very helpful	4.6
How helpful is it to show synthesized queries in Sporq?	0: not at all - 5: very helpful	1.6
How helpful is it to remove examples in Sporq?	0: not at all - 5: very helpful	4.2

Table 6. Post user study survey.

```
from Conversion c
where c.getType() instanceof IntegralType
  and c.getUnconverted().getType() instanceof FloatingPointType
select c
```

where the correct query should use `getUnspecifiedType()` instead of `getType()` because the latter does not resolve `typedef` and causes false negatives in the query result.

Similarly, only 1 participant managed to solve both Tasks 1 and 2 using Semgrep in 45.03 minutes for Task 1 and 31.97 minutes for Task 2. We also inspected the reason why participants fail to solve the given tasks by checking the Semgrep patterns. Instead of expressing the high-level concept, we found participants frequently wrote a union of ad-hoc rules. For example, one participant wrote a set of rules like "`(int) $X`" and "`(long) $X`" to match all variables cast to integer and long types. This might happen if a codebase has a small number of examples which are all covered by specific patterns such as above. However, such patterns may miss specific corner cases, for example, casts to short types. In general, writing patterns to match interesting code snippets in this way is not ideal. In Sporq's case, the user starts with more general queries and iteratively specializes them by labeling some of the results during each interaction. This allows Sporq to avoid over-fitting the examples labeled by the user which helps to prevent mistakes like the participant committed above.

Overall, our user study shows that more participants are able to solve Task 1 and Task 2 using Sporq than using Semmle or Semgrep. In addition, Table 6 summarizes the results of our post-study survey. Participants in our user study rate Sporq as 3.6 on a $0 - 5$ scale in terms of whether the tool is easy to use or not. In contrast to Sporq, Semmle gets 3.4 and Semgrep only gets 1.2 on ease to use. These results provide us with some preliminary evidence that Sporq is easier to use than Semmle and Semgrep (RQ5).

The survey results also suggest that users believe many features supported by Sporq are helpful for code exploration, including primary keys, generalized negative examples, and removing annotated examples. However, there was mixed feedback regarding whether inspecting the synthesized queries is useful. Participants with more experience in program analysis found it helpful, while participants with less or no experience disagree. This observation is reasonable because interpreting the synthesized query requires the users to have a good understanding of Datalog, and a background in program analysis. The result also indicates that Sporq can help users with different levels of experience in program analysis to explore codebases.

7 RELATED WORK

We survey related work classified into three categories: program analysis frameworks, synthesis of logic queries, and learning in program analysis.

Program Analysis Frameworks. Sporq can be viewed as a framework that allows users to craft custom program analyses. Examples of such frameworks widely used in industry include Findbugs [7], Coverity [9], Tricorder [43], Infer [12], and Clang Static Analyzer [33]. While these frameworks offer a variety of checkers, however, they are intended to be extended only by experts with specialized knowledge.

Recognizing the rich diversity of modern programming environments and application domains, a modern generation of frameworks seeks to enable non-experts to write custom queries, bug checkers, and even entire static analyzers. Examples of such frameworks include Chord [38], Doop [11], Semmle [45], SonarQube [13], and Semgrep [17]. The query languages in these frameworks are predominantly based on extensions of Datalog, such as LogiQL [22], CodeQL [6], Flix [34], Datafun [5], and Formulog [8]. While these frameworks are a step in the right direction, however, they expect users to master complex language schemas, logic programming constructs, and program analysis libraries. Sporq builds atop the program representations and query languages employed by these frameworks to provide an improved user experience while retaining their expressivity, flexibility, and performance.

Synthesis of Logic Queries. A large body of work on Inductive Logic Programming (ILP) has proposed techniques for synthesizing logic queries from relational input-output data. Conventional ILP techniques (e.g. Metagol [36, 37]) require templates (i.e. meta rules) to restrict the space of candidate queries. Modern ILP techniques (e.g. ILASP [30] and FastLAS [29]) use Answer Set Programming (ASP) which scales better by leveraging SAT solvers for the search, but still requires the user to provide syntactic bias in the form of mode declarations (e.g., signatures of invented predicates) to constrain the search space. In general, annotations such as templates or modes require extensive tuning in practice, since too small a search space limits the kinds of queries that can be synthesized, whereas too large a search space makes the synthesis procedure intractable.

Program synthesis techniques such as ALPS [48], Difflog [50], and Prosynth [40] follow the same high-level CEGIS (Counterexample-Guided Inductive Synthesis) architecture as GenSynth: they differ in the search technique used by the generator of candidate queries but are similar in the use of a Datalog solver to evaluate the generated queries. However, with the exception of GenSynth, these approaches also require candidate rules to constrain the search space. A notable exception, EGS [53], does not require candidate rules but is limited to a fragment of Datalog without recursion and negation, and does not scale to large input databases. In fact, even off-the-shelf GenSynth does not scale to our application domain, which motivated our optimizations and the relation pruning discussed in Section 4.2.

Neural learning approaches such as Neural Theorem Proving (NTP) [41], NeuralLP [57], Neural Logic Machine (NLM) [19], and ∂ILP [20] scale well and handle tasks that involve noise or require subsymbolic reasoning. However, they struggle with generalizability and data efficiency. On the other hand, GenSynth is able to generalize effectively from a few labels.

Learning in Program Analysis. Several works have applied machine learning to program analysis, either to synthesize them or to prioritize their results. Bielik et al. apply decision tree learning algorithms to synthesize static analyzers for JavaScript [10]. One approach to finding suspicious entities in the codebase is to phrase it as an anomaly detection problem [2, 58]. Neural networks have been used for program analysis tasks, notably for type inference [3, 23, 55] and invariant generation [42, 47, 49]. Concurrently, there have been efforts to use classification or ranking algorithms to predict which conclusions reached by a static analysis represent true bugs, including that of Koc et al. [26], URSA [59], Bingo [39], and Arbitrar [32]. Learning algorithms have also been proposed to intelligently devote resources to specific parts of the analyzed program [24]. These approaches are orthogonal and complementary to Sporq, for instance, to rank Sporq's alarms by their ground truth or expected payoff in labeling them.

8 THREATS TO VALIDITY, LIMITATIONS AND DESIGN IMPROVEMENTS

In this section, we discuss threats to the validity of our user study and evaluation, and describe the limitations of Sporq that we plan to address in future work.

Threats to Validity. There are several threats, both internal and external, to the validity of our empirical findings. The main threat to internal validity arises because of the small sample size of participants in our user study. Furthermore, the participants are graduate students with previous experience in logic programming. Regular programmers are likely to face an even steeper learning curve while using Semmle and Semgrep. For example, in one of the author's software engineering MOOC on Udacity, only 47 out of 171 graduate students were able to correctly write Datalog queries of difficulty comparable to those in our user study. Lastly, while it is conceivable that users will become more proficient in using tools like Semmle and Semgrep over time, we hypothesize that Sporq will be more usable even for such users, especially for crafting queries that are complex or involve predicates computed by program analyses.

The threats to external validity stem from the fact that our conclusions may not generalize to queries and codebases beyond those in our evaluation. We have mitigated this concern by employing a suite of diverse queries on widely-used programs like the GNU core utilities. Likewise, our evaluation only targets two languages C and C++, but surprises for other languages are mitigated by the substantial overlap of Semmle's schemas across different languages.

Limitations of Sporq. We outline four limitations of Sporq and suggest ways to alleviate them.

(1) *Expressiveness.* GenSynth, and by extension the queries synthesized by Sporq, do not support useful features such as aggregation, pattern matching, and foreign functions. Extending GenSynth with these features does not pose special challenges, but their absence prevents Sporq from synthesizing more complex queries written in Semmle and Semgrep.
(2) *Transferability.* The queries learnt by Sporq on one codebase may not perfectly capture the pattern intended by the user—and therefore may produce unexpected results when applied to other codebases. This is because the user may prematurely stop interacting with Sporq even when it only yields an incomplete pattern. Moreover, even exhaustive interaction could yield a query that overfits the pattern in the original codebase. While avoiding such overfitting is an outstanding challenge in the program synthesis community, GenSynth mitigates this problem by learning concise queries that are likely to generalize better.
(3) *Active Learning.* Sporq does not prioritize the examples it predicts, leaving it up to the user to decide which examples to inspect. As a result, the user could end up labeling more examples than the minimal needed. An active learning algorithm such as Query-by-Committee [46] could be employed to determine examples that prune the synthesizer's search space the most and thereby minimize user annotation burden.
(4) *Imperfect Recall.* While Sporq can achieve perfect precision through user interaction and feedback, it cannot ensure perfect recall. However, this is an open challenge across much of programming-by-example and machine learning. Some recent techniques have been proposed to address this challenge, such as through the use of active

learning [21], or by leveraging background knowledge [54]. Combining Sporq with these techniques suggests exciting directions for future research.

Design Improvements. There are many ways Sporq's user interface could be improved to make interactions more intuitive and efficient. Here we suggest only a few areas worthy of exploration in future iterations.

Currently, Sporq requires a seed example in order to generate an initial query. However, a user might have a target pattern in mind without knowing if there are examples in the current codebase. To generate a matching query, Sporq could give the programmer access to a "sandbox" window where they could type in a toy program containing an instance of the pattern. Sporq's synthesis engine would then generate a query based on this example, which could then be applied to the codebase.

The quality of Sporq's output depends on the quality of user feedback. The multi-modal approach [31] implies that there can be different forms of feedback in each mode which can let the user better express "why" an example is undesired, as opposed to just the fact that it is undesired. In particular, natural language processing techniques could be explored as a means to clarify the user's intent as well as translate the resulting Datalog programs into a format that is more easily interpretable.

9 CONCLUSION

We presented Sporq, an interactive environment for exploring code using a query-by-example approach. Sporq provides a familiar and flexible interface that allows developers to conveniently specify examples of patterns that they wish to mine in their codebases. It uses the examples to automatically learn those patterns by synthesizing database queries over relational program representations. Sporq thereby provides a significantly improved user experience over state-of-the-art tools which require programmers to manually write queries in unfamiliar or inflexible interfaces. Our experiments and user studies demonstrate that Sporq is effective in enabling programmers to discover bugs in large codebases with minimal effort.

ACKNOWLEDGEMENTS

We thank our anonymous shepherd and reviewers for insightful feedback which substantially improved the presentation. We also thank the participants of our user study, and Semmle for helping us understand the internal workings of CodeQL. This research was supported by grants from AFRL (#FA8750-20-2-0501), ONR (#N00014-18-1-2021), and NSF (#2107429 and #2107261).

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. *Foundations of Databases: The Logical Level* (1st ed.). Pearson.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* (2018).

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI).*

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, , and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the International Conference on Management of Data (SIGMOD).*

[5] Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: A Functional Datalog. In *Proceedings of the ACM International Conference on Functional Programming (ICFP).*

[6] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).*

[7] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[8] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (2010).

[10] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a Static Analyzer from Data. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*.

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[12] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *Proceedings of the NASA Formal Method Symposium*.

[13] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action*. Manning Publications Co.

[14] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.

[15] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 332–343.

[16] CWE Community. 2008. CWE 681: Incorrect Conversion Between Numeric Types. https://cwe.mitre.org/data/definitions/681.html.

[17] Return To Corporation. 2021. Semgrep. https://semgrep.dev.

[18] Andrew Cropper, Sebastijan Dumančić, and Stephen H. Muggleton. 2020. Turning 30: New Ideas in Inductive Logic Programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

[19] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. 2019. Neural Logic Machines. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[20] Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *Journal of Artificial Intelligence Research* 61 (2018).

[21] Sara Evensen, Chang Ge, Dongjin Choi, and Çağatay Demiralp. 2020. Data Programming by Demonstration: A Framework for Interactively Learning Labeling Functions. arXiv:2009.01444 [cs.LG]

[22] Todd J. Green. 2015. LogiQL: A Declarative Language for Enterprise Applications. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*.

[23] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[24] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-Aware Program Analysis Via Online Abstraction Coarsening. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[25] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

[26] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools. In *Proceedings of the ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*.

[27] Daphne Koller, Nir Friedman, Sašo Džeroski, Charles Sutton, Andrew McCallum, Avi Pfeffer, Pieter Abbeel, Ming-Fai Wong, Chris Meek, Jennifer Neville, et al. 2007. *Introduction to statistical relational learning*. MIT press.

[28] GitHub Security Lab. 2021. Bounties. https://securitylab.github.com/bounties/.

[29] Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. 2020. FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*.

[30] Mark Law, Alessandra Russo, and Krysia Broda. 2014. Inductive Learning of Answer Set Programs. In *Proceedings of the European Conference on Logics in Artificial Intelligence*.

[31] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M. Mitchell, and Brad A. Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 105–114.

[32] Ziyang Li, Aravind Machiry, Binghong Chen, Ke Wang, Mayur Naik, and Le Song. 2021. Arbitrar: User-Guided API Misuse Detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[33] LLVM. 2021. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[34] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[35] Jonathan Mendelson, Aaditya Naik, Mayur Naik, and Mukund Raghothaman. 2021. GenSynth: Synthesizing Datalog Programs without Language Bias. (2021).

[36] Stephen Muggleton. 1995. Inverse Entailment and Progol. *New Generation Computing* 13, 3 (Dec. 1995).

[37] Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. 2015. Meta-interpretive Learning of Higher-order Dyadic Datalog: Predicate Invention Revisited. *Machine Learning* 100, 1 (July 2015).

[38] Mayur Naik. 2011. Chord: A versatile platform for program analysis. In *Tutorial at the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[39] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided Program Reasoning Using Bayesian Inference. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[40] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-Guided Synthesis Of Datalog Programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*.

[41] Tim Rocktäschel and Sebastian Riedel. 2017. End-to-end Differentiable Proving. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[42] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[43] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[44] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the International Conference on Compiler Construction (CC)*.

[45] Semmle. 2021. https://lgtm.com.

[46] H. S. Seung, M. Opper, and H. Sompolinsky. 1992. Query by Committee. In *Proceedings of the Annual Workshop on Computational Learning Theory (COLT)*.

[47] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[48] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[49] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*.

[50] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs Using Numerical Relaxation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

[51] Y. Smaragdakis and M. Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the Datalog 2.0 Workshop*.

[52] Bjarne Stroustrup and Herb Sutter. 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[53] Aalok Thakkar, Aaditya Naik, Nate Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-Guided Synthesis of Relational Queries. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[54] Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-Driven Synthesis of Provably Sound Side Channel Analyses. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 810–822.

[55] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[56] J. Whaley, D. Avots, M. Carbin, and M. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*.

[57] Fan Yang, Zhilin Yang, and William Cohen. 2017. Differentiable learning of logical rules for knowledge base reasoning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[58] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the USENIX Security Symposium*.

[59] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.